GPU CUDA programming tutorial

Version 0.1 (May 16, 2023)

ALEXANDROS V. GERBESSIOTIS

CS DEPARTMENT

NJIT

NEWARK, NJ 07102.

Email: alexg@njit.edu

Printed on August 22, 2023

Preface

This material is neither final nor thoroughly proofread. It constitutes work in progress and might contain errors. It should be used IN CONJUNCTION with other references if consulted for factual checking.

Report discrepancies with other sources, or factual errors, or typos to the author.

Contents

Part I

Traditional (serial or sequential) computing

Chapter 1

Introduction to computing

1.1 What is a computer

Definition 1.1 (*Computer*)**.** *The fundamental elements of a computer include the following.*

- *The* **processor (CPU)** *also known as CPU for Central Processing Unit,*
- *the* **Main Memory (MM)** *also known as primary memory,*
- *the* **Input/Output modules** *or I/O modules that manage and control I/O and other peripheral devices, and*
- *the* **System Bus** *that is the highway that allows for the communication of the other elements of the computer.*

The name CPU or Central Processing Unit was prevalent in the early era of computing when the CPU needed multiple buildings, or multiple floors or rooms of a building to be housed.

At some point the capabilities of the CPU were miniaturized and fit inside a single chip. We started using the name microprocessor and after dropping the prefix micro, the use of the word processor prevailed. A processor is responsible for managing the computer and its other components and performs a variety of data processing functions. Functions related to arithmetic or logical operations are performed in a (usually separate) section of the microprocessor known as the Arithmetic Logic Unit (ALU). Memory management processing takes place in a area known as the Memory Management Unit (MMU).

Definition 1.2 (*Microprocessor*)**.** *A microprocessor is a CPU or processor that can fit inside a (micro)chip.*

Definition 1.3 (*Processor and its Registers*)**.** *Every processor includes a collection of limited named memory. The named memory inside a processor is referred to as the registers of the procesor.*

Definition 1.4 (*Registers and Register File*)**.** *The most important register is the Program Counter also known as Instruction Pointer (PC or IP). Another imporant register is the Instruction Register (IR), the Accumulator (AC) that collects results of operations in the ALU unit and other general or special purpose registers such as Top of Stack and Bottom of Stack pointers, etc. A special register indicates the status of the processor: it is known as FLAGS or Processor Status Word (PSW). Collectively, all registers are referred to as the register file.*

The typical number of registers is small. Excluding some special purpose register and special purpose CPU architectures (eg GPUs) it is no more than approximately 64. These are the registers that users or user programs can use directly.

The Main Memory consists of a collection of byte(s) and it is random access. We might use the term sequence of bytes to describe. Random access means getting the byte at offset (or address or index) i takes a fixed amount of constant time that is independent of the 'complexity' of the offset i. The first offset (or address) is 0. Memory M can be represented in the form of an array M of bytes. The contents of offset i would be denoted as $M[i]$ or $MM[i]$. Contents is the value contained in address/offset i. Memory stores bytes not bit (an acronym for binary digit). The name byte indicates the minimum about of memory one can 'bite' from the main memory of a computer. The unit for a byte is B and for a bit is **bit**. Aggregations of byte that will be used include KiB for kibi-byte, MiB, GiB and TiB equal to $2^{10}B$, $2^{20}B$, $2^{30}B$, and $2^{40}B$. The length of M is the number of byte its contains. The size of M includes also the unit i.e. B. Thus we can say the length of M is 8 meaning that the size of M is $8B$.

The Program Counter holds the address (in MM) of the next instruction that will be executed. If the instruction is spread over more than one byte, this would indicate the address of the first byte of the instruction. Intel architectures are little-endian which means this would be the right-most (also known as least significant byte) of the instruction.

1.1.1 Simple Instruction cycle

A traditional (simplified) processor of the works in essence by repeating the execution involved in a simple instruction cycle. An instruction cycle has two stages (if there is no support for interrupts, or interrupts are disabled) or three stages (otherwise). The first stage is the fetch stage (the CPU fetches the instruction from memory and brings it into main memory). The second stage is the execute stage where the CPU executes the instruction and thus performs the data processing task indicated by the instruction. By the end of this stage the instruction's execution is completed. Yet there is often a third stage that can be utilized. In that third stage known as interrupt stage, the CPU checks for a hardware signal sent primarily by an external (and usually) I/O device. That signal might indicate the completion of an I/O operation, a hardware condition or a hardware problem. If it is generated by a timer it might mean that a given period of time has been completed and came to a conclusion. For example at midnight one sets a timer to generate an interrupt in 18,000sec, so that an alarm clock would be triggered at 5am.

Fetch stage of instruction cycle

The first stage is the **fetch** stage and involves fetching an instruction from the address in main memory as indicated by the program counter. Thus the PC already contains the address of this instruction that is to be fetched and executed. If one wonders what happens with the 'first instruction' of a user program and what the value of the PC is, one may assume that it either contains a 0, thus assuming the address of the first instruction is 0, or that the Operating System (OS) program that loaded the user program into main memory (from, sually a secondary memory device such as a hard disk drive), set the PC to some specific address used by the OS.

The instruction fetched is then stored into the Instruction Register (IR). By the conclusion of this stage the PC already points to the address of the next instruction to be executed (eg the new address is the old one plus one byte of its previous value if the instruction fetched is one byte or plus the number of byte of the instruction just fetched). Moreover, during the next stage, the execute state, it is possible that the PC can be altered by an appropriate PC altering instruction. For example a JUMP instruction can set the value of the PC to some alternative specific address.

Execute stage of instruction cycle

In the second stage that is known as the **execute** stage, the processor "executes" the instruction that was made available in the instruction register in the prior fetch stage.

Instruction execution might involve reading from MM (a load operation from MM into a CPU register), or writing into MM (a store operation from a register into one or more MM bytes), or arithmetic or logical operations that take place in the ALU or other data processing operations. For example a JUMP to an address instruction can change the PC address to a new value. A HALT instruction can freeze the CPU. And a NOOP (no operation) instruction can do nothing (thus bringing an end to the execute stage).

Interrupt stage of instruction cycle

If there is a third stage this is the **interrupt** stage. The CPU then and only then checks an interrupt vector to determine if an I/O device that had an I/O operation active asked the CPU to interrupt the normal execution of the CPU itself.

Prior to the interrupt stage of an instruction, the CPU was involved in the execution of some user's code (say User Program A).

This was done at a low security level (user code cannot interfere with critical code). Some architectures refer to it as user mode or RING LEVE 3 (Intel).

Context switch from *A* to *OS*: preparing by saving context of *A*. As soon as the CPU realizes the interrupt request, it stops the execution of program A. This is also known as user context. The state of the user's program is the values of all its registers (that were modified during the execution of A). Some or all of these registers would form the register file that will be saved, and program A's execution will temporarily stop.

Context switch from *A* to *OS*: interrupt handling routine. Control will pass to an interrupt handling routine that will service the interrupt in question. The CPU will start executing the code of the interrupt handling routine. Prior to that the registers of the CPU (i.e. their values) associated with program *A* need to be reset and initialized for the interrupt handling routine to start its execution (more precisely the CPU to start executing the code of the interrupt handling routine).

The interrupt handling routine belongs to or is a program that used to be known as (system) monitor, then became known as operating system, and later as kernel (the part of the operating system that is in main memory all the time). When the code of the Interrupt Handling routine is running on the CPU (i.e. the CPU is executing instructions of the interrupt handling routine rather than Program A) the CPU is not in user mode (RING LEVEL 3) anymore but is (operating) system mode (may be RING LEVEL 0).

Context switch from *OS* to *A*: resuming the execution of Program *A*. When the interrupt is serviced, the CPU moves back to user mode, the register file of Program A gets reinstated into the CPU (the values of the registers reflecting the execution of Program A are copied from Main Memory and set the values of the CPU registers) and then program A executes the next instruction that was to execute as if the interrupt stage had been ignored (that instruction's address is already in the PC if not by the end of the fetch stage, definitely by the end of the execute stage of the instruction that was executed prior to the interrupt checking).

Context Switch support: OS

We skip Operating System (OS) duscussion. In the presence of a complex OS the area of Main Memory associated with the execution of Progam A is controlled by the kernel of the OS. A Program in Execution such as A is known as a Process. Every process has an area in main memory belonging to the operating system and controlled by the kernel that is known as the process control block (PCB) or task structure (TS). The PCB/TS has a area to accommodate the register file of say Program A, and to indicate whether A is RUNNING on the CPU (i.e. CPU is running the code of A) or not (during interrupt handling).

1.1.2 CPU and ALU

The ALU is where operations such as ADD (for addition), MUL (for multiply), DIV (for divisions) are performed and also XOR (for exclusive OR), OR (for inclusive OR or disjunction), AND (for a conjunctions), etc are performed.

Let us remind ourselves some definitions for programming languages. Several time we use the symbol $+$ and we call it operator, to denote the operation known as addition. The elementary addition operation has two operands x and y to which the operator applies. The result is the sum of the values of the two operands $x+y$. The form $x+y$ is an infix representation of addition where the operator is in-between the left operand x and the right operand y. In a prefix representation we would write $+xy$ and in a postfix representation $xy+$ instead. Parentheses could be used for clarity and to indicate order of evaluation. When multiple operators and operands are present different rules specify the precedence of operators and in case of a tie what the tie-breaking rule(s) are going to be.

After this long parenthesis involving the CPU let us move forward.

1.1.3 Processor and hardware

Definition 1.5 (*Motherboard*)**.** *The microprocessor, main memory, the system bus and the I/O modules are arranged and attached to a motherboard that houses them.*

Definition 1.6 (*Socket*)**.** *The microprocessor is attached to a socket, a housing area or receptacle of the motherboard that allows for easy attachment or detachment of the microprocessor.*

Definition 1.7 (*Microchip and Die*)**.** *The microprocessor is a CPU miniaturized on a (micro)chip. The microchip contains one or more dies. A die is a, silicon based usually, integrated circuit covered with epoxy inside a plastic or ceramic housing with gold plated connectors that attach on a housing or receptacle (socket) of a computer's motherboard.*

The microchip housing the CPU can fit on the receptable that a socket is.

Definition 1.8 (*SoC*)**.** *A SOC is a System On a Chip, a computer housed on microchip.*

Definition 1.9 (*Register file*)**.** *The collection of registers associated with an execution unit known as the CPU (microprocessor, or simply processor). One of the registers is the Program Counter (PC).*

Definition 1.10 (*PC*)**.** *PC would read as Program Counter. In very rare cases it would denote a Personal Computer but this would be quite clear from the context.*

1.2 More and more Processors: Multi

A CPU or a microprocessor can be referred to as an execution unit. An execution unit is characterized by the register file of the CPU and is abstracted by the most important register of all, the Program Counter (PC). Thus in early architectures one CPU mapped to one execution unit, and a PC (Personal Computer now) contained one CPU, i.e. one execution unit. Therefore an execution unit used to imply one PC associated with it, containing one CPU.

Definition 1.11 (*Early microprocessor*)**.** *An early computer era microprocessor was one execution unit containing thus one PC (Program Counter) as part of the register file.*

Definition 1.12 (*Uniprocessor*)**.** *Uniprocessor means one processor. It refers to a processor with one execution unit (one PC of one register file). Sometimes it is interpreted to mean Unicore processor.*

Definition 1.13 (*Core*)**.** *A core is an execution unit inside a microprocessor, i.e. what makes and defines a microprocessor.*

Early microprocessors had one execution unit i.e. one core. Thus the term microprocessor and its execution unit meant the same thing. The latter term was not much in use.

We started using the term core and execution unit when we started having multi-core processors. We use then the term core to refer to the execution units inside a microprocessor that is then defined as a multi-core processor.

The number of transistors in modern processor architectures can range from about a billion to 5 billion or more (Intel Xeon E5, Intel Xeon Phi, Oracle/Sun Sparc M7).

A **chip** is the package containing one or more **dies**, actual silicon integrated circuits (IC) that are mounted and connected on a processor carrier and possibly covered with epoxy inside a plastic or ceramic housing with gold plated connectors. A **die** contains or might contain multiple cores, a next level of cached memory adjacent to the cores (e.g. L3), graphics, memory, and I/O controllers. If L3 cache is adjacent to the cores, this also means that L1 cache or L2 cache is in the cores as well.

Definition 1.14 (*Multi-core Processor*)**.** *A multi-core (or many-core) processor is a microprocessor with many execution units (each one of them having a PC or equivalently associated with a register file) in it. The execution units of a multi-core processor are known as cores.*

From now on we drop the hyphen and write multicore processor. The term manycore processor is to be used only when the number of cores is too large. Then we will define what ”too large” means.

Definition 1.15 (*MultiComputer*)**.** *A multicomputer (system) is a system consisting of more than one computers.*

It can be a non-homegeneous collection of computers or a homogeneous one. Think about one computer having a plain processor, a processor that is a uniprocessor. Another computer having a multicore processor. And so on.

Definition 1.16 (*Multiprocessor*)**.** *A multiprocessor is a computer with more than one processors.*

We could even equivalently say that a multiprocessor is a computer with more than one execution units.

Definition 1.17 (*Symmetric Multiprocessor*)**.** *A symmetric multiprocessor (SMP) is a computer with more than one processors that are all of the same type (and all sharing the same shared main memory).*

Definition 1.18 (*Multiprocessor System*)**.** *It can refer to a Multiprocessor, or a Multicomputer, or a Symmetric multiprocessor.*

Definitions

ExecutionUnit	=	Data Processing Unit	$\wedge$	(Has) RegisterFile
MicroProcessor	$=_{<2000}$	One ExecutionUnit	$\wedge$	InOneChip
Processor	$=_{<2000}$	MicroProcessor		
Core	=	One ExecutionUnit	$\wedge$	$\exists Processor : Core \in Processor$
MulticoreProcessor	=	$\#Processor = 1$	$\wedge$	$\#ExecutionUnit > 1 \wedge InOneChip$
MulticoreProcessor	$\overset{Alternative}{=}$	$\#Processor = 1$	$\wedge$	$\#Core > 1$
Uniprocessor	=	$\#Processor = 1$	$\wedge$	$\#ExecutionUnit = 1$
UniProcessor	$\overset{Alternative}{=}$	Processor(< 2000)		
(Unicore) Processor	=	Uniprocessor		
Processor	$=_{>1999}$	UniProcessor	$\vee$	MulticoreProcessor
Multicomputer	=	$\#Computer > 1$		
SymmetricMultiprocessor (SMP)	=	$\#Computer = 1$	$\wedge$	$\#Processor > 1 \wedge SameType$
Multiprocessor (< 2000)	=	$\#Computer = 1$	$\wedge$	$\#Processor > 1$
Multiprocessor (> 1999)	=	$\#Computer = 1$	$\wedge$	$\#ExecutionUnit > 1$
MultiprocessorSystem	=	$Multicomputer$	$\vee$	$SMP \vee Multiprocessor$

1.3 Software front: Tasks

We use terminology from the Unix-like operating system referred to as Linux.

Definition 1.19 (*Task*)**.** *A task is a process or a thread.*

Definition 1.20 (*Process*)**.** *A process is a program in execution; the OS knows about it. The kernel of the OS maintains in data structures information about a process. A process has a PC (program counter) in its register file, a User and Kernel stack, and of course text, initialized and uninitiliazed global variables, a heap, and shared memory space (optional).*

Definition 1.21 (*Thread*)**.** *A thread is a lightweight process. A thread is created inside a process. A thread has a PC, User and Kernel stack and shares with the process in which it was created the text, initialized and uninitiliazed global variables, the heap, and shared memory space (optional).*

In Linux a task for which getpid() and gettid() return the same values is a process. If gettid() is different from getpid() the task is a thread created within the process of the given getpid().

Definition 1.22 (*Thread of execution*)**.** *A thread of execution is a task defined by its PC (and register file).*

A thread of execution can be that of a process or the threads of some process.

Example 1.1. *A process with PID 10 created three threads. There are four threads of execution, one is that of the process (sometimes referred to as the primary thread), and three are of the three threads created within a process.*

In Linux all four are called tasks. They have individual entries in the task table that contains task structures. So the Linux task table combines a PCB (Process Control Block) and a TCB (Thread Control Block) functionality.

1.3.1 Multiprocessing and Multithreading

Definition 1.23 (*Multithreaded Process*)**.** *A multithreaded process is a process with more than one threads of execution. One is the thread of execution of the process itself that is defined by it PC (and register file), and the others are threads of execution of threads created within that process.*

Definition 1.24 (*Multiprocessing*)**.** *It is an approach or framework that includes two or more processes each one with a single thread of execution defined by its PC (and register file).*

A multiprocessor is a piece of hardware; a multiprocessing framework is the splitting of an application in more than one programs whose execution generates more than one processes.

Definition 1.25 (*Multithreaded Multiprocessing*)**.** *It is an approach or framework that includes two or more processes each one with a single or more threads of execution each one defined by its PC (and register file).*

Processing and Threading Definitions

(Plain) Process	=	$\#Process = 1$	$\wedge$	$\#\text{ThreadofExecution} = 1$
Multithreaded Process	=	$\#Process = 1$	$\wedge$	$\#\text{ThreadofExecution} > 1$
Multiprocessing	=	$\#Process > 1$	$\wedge$	$\#\text{ThreadofExecution}/\text{Process} = 1$
Multithreaded Multiprocessing	=	$\#Process > 1$	$\wedge$	$\#\text{ThreadofExecution} > \#Process$

The last definition is too loose. We could claim that the threads of execution in each process (per process) is at least two. It suffices if in just one process this would be true. Thus we could replace a $\#\text{ThreadofExecution}/\text{Process} \geq 2$ or $\#\text{ThreadofExecution}/\text{Process} > 1$ with a $\#\text{ThreadofExecutionofAllProcesses} > \#Process$. The latter using the pigeonhole principle establishes that a process will have at least two threads.

1.4 Memory lane(s)

Definition 1.26 (*Register*)**.** *A register is small named memory inside a CPU.*

Definition 1.27 (*Cache*)**.** *A cache is a specially structured memory that is slower than registers but faster than Main Memory.*

Definition 1.28 (*Main Memory (MM)*)**.** *It is the main memory of a computer; also known as primary memory.*

There are two fundamental CPU operations (instructions) related to Main Memory: a read and write operation. If we conceptualize Main Memory as an array M of bytes, a read operation accepts one argument, an address A of MM, and its output is the contents $M[A]$ of that address A. The contents are returned to the CPU. A write operation accepts two arguments, an address A and a byte value t. The effect is to set the contents of address A of MM to t that is $M[A] = t$. The communication between the CPU and MM are done through a set of register such as MAR and MDR for Memory Address Register and Memory Data Register respectively (sometimes MDR is referred to as MBR, the Memory Buffer Register). Thus in a read operation MAR is set to A and at the conclusion of the read step $M[A]$ is available from MDR. In a write operation MAR is set to A and MDR is set to t.

Main Memory sometimes is referred to as RAM for Random Access Memory. Such a characterization means that accessing byte $M[0]$ is no more expensive than accessing byte $M[0x002F3F4F]$, i.e. $M[i]$ takes constant time to retrieve.

The main memory is external to the die that forms the multi-core processor. Latency is 32-128cycles (60-110ns) and bandwidth 20-128GB/s (DDR3 is 32GiB/sec).

Moreover RAM (main) memory can be of two varieties. A DRAM (Dynamic RAM) uses 1 transistor per bit and 1 capacitor. It is cheap, it needs a periodic refresh (every millisecond or so) and it is thus power consuming and after every read operation, the read bit needs to be rewritten. It is volatile and has minimal persistence (1-10s). Further more it is EMI (Electro Magnetic Interference) sensitive. A SRAM (Static RAM) uses 4-6transistors per bit, it is x10 to x20 faster than DRAM. No refresh is needed and can be non-volatile but is usually volatile with some remanence (preservation of data) It is non EMI sensitive and it is being used for MM in space apps or caches in CPUs. This makes SRAM 10x to 100x more expensive than DRAM.

In a pyramid that represents the memory hierarachy, the registers are at the top, followed by the cache(s), followed by MM, and then followed at the (or close to the) bottom of the hierarchy by other types of memory.

Definition 1.29 (*Secondary Memory*)**.** *Secondary memory is memory other than MM that requires communication through the I/O modules.*

"Other than MM" implies an "other than MM and lower in the memory hierarchy". Such memory includes several disk drives such as a floppy disk drive (FDD), a hard disk drive (HDD), a CD-ROM drive, etc and also a solid state drive (SSD), tape drives, etc.

A cached memory is a very fast memory. Its physical proximity to the CPU (or core) determines its level. Thus we have L1 (closest to the CPU, in fact "inside" the CPU), L2, L3, and L4 caches. The latter L4 might be available as a DRAM attached to a graphics unit (GPU) on the CPU die (e.g. Intel Iris). There are references to an L0 cache which is much smaller in size than the L1 and is being used by the CPU to store microcodes (microinstructions) outside of the registers.

Definition 1.30 (*Memory attributes*)**.** *Memory has three attributes: cost, speed and size.*

Prefix	Name	Multiplier
d	deca	$10^1 = 10$
h	hecto	$10^2 = 100$
k	kilo	$10^3 = 1000$
M	mega	10^6
G	giga	10^9
T	tera	10^{12}
P	peta	10^{15}
E	exa	10^{18}
d	deci	10^{-1}
c	centi	10^{-2}
m	milli	10^{-3}
μ	micro	10^{-6}
n	nano	10^{-9}
p	pico	10^{-12}
f	femto	10^{-15}

Figure 1.1: SI system prefixes

Prefix	Name	Multiplier
Ki	kibi or kilobinary	2^{10}
Mi	mebi or megabinary	2^{20}
Gi	gibi or gigabinary	2^{30}
Ti	tebi or terabinary	2^{40}
Pi	pebi or petabinary	2^{50}

Figure 1.2: SI binary prefixes

Definition 1.31 (**Bit**)**.** *The word* **bit** *is an acronym derived from* **bi**`nary digit` *and it is the minimal amount of digital information. The correct notation for a bit is a fully spelled lower-case* **bit***.*

Definition 1.32 (**Byte**)**.** *A* **byte** *is the minimal amount of binary information that can be stored into the memory of a computer and it is denoted by a capital case* **B***.*

Etymologically, a `byte` is the smallest amount of data a computer could `bite` out of its memory! Nowadays, **1B** is equivalent to 8bit; it was not like this in the early years of computing. Some architectures were using 4bit, or 6bit for a what is known nowadays as a byte.

Definition 1.33 (**Word).** *Word is a fixed size piece of data handled by a microprocessor. The number of bit or sometimes equivalently the number of byte in a word is an important characteristic of the microprocessor's architecture.*

A 32-bit architecture has word size 32 bit.

Definition 1.34 (**Octet).** *A sequence of 8 bit. Nowadays it is a synonym (alias) for a byte.*

We never store bits in memory (as in main memory): we can only store a byte in it that contains the bit in question. Thus to store a bit we embed it into a byte, store the byte, and we need to remember which of the eight bit of the byte is our stored bit when later we retrieve the full byte with the intent of accessing the stored bit! Thus in order to store one bit we waste the 7 remaining bit of a byte.

Definition 1.35 (**Memory size). Memory size** *is expressed in bytes or its multiples.*

We never talk of 8,000bit memory, we prefer to write 1,000B rather than 1,000byte, or 1,000Byte.

Prefix	**Name**	**Multiplier**
1KiB	1kibibyte	$2^{10}B$
1MiB	1mebibyte	$2^{20}B$
1GiB	1gibibyte	$2^{30}B$
1TiB	1tebibyte	$2^{40}B$
1PiB	1pebibyte	$2^{50}B$

Figure 1.3: SI aggregates of a byte

Name	**Multiplier**
1 short	2B = 16bit
1 word	4B = 32bit
1 double word	8B = 64bit

Figure 1.4: Other aggregates of a byte

Definition 1.36 (*Memory Latency*)**.** *Memory latency is the time it takes to retrieve one byte from memory.*

Definition 1.37 (*Memory Throughput*)**.** *Memory throughput is the times it take to retrieve additional, sometime closely located, byte(s).*

An HDD has memory latency of 5-20ms. Memory throughput is more than 20,000,000B/s. Main memory latency is 80-120ns. Throughput is 100GiB/s

Memory Speed

Level	Typical Size	Latency
CPU register	8-16	1 cycle
L1D or L1I cache	8-32KiB	1-5 cycle
L2 unified cache	256-4096KiB	15 cycle
L3 unified cache	4-128MiB	45 cycle
SRAM	1-16MiB	L0-L2 level
DRAM	several GiB	45 cycle + 80ns
SSD	hundred GiB	0.1-0.2ms
HDD	several TiB	5-20ms

1.5 Cached Memory

A cached memory is not as efficient as the on-chip registers; yet it is faster than main memory. It is more abundant than the register file but its size is not as large as that of main memory.

Its existence allows for more effective memory transfer rates thus increasing a processor's performance since small segments of memory are speculatively fetched from main memory on the expectation that a program code will exhibit **temporal** or **spatial** locality.

1.5.1 Temporal locality

Definition 1.38 (*Temporal Locality*)**.** *Temporal locality is the propensity of a piece of code to reuse recently executed segments of program or data (or both).*

For a example, in a C-like for-loop structure such as **for(**$i = 0$**;**$i < n$**;**$i++$**)**$\{$**body**$;\}$, we expect the data associated with variables i and n to be continuously retrieved and used during the execution of the for loop. This is data temporal locality. Moreover we expect instructions such as $i < n$ or $i++$ or **body** to be continuously executed during the execution of the for loop.

1.5.2 Spatial locality

Definition 1.39 (*Spatial Locality*)**.** *Spatial locality is the propensity to access program/data close to program/data that were recently retrieved.*

Thus if the instruction $i < n$ has been retrieved (and executed), we reasonably expect to have the instructions of **body** to execute then followed by the instructions associated with $i++$. Furthermore if **body** expands to (the code maps to the C statement) **sum**$+= M[i]$, we can reasonably expect that access to $M[500]$ would follow (in the next iteration) by an access to $M[501]$ and may be $M[502]$ and so on. This is data spatial locality of stride 1 (memory address 500, followed by memory address 501, followed by 502, etc). One of course also observes the data temporal locality associated with variables i but also variable **sum**. The temporal locality associated with instruction $+=$ is also clear.

```
 Little endian architectures
Bit  31..24..23..16..15...8..7 .. 0
     Byte31  Byte30  Byte29  Byte28       28 Byte Offset
                ...
     Byte7   Byte6   Byte5   Byte4         4 Byte Offset
     Byte3   Byte2   Byte1   Byte0         0 Byte Offset
```

1.5.3 Cached memory and its levels: L1

L1 cache is usually exclusive (private) to a core, and can be unified or not. In case it is not unified there is a separate L1 cache for instructions and a separate L1 cache for data. They may or not have the same size and the same form and properties.

Intel CPUs have separate L1D (Data L1) and L1I (Instruction L1) caches for data and instructions respectively. This is also the case for modern ARM CPUs. The (combinded L1D and L1I) size of an L1 cache ranges from an 8KiB (in the 1980s) to 96KiB.

For non-unified L1 caches L1D and L1I used to be of the same size. However in recent Intel architectures (e.g. Ice Lake) L1D increased to 48KiB from 32KiB and L1I remained 32KiB, and this explains the 80KiB combined size. Such sizes are per core.

Note that for some AMD architectures L1I is larger in size than L1D (64KiB vs 16KiB for Bulldozer, or 64KiB vs 32KiB for Zen), and this was also the case, at least around 2008, for some Intel Atom architectures as well (32KiB vs 24KiB).

The size of an L1 is determined by studies with the intent to maintain a hit ratio of at least 90% to 95%. By hit ratio we mean the fraction of memory accesses of a program that can be retrieved through (are found in) L1.

They are implemented using SRAM and latency to L1D is 4 cycles in the best of cases; transfer rate is 32-64B/cycle. Note that if L1D data is to be copied to other cores this might take a longer 40-64 cycles.

L1 caches used to be write through. When a datum is modified it is modified in both L1 and Main Memory (MM). In recent years L1 caches are write back: only L1 is updated but not MM. This makes them more efficient than those using a write through mechanism.

1.5.4 Cached memory and its levels: L2

An L2 cache is larger in size than L1 (typically 8x to 16x larger than L1D). Sometimes two or more cores share an L2; this used to be the case in early architecture that lacked an L3 cache. Most Intel and AMD general purpose architectures use private L2 caches (one per core).

An L2 cache can be inclusive (older Intel architectures e.g. Intel Nehalem) or exclusive (AMD Barcelona) or neither inclusive nor exclusive (Intel Haswell).

Inclusive means that the same data will be in L1, L2 (and L3). Exclusive means that if data is in L2, it can't be in L1 and L3. Then if it is needed in L1, a cache "line" of L1 will be swapped with the cache line of L2 containing it, so that exclusivity can be maintained: this is a disadvantage of exclusive caches. Inclusive caches contain fewer data because of replication. In order to remove a cache line in inclusive caches we need only check the highest level cache (say L3). For exclusive caches all (possibly three) levels need to be checked in turn. Eviction from one requires eviction from the other caches in inclusive caches.

L2 caches are usually coherent; changes in one are reflected in the other ones. In some architectures (e.g. the obsolete Intel Phi), in the absence of an L3 cache, the L2 caches are connected in a ring configuration thus serving the purpose of an L3.

The latency of an L2 cache is approximately 12-16 cycles (3-7ns), and up to 64B/cycle can be transferred (for a cumulative bandwidth over all cores as high as 1000-1500GB/s). Note that if L2 data is to be copied to other cores this might take 40-64 cycles.

When neither L1 nor L2 can accommodate a piece of datum, one need to be evicted to make room for a new one. The evicted piece moves to a third level cached memory, L3.

1.5.5 Cached memory and its levels: L3

This is because a larger L3 cache is shared among all or a fraction of the cores of a processor. Accessing a piece of datum in L3 becomes non-uniform as the L3 cache might contain data from different cores in different states (read vs write).

In Intel's Haswell architecture, there is 2.5MiB of L3 cache per core (and it is write-back for all three levels and also inclusive).

In Intel's Nehalem architecture L3 contained all the data of L1 and L2 (i.e. $(64+256)*4$KiB in L3 are redundantly available in L1 and L2).

Thus a cache miss on L3 implies a cache miss on L1 and L2 over all cores! It is also called LLC (Last Level Cache) in the absence of an L4 of course. An L3 cache can also be exclusive or somewhat exclusive cache (AMD Barcelona/Shanghai, Intel Haswell).

Data evicted from the L1 cache can be spilled over to the L2 cache (victim's cache). Likewise data evicted from L2 can be spilled over to the L3 cache. Thus either L2 or L3 can satisfy an L1 hit (or an access to the main memory is required otherwise).

In AMD Barcelona and Shanghai architectures L3 is a victim's cache; if data is evicted from L1 and L2 then and only then will it go to L3. Then L3 behaves as in inclusive cache: if L3 has a copy of the data it means 2 or more cores need it. Otherwise only one core needs the data and L3 might send it to the L1 of the single core that might ask for it and thus L3 has more room for L2 evictions.

The latency of an L3 cache varies from 25 to 64 cycles and as much as 128-256cycles depending on whether a datum is shared or not by cores or modified. Data throughput is 16-32B/cycle. The bandwidth of L3 can be as high 250-500GB/s (indicative values).

1.5.6 Cached memory and its levels: L4

In 2016, a Level-4 cache memory was made available available in some architecture (Intel Haswell) as auxiliary graphics memory on a discrete die. It was uding DRAM technology.

It used to be 128MiB in size, with peak throughput of 108GiB/sec (half of it for read, half for write). It is a victim cache for L3 and not inclusive of the core caches (L1, L2). It has three times the bandwidth of main memory and roughly one tenth its memory consumption. A memory request to L3 is realized in parallel with a request to L4.

1.5.7 Cache Characteristics

As mentioned earlier, **write-back** or copy-back cache is one when a write into L1 is not immediately copied into a higher level memory. Only when the datum is to be discarded from L1 is it copied into L2, or discarded from L2 (or L3) is it then copied into L3 (or main memory) respectively.

In a **write-through** cache a write into L1 also causes a write into higher memory hierarchies as needed.

In a **write-miss** cached either a **write-allocate** is performed when a write miss causes the loading of a block/line into the cache and then the writing occurs in the cache, or **no-write-allocate/write-around** when the data is modified in main memory but not in cache.

1.5.8 Cache Structure

Let x be the number of bit of an architecture that determines the number of bit of an address. For 32-bit architectures $x = 32$; for 64-bit architectures though nominally $x = 64$, the x86-64 addressing scheme uses only 48bit and thus $x = 48$.

Cache lines

For the moment let us assume that one set is one cache line.

Definition 1.40 (*Cache: sets or lines*)**.** *A cached memory contains a number of say C sets and each such set contains one or more "cache lines".*

Definition 1.41 (*Number of sets or lines C*)**.** *Parameter C is referred to as the number of line, or equivalently, the number of sets of the cache. It is a power of two, and thus* $\lg C$ *is an integer. Let* $c = \lg C$.

An index in the range $0 \dots C-1$ is used to describe an index or offset of a line of the set. Such an index uses $c = \lg C$ bit.

Definition 1.42 (*Line: A block of byte plus more*)**.** *A cache line stores a block of K bytes. Thus K is defined as the block length or block size; the term line size can also be used.*

Definition 1.43 (*Block length or size K*)**.** *Parameter K is referred to as the block length or equivalently, the block size but the unit of memory size B must be appended then. Rarely will it be referred to as line size. It is a power of two, and thus* $\lg K$ *is an integer. Let* $k = \lg K$.

An index in the range $0 \dots K-1$ is used to describe an index or offset of a byte in a block of a line of the cache. Such an index uses $k = \lg K$ bit.

Definition 1.44 (*Tag, Block index, Line index*)**.** *A cache line in addition to a block of K bytes that it stores, it also stores a tag. The tag, the line index, and block index identify a byte in main memory and thus forms its address. Conversely, the address in main memory of a byte in the cache can be formed by the union of the bit of the tag, the line index, and the block index.*

Definition 1.45 (*Number of bit of tag t*)**.** *The number t of bit of the tag is given as follows*

$$x = t + c + k \Rightarrow t = x - c - k = x - \lg C - \lg K$$

The set address itself consists of a tag and (possibly of) a line index (offset). The line index of the set address is used to identify (index) a set in cache and thus a "cache line".

Definition 1.46 (*Control/Flag bit of a line*)**.** *A cache line in addition to the tag bit (t) and the byte of the block (K byte), it also contains a number of control (flag) bit. Let r be the number of control bit of a line of the cache.*

The flag (control) bits identify whether the cache-line is available to be replaced (not-valid) or not (valid) or how long data have been stored in it. A cache-line that is not valid can be replaced; the block of its K bytes gets replaced, not individual bytes within that block. How long data have been using a cache-line depends on the policy of replacing data in it (e.g. Least Recently Used).

The structure so defined is a two-dimensional structure of byte containing C rows of K bytes. We can view it as a $C \times K$ matrix of bytes (ignoring tag bit). The nominal size of this structure (in B) is $C \times K$ bytes. The nominal size of this structure (in bit) is $C \times K \times 8$ bit. The actual size of this structure (in bit) is $C \times K \times 8 + C \times (t + r)$ bit.

Definition 1.47 (*Direct Mapped Cache or One-way Associative cache*)**.** *A cache such as the one described is known as a Direct Mapped Cache or One-way Associative cache*

Definition 1.48 (*Set associative cache*)**.** *A cache containing A copies of a One-way Associative cache, is called a set associative cache. The parameter A is the associtivity of the cache, the number of ways, or the number of blocks per set.*

Definition 1.49 (*Associativity A of a cache*)**.** *The associativity or number of ways A of a cache is the number of blocks (along with their common tag and control bit) associated with a given line (index).*

A 4-way associative cache contains 4 copies of a One-way associative cache. A does not need to be a power of two. It is not uncommon in L3 caches to see A values of 13 and 15.

Cache associative greater than one allows us to store in the cache two bytes whose addresses have identical bit for a line index and possibly the same bit for a block index (offset). In a one-way associative cache we can only store those two byte in the cache if their tag bit are the same. If they are different we can only store them in the cache in case $A > 1$. One would be stored in a block of a line of one set, and the other would be stored in a block of a line of another set. Both lines are to have the same line index (offset).

Example 1.2. *Consider a byte with 32-bit address in hexadecimal* **0x4FA12345**. *Let* $K = 16$, $A = 4$, *and let* $C = 4096$. *Then* $c = \lg C = 12$, $k = \lg K = 4$ *and thus* $t = 32 - c - k = 12bit$. *12,4, and 12 bit map to 3, 1, and 3 hexadecimal digit. Starting from right to left, the block index (offset) is* **0x5**, *the line index (offset) is* **0x1234**, *and the tag is* **0x4FA**,

Example 1.3. *Consider a Pentium-4 L1D 4-way associative cache, with cache size 8KiB, and 64B block size. Considering that Pentium-4 is a 32-bit architecture, reading out the rest of the information we gather that* $A = 4$, $K = 64$ *and thus* $k = 6$, *and* $C \times K \times A$, *the nominal size of the cache is equal to 8KiB. Solving for C we obtain* $C = 32$ *and thus* $c = 5$. *Assuming an 8-bit flag per line, we can conclude that the nominal size of the cache in bit is* $8KiB \times 8bit/B = 65536bit$ *but the real cache size is* $69248bit$.

Example 1.4. *In order to determine whether a byte of address* **0x4FA12345** *resides in the cache we first determine the tag, the line index and the block index (offset). They are for the currenly discussed example, starting from the right of the address,* **0x5** *for a block index,* **0x1234** *for a line index, and* **0x4FA** *for the tag. We then go to the cache line index* **0x1234** *and we check in all four copies of the set for that line whether a tag includes tag number* **0x4FA**. *If none of the four sets contains the tag number the byte with address* **0x4FA12345** *is not in the cache. If however the tag is found in a set, we access the byte indexed* **0x5** *of that set, i.e. the sixth byte. Its contents map to the contents of the byte of MM with address* **0x4FA12345**. *If we retrieve the contol (flag) bit, we might detect that the byte is "dirty" i.e. modified. This means the cached byte contains the currently correct value, and MM might contain a previous value. It also indicates that the cache is a write-back cache.*

Therefore a variety of memory levels imposes a variety of requirements or access patterns. Some memory levels are exclusive to a particular core, whereas others (including the shared memory) are shared. Thus access patterns become important in figuring out what the cost of accessing a memory level will be. Needless also to say that the increasing memory level and size complicates memory modeling even more. Whether a level is exclusive or shared among cores becomes hardware and architecture-dependent. A model that makes some explicit assumptions and works well for some architectures might fail miserably in modeling others. Thus for a model to be useable (i.e. it is simple enough for people to use it in analyzing program behavior) and useful (i.e. the behavior it predicts does not contradict observed results) a certain abstraction of hardware intricacies is desirable.

Sometimes architects can only afford these extra transistors and power dissipation that becomes the cache by reducing the clock speed per core. Although as a whole speeds are higher, individual core speeds might become slightly lower. This might affect memory transfer rates as well and impose certain synchronization issues. Thus moving from one generation to the next, modeling issues become much more complex and less straightforward.

1.6 Intel Dunnington

At approximately 2 billion transistors, it accommodates six Core 2 cores at clock speeds between 2.13 and 2.67GHz. Its 64-bit architecture allows access to 3MiB of L2 cache per pair of cores for a total of 9MiB for the six cores. A L3 cache is shared by all six cores providing an additional 16MiB of cache memory. All-in-all 25MiB of fast access memory is available to the six cores. Recently, all L2 caches are exclusive to cores and not shared and smaller in size. L1 caches are also smaller in size (32KiB for data, 32KiB for instructions) than the 96KiB of Dunnington.

Part II

Parallel computing

Chapter 2

Architecture Taxonomies

2.1 Taxonomy based on address-space

2.1.1 Distributed memory

Definition 2.1 (*Distributed Memory*)**.** *In a multicomputer or multiprocessor system each computer has its own memory and makes it available to other processors or execution units of the multicomputer/multiprocessor. The time it takes to retrieve one byte of main memory it depends on the distance between the execution unit and the byte in question.*

It is an architecture in which each processor has its own memory. Each processor can access its own memory faster than it can access the memory of a remote processor. This difference (or non-uniformity) in memory access time is also known as NUMA for **Non-Uniform Memory Access**.

In such a setup every execution unit has a unique ID assigned to it. A byte $M[0]$ for example is available in the main memory of each one of the computers of the multicomputer or multiprocessor system. Thus a byte $M[0]$ is better identifiable by a pair $(memID, address)$, where memID identifies uniquely every participating Main Memory unit, and address is the address within that unit (e.g. 0 of byte $M[0]$). The cost of accessing $(MemID, Address)$ depends on the distance between the execution unit issuing the request and the main memory with identifier memID. If memID is the main memory of the execution unit issuing the request the access would be fast. Otherwise it might require a network operation if MemID belongs to one computer and the execution unit issuing the request is on another computer.

In a distributed memory organization, every memory unit has an address 0. The number of addresses 0 is the number of main memory units available. For example in a multicomputer with 8 computers we expect to have 8 main memories (one per computer) and each one of them has an address 0.

Definition 2.2 (*Non-Uniform Memory Access (NUMA)*)**.** *It is the situation where for some $(MemID, Address)$ memory request, an execution unit retrieves one byte from address Address from a main memory unit of ID memID in a varying amount of time depending on the proximity of the memory unit containing the byte and the speed of access between the computer of the client execution unit (generating the memory request) and the computer of the server execution unit (satisfying the memory request) that contains the memory unit with ID MemID.*

Accessing the byte with address 0 from the local memory of an execution unit could take time close to say 80ns. Accessing the byte with address 0 from the remote memory of another computer unit could take time close to say 80microseconds or 80milleseconds depending on the proximity of the remote memory, and the commmunication medium used (e.g. network).

2.1.2 Shared address-space architecture

Definition 2.3 (*Shared Address-space architecture*)**.** *The architecture provides hardware support for read and write operations to a shared address-space. There are two subcategories in it: (a) shared memory, and (b) distributed shared memory.*

Machines built according to this architecture are often called (or used to be called) **multi-processors**.

Shared memory

Definition 2.4 (*Shared memory*)**.** *All processors (or execution units or cores) share the same and one main memory.*

A shared-memory machine has a single address-space shared by all processors. The cost of accessing this shared memory is the same for all processors. This is the case of a UMA (Uniform Memory Access) architecture. Examples include the SGI Power Challenge, and Symmetric Multi Processor (SMP) machines, the Encore Multimax, and the Sequent Symmetry.

Machines built according to this architecture are often called SMP (symmetric multi processor). The cores of a multicore CPU (multicore processor) interact in an identical way.

Definition 2.5 (*Uniform Memory Access (UMA)*)**.** *It is the situation where every execution unit retrieves one byte from main memory in the same amount of time as any other execution unit.*

This is the case of an SMP and also the cores of multicore CPU.

Distributed shared memory

A **distributed shared memory** system is a hybrid between distributed memory and shared memory. A global address space is shared among the processors but is distributed among them. This is also an instance of a NUMA (Non-Uniform Memory Acess) architecture. An example of such an architecture is an SGI Origin 2000. Other examples of such machines is the BBN TC2000 butterfly supporting up to 128 processors, IBM SP1/SP2, SGI/Cray T3D/T3E. Several multi-core designs also fall into this group.

Definition 2.6 (*Distributed Shared Memory (DSM)*)**.** *In a multicomputer or multiprocessor system each computer has its own memory and makes it available to other processors/execution units of the multicomputer / multiprocessor system in the form of a shared memory.*

Let us call the distributed shared memory DSM[]. DSM is the union of all main memories M[] of each one of the computers available. In the simple case where 4 computers have their own main memory of maximum size of X bytes, the address space $DSM[0..X-1]$ is that of $M[0..X-1]$ of computer 0, the address space $DSM[X..2X-1]$ is that of $M[0..X-1]$ of computer 1, the address space $DSM[2X..3X-1]$ is that of $M[0..X-1]$ of computer 2, and the address space $DSM[3X..4X-1]$ is that of $M[0..X-1]$ of computer 3. There is only one address 0 across all processors. Thus DSM[0] resolves uniquely in a $(MemID, Address)$ pair that of $(0,0)$.

The existence of a cache in shared-memory parallel machines causes **cache coherence problems** when a cached variable is modified by a processor and the shared-variable is requested by another processor. An instance of a NUMA architecture is a **cc-NUMA** that stands for cache-coherent NUMA architectures (SGI Origin 2000) and resolves such issues.

Definition 2.7 (*Cache Coherent Non-Uniform Memory Access (cc-NUMA)*)**.** *It is the situation where a cached shared variable is modified by one processor and that shared variable is requested by another processor.*

A multicomputer or multiprocessor system is an interconnected system.

2.2 Taxonomy based on instruction and data separation

Definition 2.8 (*Harvard Architecture*)**.** *In a Harvard Architecture there is a separate main memory for the data of a program, and a separate main memory for the program's instructions.*

Definition 2.9 (*Von-Neumann Architecture*)**.** *In a Von-Neumann Architecture there is one main memory for the data of a program, and for the program's instructions.*

Sometimes a Von-Neumann Architecture is referred to as a 'Princeton Architecture'. Most modern computers are Von-Neumann Architectures.

In Intel CPUs there is a level-1 cache for instructions referred to as L1I and there is a level-1 cache for data referred to as L1D. Thus L1 design on Intel CPUs follows a Harvard Architectural approach (but for the cache not the main memory). In AMD CPUs there is a unified L1 cache storing instructions and data. In CPUs a level-2 or level-3 cache is a unified cached memory.

When we store programs in memory memory we split them into segments (sections). One segment is the text (instructions), another is the global initialized variables, another is the global uninitialized variables, another is the stack (user or kernel), another is the heap, etc. Elements of the Harvard Architecture approach have encroached into modern von-Neumann architectures.

2.3 Taxonomy based on communication

Definition 2.10 (*Shared Memory-based communication*)**.** *The execution units of an SMP use shared memory to communicate with each other directly bypassing the available communication network.*

No 'message' is formed for this communication.

Definition 2.11 (*Message Passing*)**.** *Communication of execution units in a multiprocessor system and in particular a multicomputer is realized by an execution unit (source) sending a message to another execution unit (destination) by using the regular network interface cards (networking) of the computers hosting the two execution units or a specialized network that is fast and optimized for the particular communication.*

This architecture is also known as a **message-passing** architecture and systems that use it are commonly referred to as **multicomputers**.

In message passing-based architectures, each processor sends/receives messages to/from other processor.

A **store-and-forward** protocol (used in nCUBE/10, T800 transputers) requires that the message be copied into the memory of the receiving processor before it is dispatched away.

In **circuit-switched** message-passing (e.g. ATM, nCUBE 2) no intermediate processor (other than source and destination) stores the message. A communication "pipe" is opened between source and destination and then a communication is initiated. Examples include Cray T3D/T3E, IBM SP1/SP2.

A 'message' is formed in this communication. It is possible that message passing is realized using shared memory based communication if the two execution units are inside the same computer. However by default message passing communication would involve by default the network interface cards (networking) of that computer.

2.4 Taxonomy based on instruction and data

Flynn's taxonomy on instruction execution and data stream.

Flynn's taxonomy is based on the available control mechanism in a computer architecture. Depending on their execution and data streams, computer architectures can be distinguished into the following four groups.

2.4.1 SISD

Definition 2.12 (*Single Instruction Single Data (SISD)*)**.** *The architecture executes a single instruction on a single piece of data at a time (instruction cycle).*

This is a sequential computer (uniprocessor) which exploits no parallelism at either instruction-level or data-level/stream. Older generation microprocessors (pre 1990s) are candidates.

Most modern microprocessors of this century, that exploit ILP (Instruction-Level Parallelism) by using pipelining or specialized instruction sets (eg Intel's MMX) would even fail to be included in this group.

2.4.2 SIMD

Definition 2.13 (*Single Instruction Multiple Data (SIMD)*)**.** *The architecture executes a single instruction on multiple pieces of data. This can be done sequentially (e.g. pipelining) or in parallel.*

This is a parallel architecture where the same instruction is executed on a large data set. SIMD architectures are suited for **data-parallel** programs e.g. image processing and multimedia processing. SIMD execution is witnessed in specialized instruction sets (e.g. Intel's MMX).

Example architectures include,

- Thinking Machines' CM-1, CM-2, and CM-200 of the 1980s/1990s,
- Cray Inc's X-MP series supercomputer of the 1980s,
- the extension of Intel's x86 architecture to support the MMX extensions/instructions is the first wide deployment of the SIMD paradigm,
- ILLIAC IV (1974), and MassPar MP-1 and MP-2 (1980s),
- modern GPU (Graphics Processing Units) are also offering extensive SIMD capabilities.

A vector processor (or array processor as it was known at the time of its introduction into Flynn's taxonomy), is the old name for the modern paradigm of Single Instruction Multiple Thread execution, which is a subdivision of SIMD. In the **Compute Unified Device Architecture** (CUDA) GPUs offered by NVIDIA provide for a Single Instruction Multiple Thread model of execution.

Definition 2.14 (Single Instruction Multiple Thread)**.** *The architecture executes a single instruction on multiple pieces of data in parallel by execution subunit that have their own register file and memory (cache or shared).*

2.4.3 MISD

Definition 2.15 (*Multiple Instruction Single Data (MISD)*)**.** *The architecture executes multiple instructions on a single piece of data.*

Some consider a **systolic array** a member of this group. A systolic array is a piped geometry of elementary processing units that have the capability to compute, store and receive/forward data to other units directly connected to themselves. A form of a systolic array is the pipeline of a modern-era microprocessor.

A fault-tolerant computer can also be considered an extreme example of an MISD architecture (think of the flight control computers of an airplane which consist of a number of identical processing units that repeat the same computations multiple times and can tolerate the loss of a number these units).

2.4.4 MIMD

This category includes all other architectures. It includes parallel computers, multicomputers, multiprocessor systems, SMPs and multicore processors.

Definition 2.16 (*Multiple Instruction Multiple Data (MIMD)*)**.** *The architecture executes multiple instructions on multiple pieces of data.*

An MIMD architecture can be an MPMD or an SPMD. (Note that Flynn's original taxonomy did not indlude these two subcategories.)

Definition 2.17 (*Multiple Program Multiple Data (MPMD)*)**.** *In a* **Multiple-Program Multiple-Data** *(MPMD) architecture, each execution unit of the architecture executes its own program on multiple data.*

Definition 2.18 (*Single Program Multiple Data (SPMD)*)**.** *In a* **Single-Program Multiple-Data** *(MPMD) architecture, each execution unit of the architecture executes the same one program on multiple data.*

Examples include Intel iPSC, CM-5, Kendall Square Research KSR-1, Cray T3D/T3E. Silicon Graphics Inc (SGI) Power Challenge, IBM SP2, and clusters of workstations. Multi-core designs fall into this group as well.

2.4.5 Hybrid architectures

Some consider a Thinking Machines CM-5 as a combination of an MIMD and SIMD as it contains control hardware that allows it to operate in an SIMD mode.

2.5 Taxonomy based on process or processor granularity

The term **granularity** sometimes refers to the power of individual processors. Sometimes it is also used to denote the degree of parallelism.

Process Granularity refers to the amount of computation assigned to a particular process/processor of a parallel machine for a given parallel program. It also refers, within a single program, to the amount of computation performed before communication is issued. If the amount of computation is small (low degree of concurrency) a process is **fine-grained**. Otherwise granularity is **coarse**.

(1) A **coarse-grained** architecture consists of (usually few) powerful processors (e.g. the first Cray machines).

(2) a **medium-grained** architecture is a hybrid between the two (e.g. CM-5).

(3) a **fine-grained** architecture consists of (usually many inexpensive) processors (e.g. TM CM-200, CM-2) or the processors of a systolic array, a pipeline or the units of the MMX extensions to x86.

Further refinement is also possible. When synchronization takes place at software level (thread or process) we have the following interpretations of multiprocessor system synchronization

Definition 2.19 (*Multiprocessor System Synchronization*)**.** *A multiprocessor system can have its program-s/processes synchronizing at different granularities. These include*

- *Independent (e.g. two processes of two different users on same computer),*
- *Very Coarse granularity (e.g. a client and server process),*
- *Coarse granularity (e.g.* `ps -ef | egrep somestring`*),*
- *Medium granularity (e.g. multithreaded or multiprocessing application),*
- *Fine granularity (e.g. exhibited in a parallel program).*

2.6 Taxonomy based on processor synchronization

(1) In a **fully synchronous** system a global clock is used to synchronize all operations performed by the processors.

(2) An **asynchronous** system lacks any synchronization facilities. Processor synchronization needs to be explicit in a user's program.

(3) A **bulk-synchronous** system comes in between a fully synchronous and an asynchronous system. Synchronization of processors is required only at certain parts of the execution of a parallel program.

Chapter 3

Introduction to parallel computing

3.1 What is parallel computing?

Parallel computing is the concurrent manipulation ("in-parallel") of data distributed on one or more processors to cooperatively solve a computationally intensive problem faster than other methods. The collection of processors used in parallel computing is called a parallel computer.

There are several forms of parallel computing.

- *instruction-level parallelism* (ILP), in which several operations are executed simultaneously employed techniques such as pipelining, superscalar execution, speculative execution, and branch prediction to exploit ILP,
- *data parallelism*, in which data are distributed over a number of processors.
- *task parallelism*, in which processes (or threads) are distributed over a collection of processors.

3.2 What is a parallel computer?

A parallel computer is a multiple-processor computer capable of parallel computing. Sometimes the term parallel computer is confused with a supercomputer. The term supercomputer refers to a computer that can solve computationally intensive problems faster than traditional computers. A supercomputer may or may not be a parallel computer.

A parallel computer can be a very complex highly specialized system or just a collection (usually referred to as a cluster) of pc workstations connected through a high speed (e.g. Ethernet-based) switch. It can be a symmetric multi-processor (SMP) in which several (identical) processors access a single (shared) memory, or a Chip Multi-Processor (CMP) that contains several control units (processors that are usually called cores) on a single die/chip. A CMP or multi-core processor differs from superscalar processor in the sense that they maintain multiple instruction streams from which they can issue multiple instructions. In superscalar design, there is only one instruction stream. The cores within a CMP can in fact be superscalar processor themselves. Multithreading (e.g. Intel's Hyperthreading) is a form of virtual multi-core approximation, in which multiple threads are using one execution unit.

A distributed computer is a parallel computer in which the multiple (homogeneous or non-homogeneous) processors/computers (i.e. processing units or elements) are connected through a high-speed external (to the processors, or cabinets, housing the processors) network. In a cluster, the computers are loosely coupled

and are more frequently non-homogeneous. Beowulf clusters refere to homogeneous, identical computer connected with a TCP/IP (fast) Ethernet network.

Specialized parallel computers come in the form of reconfigurable computers such as Field-Programmable Gate Arrays (FPGA) that act as a coprocessor to a general-purpose processor, General-Purpose Graphic Processing Units (GPGPU) that serve the same purpose that an FPGA serves, except that they specialize in graphics processing, and vector processor.

3.3 Parallelism and Parallelization: Issues and Challenges

One example in real life that illustrates the principles related to parallel computing is book shelving in a library. One can accomplish this simple task by using just a single worker. If the number of books to be shelved is quite large, this task might take plenty of time to complete. If one wants to speed up the process, one can take advantage of the inherent **parallelism** available in this process. Some form of **parallelization** of the shelving task may be required.

Idea 1. A number p of workers is available. Then n books that are to be shelved are split evenly among these workers so that each one stacks about n/p books. If such an approach is used, several problems may occur when many workers try to stack the next available for stacking book in the same shelf or location. Some kind of arbitration is required on who works first or enters the stack first.

Question 1. Does it matter if some books are heavier than others?

Question 2. Does it matter if some books are located in adjacent shelves.

Idea 2. A number p of workers is available. Then n books are split evenly among the workers so that each one is to stack about n/p books. To avoid arbitration problems observed in Idea 1 we split the books so that each worker works on a different set of shelves (e.g. by grouping books by topics/call numbers). In this way, no book from two different workers will end up in the same shelf.

Question 3. How more complex is the splitting task? Does it require too much time?

Therefore the most important issue in parallelizing a sequential task is identifying the inherent and apparent parallelism in that task, and then developing methods to exploiting it in an efficient manner. The following become then important aspects of the process of parallelization.

1. **Task/Program Partitioning.** It describes the process of splitting a single task among the several available processors so that each processor is assigned approximately the same workload, and all processors can and will work collectively to complete the task. The object of task partitioning is that the processors finish the work faster than the time it would take a single processor to perform that same task. This is also referred to as **task parallelism**.

2. **Data Partitioning**. It describes the potential splitting of the data used by the task among the available processors in such a way that processor interaction is minimized and the processors complete the task concurrently faster than an individual processor. This is also referred to as **data parallelism**.

3. **Communication/Arbitration**. It describes the process of information of data exchange among the processors and how one can rbitrate communication-related conflicts.

In order to realize an efficient parallelization of a sequential task several challenges need to be overcome, or support becomes available for the realization of task or data partioning, communication and synchronization of parallel tasks.

1. Design of parallel computers that can effectively support parallel computing requirements.

2. Effective use of memory and multiple-memory hierarchies in the case of multi-core architectures is undertaken.

3. Design, analysis and evaluation of parallel algorithms run on such machines is successfully undertaken.

4. Portability and scalability issues related to parallel programs and algorithms are understood and analyzed.

5. Tools and libraries become available in and for such systems for the benefit of the software developer and other users.

3.4 Requirements for parallel software development

A **parallel algorithm** is an algorithm designed for a parallel computer.

A **parallel computer** is a collection of processors or computers that cooperatively solve computationally intensive problems faster than other computers.

Parallel algorithms allow the efficient programming of parallel computers. This way the (or some) waste of computational resources can be avoided.

The term **supercomputer** also refers to a general-purpose computer that can solve computationally intensive problems faster than traditional computers. A supercomputer may or may not be a parallel computer.

The cost of building a parallel computer has dropped significantly in the past years. One is able to build a parallel computer using off-the-shelf commercial components. This way workstations can be arranged together in a cluster/network and interconnected with a high-speed network (e.g Myrinet switches, 100Mb/sec or Gigabit ethernet switches) and form a high-performance parallel machine. A multi-core computer with several cores is readily available at no (or insignificant) extra cost.

The construction of a parallel computer is more complicated than that of a sequential machine. One needs to take into consideration not only those issues related with building a sequential machine but also issues related with the interconnection of multiple processors and their efficient coordination. A complete network where each processor can directly communicate with every other processors is not currently technologically feasible if the parallel machine consists of more than a few processors. Methods thus need to be devised that allow the efficient dispatch of information from one processor to another (routing protocols and algorithms). Problems like congestion (bottlenecks) may occur if someone is not very careful in designing such methods.

An efficient sequential algorithm does not necessarily translate into an efficient parallel algorithm. Implementation of an inappropriate algorithm in parallel may waste resources (e.g. computational power, memory). A sequential algorithm needs perhaps to be split into pieces so that each piece is executed by a processor of the parallel machine. Each piece, however, may need to interact with the piece held at another processor (say because it wants to access a memory location held there). If a sequential algorithm utilizes M memory locations and solves a problem instance within time T, one expects a parallel algorithm solving the same problem instance to utilize as many resources as its sequential counterpart. If the algorithm is run on a p processor machine it is thus expected to utilize memory of overall size (over all p processors) that is comparable to M and the sum of execution time over all p processors to be around T. The best one can hope for is for the parallel algorithm to use M/p memory per processor and its parallel time to be T/p. In order to achieve such a running time (and it is not clear that this is possible, even if it is feasible) an algorithm designer/programmer needs to take into consideration the architecture of the parallel machine (e.g. how processors are interconnected). This optimal speedup of a parallel program is more the exception rather than the rule in practice. The reason is that **interprocessor communication** (i.e. conflict resolution when accessing the same date) and **synchronization** (i.e. arbitrations) are expensive operations and unavoidable in a parallel environment.

3.4.1 Users of parallel computers

There are many science problems that can only be solved and studied by using simulations. Such problems require extraordinary powerful computers and cannot be solved in reasonable amount of time today. A collection of such problems that can be solved in parallel include some of the following problems that one might refer as *grand challenge problems*.

- Web Searching,
- Data Mining (find preferences of customers and then use more efficient direct marketing methods).
- Quantum chemistry, statistical mechanics, cosmology and astrophysics.
- Material sciences (e.g. superconductivity).
- Biology, biochemistry, gene sequencing.
- Medicine and human organ modeling (e.g. to study the effects and dynamics of a heart attack).
- Environmental modeling and global weather prediction.
- Visualizaton (e.g. movie industry).
- Computational-Fluid Dynamics (CFD) for aircraft and automotive vehicle design.

The following example might illustrate the importance and also the challenges one is facing with parallel computing.

For local weather prediction, an area $2000nm \times 2000nm$ is broken into cubic cells each $0.1mile$ wide, from the surface to an altitude of 10-12 miles. There are $20000 \times 20000 \times 100 = 4 \cdot 10^{10}$ cubic cells overall, each one requiring around 200 floating operations for each iteration of a weather prediction program.

Each iteration gives the state of the atmosphere for a 15 minute interval and therefore approximately 200 iterations are required for a 3 day forecast for a total computation count of 10^{15} flops. **A commercial processor rated at 1 Gigaflops would require 11 days to predict the weather more than one week ago!**

N-body simulation involves the calculation of forces and therefore of the position of N bodies in three-dimensional space. There are approximately 10^{11} stars in our galaxy. A single iteration of such a program would require several divisions and multiplications. On a uniprocessor machine it would take a year to complete a single iteration of the simulation. In order to predict whether a meteorite is going to hit earth in the distant future one would need to sacrifice precision over running time or the other way around.

3.5 Past, Present and Future Challenges

In the 1980's a Cray supercomputer was 20-100 times faster than other computers (mainframes, minicomputers) in use at that time. Because of this, there was an immediate pay-off in investing on a supercomputer: a tenfold price increase was worth 20-100 times improvement in performance. In the 1990's a "Cray"-like CPU was on the average twice - four times as fast as (and sometimes, slower than) a microprocessor. Paying 10-20 times more to buy a supercomputer started making not much sense.

The solution to the need for computational power is a *massively parallel* computer, where tens to hundreds of commercial off-the-shelf processors are used to build a machine whose performance is much greater than that of a single processor.

Various questions arise when such a combining of processor power takes place.

- How does one combine processors efficiently?
- Do processors work independently?

- Do they cooperate? If they cooperate how do they interact with each other?
- How are the processors interconnected?
- How can we make programs portable?
- How does one program such machines so that programs run efficiently and do not waste resources?

Most of the parallel computers manufactured in the 80s and 90s were used to solve technical problems.

Several companies built proprietary systems that were hard to program in a general-purpose way. This led to the demise of several of these companies early pioneers of parallel computing. The names of MassPar, Thinking Machines, Cray, and Silicon Graphics are a fading memory of their glorious past.

In 1991, microprocessor speeds were at the 33MHz level. By 2002 they had reached the 2.2GHz level, an annual increase of roughly 45% roughly confirming Gordon Moore's Law (a co-founder of Intel) that says that that transistor capacity would double roughly every 2 years.

> **Historical Parenthesis**: In his original 1965 paper, Moore was predicting a doubling of capacity every year. In 1975 he revised it to "every couple of years", and an Intel engineer (Moore himself suggesting that engineer to be Dave House) modified it to "every 18 months" that is *often quoted* in the media.

In the years leading to 2002, Moore's law could only be maintained by incorporating several interesting hardware tricks. This started in the late 80s with the addition of cache memories, and continued with the use of **superscalar designs**, where multiple instructions are executed per clock cycle; such designs are also pipelined, may have multiple FPUs (Floating Point Units) or ALUs (Arithmetic Logic Units).

Yet since the 2.2GHz level reached in 2002, microprocessor speeds have remained at the 3.0-4.0GHz or lower level ever since.

One reason for this stall is technology limitations: we run out of (high-performing) hardware tricks. Another reason is power consumption. CPU architectures started to consume too much power. The power density of a microprocessor is around $10W/cm^2$!

On a hot summer day, the sunlight reaching us has density less than $0.1W/cm^2$*, and we feel hot and uncomfortable.* Yet microprocessors operate at 100 times that comfort level!

3.5.1 Limitations of sequential (serial) processors

1. **Power consumption:** Remains steady at around 100W

2. **Clock Speed:** Recent multi-cores venture at clock speeds in the 2-3GHz level, which is roughly the same as that achieved by single-core designs back in 2001-2002 (and some of those 2001-2002 designs had the potential to reach clock speeds as high as 3.5-4.0GHz).

3. **Transistors:** Several billions per chip.

Limitation 1: Power Consumption

Power consumption is linear to frequency and capacitance and quadratic to voltage. In other words $P \equiv C \cdot F \cdot V^2$, where C, V, F are capacitance, voltage, and frequency respectively.

Cutting voltage levels by 25% or so, one can decrease power consumption by 50%.

Lowering the frequency and the voltage but doubling the cores has the effect of overall increasing performance by significantly reducing power consumption.

This byitself constitutes a benefit.

To further increase performance, cache memories were added in a variety of forms and cache sizes have also increased considerably. Level 2 caches increased from several hundred kilobytes to several megabytes. Level 3 caches has been added in some cases amounting of tens of megabytes.

Limitation 2: Hidden parallelism (superscalar design)

Superscalar designs (multiple ALUs, FPUs), Instruction-Level parallelism (ILP) in the form of pipelining that allowed microprocessors to maintain their performance edge through 2002 have used up most of the tricks available.

Limitation 3: Chip densities

Chip density has increased reaching now a billion or more transistors per chip. But this comes at an extreme cost of building the manufacturing facilities: costs run up to billion of dollars.

3.6 The era of multi-core computing

In recent years parallel computing has been revitalized in the form of multi-core computing. Parallelism has provided a new set of tricks to increase performance. Given the limitations of a single microprocessor, the hardware industry came up with the concept of including multiple cores on a single die to increase overall performance. In order to achieve this and also reduce power consumption (an impediment for designing mobile devices and also cooling mechanism for non-mobile components) several trade-offs (i.e. compromises) were made: each individual core operated at a lower frequency, all cores used a common main memory and voltage levels were reduced.

We mentioned earlier some interesting issues related to parallelism. The emergence of multi-core computing has added urgency to several of them.

- **Every computer is a parallel computer**. How does one put several cores on a single chip? How does one organize the memory to increase performance? How does one allow interaction between the cores?
- **Every program is (or is going to be a) a parallel program** to achieve high performance and utilize the underlying hardware. Do we need to have a new software or programming models for these computers? What will be the hardware abstraction equivalent to the von-Neuman model of sequential computing? What are the killer apps?
- **Every programmer who programs such a computer is (or ought to be) a parallel programmer**. How do you educate software-engineers and instill in them that (parallel) performance optimization will become the number one issue/priority (whether it had been so in the past or not)?
- **Software support for such architectures**. In the meantime while programmers are being (or are to be) retrained and reeducated, will software libraries fill the gap?

In order to be able to write efficient parallel programs the following issues need to be addressed. Several of them have no sequential computing counterpart.

- **Task Parallelism**. Is there enough parallelism in the application. Amdahl's law that will be discussed later addresses several limitations.
- **Data Parallelism**. Can we split the data efficiently?
- **Task/Process/Thread Granularity** refers to the amount of computation assigned to a particular processor or core for a given parallel /task/process/thread. It also refers to the amount of computation performed before a communication is issued. It is task and data parallelism that decide **process granularity**.
- **Load-Balance** Are tasks performing about the same amount of work or are several of them idle?
- **Locality of reference** Is memory available locally or does one need to retrieve it from a remote location? Such a remote location can be another processor's memory bank, the main memory residing outside the CPU chip, the level-3 (i.e. L3) cache that is often outside the CPU chip, or the level-2 (L2) cache shared by multiple cores. Multiple memory hierarchies are being used to increase performance and their efficient use will determine how efficient a program is.
- **Synchronization** Coordinate simultaneous memory accesses, or transition to the next common task in the program sequence.
- **Latency/Startup issues** Cost of synchronization, communication, or starting a program which is a process or a thread, or a memory access to an alternate memory hierarchy.
- **Modeling/Abstraction** What is the hardware abstraction that will provide a reliable model to describe the performance of an algorithm/program and predict its performance on multiple multi-core or multi-computer platforms or its scalability on similar but many-core systems?

3.7 Performance Characteristics

The performance of a parallel algorithm A that solves some problem instance can be measured in terms of various measures.

- **Parallel Time** T gives the execution time of the parallel algorithm and $T = max_i T_i$, where T_i is the running time of algorithm A on processor i.
- **Processor size** P is the number of processors assigned to solve a particular problem instance.
- **Work** is the product $P \cdot T$. Sometimes the **actual work** $\sum_{j=1}^{T} P_j \leq W$ is used instead, where P_j is the number of processors that are active in step j.
- **Speedup** given by $s = T_s/T$ is the ratio of the execution time T_s of the most efficient sequential algorithm that solves a particular problem instance to parallel time T of A on the same instance.
- **Scaled speedup** is the ratio T_1/T_p, where T_i here denotes the parallel time T of A when i processors are used to solve that particular problem instance.
- **Efficiency** $e = s/p$, which is sometimes expressed as a percentage. It is also the case that $e = T_s/W$, where $W = TP$.
- **Scaled Efficiency**, which is derived similarly from scaled speedup.

3.7.1 Amdahl's Law

Gene Amdahl was first to argue about the limitations of parallelism. His observations can take the form of the following theorem.

Theorem 3.1 (Amdahl's Law)**.** *Let f, $0 \leq f \leq 1$, be the fraction of a computation that is inherently sequential. Then the maximum obtainable speedup S on p processors is*

$$S \leq \frac{1}{f + (1-f)/p}$$

Proof. Let T be the sequential running time for the named computation. fT is the time spent on the inherently sequential part of the program. On p processors the remaining computation, if fully parallelizable, would achieve a running time of at most $(1-f)T/p$. This way the running time of the parallel program on p processors is the sum of the execution time of the sequential and parallel components that is, $fT + (1-f)T/p$. The maximum allowable speedup is therefore

$$S \leq T/(fT + (1-f)T/p)$$

and the result is proven. ■

Amdahl used this observation to advocate the building of even more powerful sequential machines as one cannot gain much by using parallel machines. For example if $f = 10\%$, then $S \leq 10$ as $p \to \infty$. The underlying assumption in Amdahl's Law is that the sequential component of a program is a constant fraction of the whole program. In many instances as problem size increases the fraction of computation that is inherently sequential decreases with time. In many cases even a speedup of 10 is quite significant by itself.

Consider the following alternative argument to Amdahl's law.

Example 3.1. *You are commander of a Patriot missile battery whose command processor is a sequential computer with response time to a missile threat of 10 seconds. A new missile threat requires a response time of only 5 seconds. You are given two options.*

(1) Wait 2-3 years until a faster command processor comes out with a two-fold increase in performance.

(2) Order a parallel machine that uses the current generation of command processors. The inherently sequential component of the sequential missile response program is at most 20%. The parallelization of the program takes a couple of days because the programmers are NJIT graduates who had taken a parallel computing course at NJIT.

Question 3.1. *What do you do? Amdahl's law says that no matter how many processors you use you can only get a speedup of 5. Do you wait until option (1) becomes available or take up option (2)?*

This is an example where efficiency does not matter. What it really matters is whether by choosing a parallel computer a significant increase in performance can be obtained. In addition Amdahl's law is based on the concept that parallel computing always tries to minimize parallel time. In some cases a parallel computer is used to increase the problem size that can be solved in a fixed amount of time. For example in weather prediction this would increase the accuracy of say a three-day forecast or would allow a more accurate five-day forecast.

3.7.2 Parallel vs Sequential Computing: Gustafson's Law

The following Law is due to Gustafson and sometimes referred to as the Gustafson-Barsis law. It states that any sufficiently large program can be parallelized efficiently.

Theorem 3.2 (Gustafson's Law)**.** *Let the execution time of a parallel algorithm consist of a sequential segment fT and a parallel segment $(1-f)T$ and the sequential segment is constant. The scaled speedup of the algorithm is then.*

$$S = \frac{fT + (1-f)Tp}{fT + (1-f)T} = f + (1-f)p$$

Proof. Let T be the parallel running time. Tf is the portion of the running time that is inherently sequential and $T(1-f)$ is the portion that has admitted parallelization. Thus on a sequential machine consisting of only one processor that program would take time equal to the sequential component Tf plus at most $T(1-f)p$ that results if the p parallel segments are emulated on a single processor. Total time (sequential) is thus $T(1-f)p + Tf$ providing a (scaled) speedup of $f + (1-f)p$. For $f = 0.05$, we get $S = 19.05$, whereas Amdahl's law gives an $S \leq 10.26$. Amdahl's law assumes that problem size is fixed, while Gustafson's law assumes that running time is fixed. This is similar to the question

Question 3.2. *In a 1-day timeframe what is the grid-size/accuracy to run a 3-day or a 10-day weather prediction code?*

For such a setting we are not interested in minimizing the runtime. We are more interested in improving the information we get by running a bigger problem size in the same amount of time.

□

Amdahl's Law assumes that problem size is fixed when it deals with scalability. Gustafson's Law assumes that running time is fixed.

Example 3.2. *For $f = 0.05$, we get $S = 19.05$, whereas Amdahl's law gives an $S \leq 10.26$.*

Example 3.3. *Consider a 1 vs p processor scenario. Time fT is inherently sequential. Sequential $p = 1$ parallelizable time of $(1-f)Tp$ maps to $(1-f)T$ for p processors. The total time on p processors is T where $T = ft + (1-f)T$ and for one processors it becomes $T(f + (1-f)p)$.*

3.8 Brent's Scheduling Principle: Emulations

Suppose we have an unlimited parallelism efficient parallel algorithm, i.e. an algorithm that runs on zillions of processors. In practice zillions of processors may not available. Suppose we have only p processors. A question that arises is what can we do to "run" the efficient zillion processor algorithm on our limited machine.

One answer is emulation: simulate the zillion processor algorithm on the p processor machine.

Theorem 3.3 (Brent's Principle)**.** *Let the execution time of a parallel algorithm requires m operations and runs in parallel time t. Then running this algorithm on a limited processor machine with only p processors would require time $m/p + t$.*

Proof. Let m_i be the number of computational operations at the i-th step, i.e. $\sum m_i = m$. If we assign the p processors on the i-th step to work on these m_i operations they can conclude in time $\lceil m_i/p \rceil \leq m_i/p + 1$. Thus the total running time on p processors would be.

$$\sum_i \lceil m_i/p \rceil \leq \sum_{i=1}^{t} m_i/p + 1 = t + \sum_i m_i/p = t + m/p.$$

□

3.9 Terminology

Parallel Computing/Processing is the concurrent manipulation of data distributed on one or more processors solving a single computationally intensive problem.

Parallel computer is a multiple-processor computer capable of parallel computing.

Supercomputer is a general-purpose computer capable of solving problems at high computational speeds and faster than traditional computers. It can be a parallel computer or not.

Process Granularity refers to the amount of computation assigned to a particular processor of a parallel program. It also refers, within a single program, to the amount of computation performed before communication is issued. If the amount of computation is small the granularity is **fine-grained**. Otherwise granularity is **coarse**.

Throughput is the number of results produced in one time unit (usually, one second).

UMA (Uniform Memory Access). It refers to that processor organization where the cost of accessing a memory location is the same for any processor. In a machine supporting UMA all processors work through a centralized switch to reach a shared memory.

NUMA (Non-Uniform Memory Access). It refers to that processor organization where the cost of accessing a memory location is not uniform. Parallel machines with multiple memory hierarchies fall into this category.

cc-NUMA (cache-coherent NUMA) refers to the architecture in which the existence of a cache causes **cache coherence problems** when a cached variable is modified by a processor and the shared-variable is requested by another processor. In cc-NUMA architecture such issues are resolved.

Multi-processor system. A collection of processors sharing a global shared memory. An SMP (Symmetric Multi Processor) is an instance of a UMA multiprocessor system.

Multi-computer system. A collection of computers (each one consisting of at least one processor with its own memory).

Multi-core is an architecture in which multiple processor cores are placed on the same chip/die.

Message-Passing is the method of interprocessor communication where each processor sends/receives messages. A **store-and-forward** protocol (used in nCUBE/10, T800 transputers) requires that the message be copied into the memory of the receiving processor before it is dispatched away. In **circuit-switched** message passing (e.g. ATM, nCUBE 2) no intermediate processor (other than source and destination) stores the message. A communication "pipe" is opened between source and destination and then a communication is initiated.

Pipelining. One way one can increase the level of parallelism is to use the technique of **pipelining** where a computation is split into stages so that the output of one stage becomes the input of the following one. This way after some initial delay different data are processed at different stages/levels of the pipeline. An alternative to pipelining is **data parallelism** where multiple units operate on different pieces of data. Note that the level of parallelism for pipelining is fixed (number of stages) whereas for data parallelism it is not fixed.

Scalability refers to both algorithm scalability and architectural scalability. Algorithmic scalability means that the level of parallelism increases at least linearly with problem size. For architectural scalability, it means that the same performance per processor is maintained if number of processors increases.

Cache is a memory place between the processor and the main memory of a computer system. There are level 1, level 2, and level 3 caches available. Usually level 1 is stored on chip/die and level 2 outside the processor.

Multitasking is the execution of one or more processes apparently at the same time.

3.10 Famous scalability-related (mis) quotes

Historical Quote: "640K OF MEMORY OUGHT TO BE ENOUGH FOR ANYBODY", *Bill Gates, Chairman, Microsoft, 1981.* He denies ever making this statement attributed to him.

Historical Quote: "(THERE IS) NO REASON FOR ANY INDIVIDUAL TO HAVE A COMPUTER IN HIS HOME", *Ken Olsen, Chairman*, Digital Equipment Corporation, in a 1977 World Future Society meeting in Boston, MA.

Historical Quote: "I THINK THERE IS A WORLD MARKET FOR MAYBE FIVE COMPUTERS", *Thomas J. Watson Sr.*, Chairman, IBM, 1943. Another probably incorrect misquote. No one has ever established that this quote was actually made.

3.11 The Parallel Random Access Machine

The Parallel Random Access Machine (PRAM) is one of the simplest ways to model a parallel computer. A PRAM consists of a collection of (sequential) processors that can *synchronously* access a global *shared* memory in unit time. Each processor can thus access its shared memory as fast (and efficiently) as it can access its own local memory. The main advantages of the PRAM is its simplicity in capturing parallelism and abstracting away communication and synchronization issues related to parallel computing. Processors are considered to be in abundance and unlimited in number. The resulting PRAM algorithms thus exhibit *unlimited parallelism* (number of processors used is a function of problem size). The abstraction thus offered by the PRAM is a fully synchronous collection of processors and a shared memory which makes it popular for parallel algorithm design. It is, however, this abstraction that also makes the PRAM unrealistic from a practical point of view. Full synchronization offered by the PRAM is too expensive and time demanding in parallel machines currently in use. Remote memory (i.e. shared memory) access is considerably more expensive in real machines than local memory access as well and UMA machines with unlimited parallelism are difficult to build.

Depending on how concurrent access to a single memory cell (of the shared memory) is resolved, there are various PRAM variants. ER (Exclusive Read) or EW (Exclusive Write) PRAMs do not allow concurrent access of the shared memory. It is allowed, however, for CR (Concurrent Read) or CW (Concurrent Write) PRAMs. Combining the rules for read and write access there are four PRAM variants: EREW, ERCW, CREW and CRCW PRAMs. Moreover, for CW PRAMs there are various rules that arbitrate how concurrent writes are handled.

(1) in the *arbitrary* PRAM, if multiple processors write into a single shared memory cell, then an arbitrary processor succeeds in writing into this cell,

(2) in the *common* PRAM, processors must write the same value into the shared memory cell,

(3) in the *priority* PRAM the processor with the highest priority (smallest or largest indexed processor) succeeds in writing,

(4) in the *combining* PRAM if more than one processors write into the same memory cell, the result written into it depends on the combining operator. If it is the *sum* operator, the sum of the values is written, if it is the *maximum* operator the maximum is written.

The EREW PRAM is the weakest among the four basic variants. A CREW PRAM can simulate an EREW one. Both can be simulated by the more powerful CRCW PRAM. An algorithm designed for the common PRAM can be executed on a priority or arbitrary PRAM and exhibit similar complexity. The same holds for an arbitrary PRAM algorithm when run on a priority PRAM.

Assumptions

In this monograph we examine parallel algorithms on the PRAM. In the course of the presentation of the various algorithms some common assumptions will be made. The input to a particular problem would reside in the cells of the shared memory. We assume, in order to simplify the exposition of our algorithms, that a cell is wide enough (in bits or bytes) to accommodate a single instance of the input (eg. a key or a floating point number). If the input is of size n, the first n cells numbered $0,\ldots,n-1$ store the input. In the discussion below, we assume that the number of processors of the PRAM is n or a polynomial function of the size n of the input. Processor indices are $0,1,\ldots,n-1$.

3.12 PRAM Algorithm: Parallel sum

Problem 3.1 (*Parallel Sum*)**.**
Input. *Array/Sequence* $x[0\ldots n-1]$ *of n values.*
Output. *Evaluate* $\sum_{i=0}^{n-1} x[i] = x[0]+\ldots+x[n-1]$.

Solution.
An associative operator $\otimes$ is one such that for all a,b,c we have $(a\otimes b)\otimes c = a\otimes(b\otimes c)$. The operator $+$ can be replaced with any other associative operator such as $\times$, min, max or more complex operators (e.g. matrix multiply or matrix add). A sequential algorithm that solves this problem requires $n-1$ additions. If a combining CRCW PRAM with arbitration rule `sum` is used to solve this problem, the resulting algorithm is quite simple. In the first step each processor i reads memory cell i storing x_i. In the following step every processor i concurrently with the remaining processors writes the read value into an agreed cell say 0. Arbitration ruls `sum` guarantees that the sum would be written into cell 0. The time is $T = O(1)$, and processor utilization is $P = O(n)$.

A more interesting algorithm is the one presented below for the EREW PRAM. For a PRAM implementation, value x_i is initially stored in shared memory cell $M[i]$. □

Proposition 3.1. *The sum* $x_0+x_1+\ldots+x_{n-1}$ *is to be computed in in an EREW PRAM in* $T = O(\lg n)$, $P = n$ *and* $W = O(n\lg n)$, $W_2 = O(n)$. *Without loss of generality, let n be a power of two.*

Proof. The algorithm consists of $\lg n$ steps. In step i, processor $j < n/2^i$ reads shared-memory cells $M[2j]$ and $M[2j+1]$ combines (sums) these values and stores the result into memory cell $M[j]$. After $\lg n$ steps the sum resides in cell 0. Algorithm `Parallel_Sum` has $T = O(\lg n)$, $P = n$ and $W = O(n\lg n)$, $W_2 = O(n)$.
□

Algorithm `Parallel_Sum` can be extended to include the case where n is not a power of two. `Parallel_Sum` is the first instance of a sequential problem that has a trivial sequential but more complex parallel solution. Instead of operator *Sum* any associative operator could have been used.

Exercise 3.1. *Can you improve* `Parallel_Sum` *so that T remains the same,* $P = O(n/\lg n)$, *and* $W = O(n)$*? Explain.*

Exercise 3.2. *What if i have p processors where* $p < n$ *? (You may assume that n is a multiple of p).*

Exercise 3.3. *Generalize the Parallel Sum algorithm to any associative operator.*

```
1  //  pid() : ID of the processor issuing the call
2      void ParallelSum(n) {
3         i=1 ; j=pid() ;  bound = n;
4         while ( i <= n ) {
5            if ( j <= bound/2) {
6               a = (2*j   <= bound) ? M[2j]   : 0);
7               b = (2*j+1 <= bound) ? M[2j+1] : 0);
8               M[j] = a+b$;
9            }
10           i=2*i$; bound=bound/2;
11        }
12     }
```

Figure 3.1: Parallel sum on an EREW PRAM

```
M[0]     M[1]     M[2]     M[3]     M[4]     M[5]     M[6]     M[7]

 x0       x1       x2       x3       x4       x5       x6       x7    t=0

x0+x1     x2+x3    x4+x5    x6+x7                                      t=1

x0+...+x3 x4+...+x7                                                   t=2

x0+...+x7                                                             t=3
```

Figure 3.2: Parallel Sum: Schedule

3.13 PRAM Algorithm: Broadcasting

In parallel programs, several pieces of data need to be known by all processors. For example problem size, or other problem parameters. The operation that informs all processors on this piece or pieces of data is called broadcasting. In its simpler form, broadcasting involves making a copy of the original datum to memory exclusively (or with no delay) accessible by each individual processor.

Problem 3.2 (*Broadcasting*)**.**
Input. *A piece of data (datum) d is stored in memory location $M[0]$.*
Output. *Make a copy of d to memory locations $M[1]\ldots M[n-1]$ so that processor i can have exclusive access of the copy $M[i]$, $0 \leq i \leq n-1$.*

Solution.
Broadcasting is a parallel copy algorithm. It replicates d from $M[0]$ to $n-1$ additional locations $M[1\ldots n-1]$. On a CREW PRAM no broadcasting operation is needed. A CR?W PRAM algorithm that solves the broadcasting problem has performance $P = O(n)$, $T = O(1)$, and $W = O(n)$. It works as follows. Processor i reads d concurrently with other processors from $M[0]$ in one parallel step. On an EREW PRAM broadcasting can be performed in $O(\lg n)$ steps. The structure of the algorithm is structurally the reverse of the parallel sum algorithm. In $\lg n$ steps d is broadcast as follows. In step i each processor with index j less than 2^i reads the contents of cell $M[j]$ and copies it into cell $M[j+2^i]$. After $\lg n$ steps each processor i reads the message by reading the contents of cell i. This approach requires n to be a power of two; this however can be relaxed. □

Proposition 3.2. *Broacasting of d originally located at $M[0]$ in an EREW PRAM can be realized in $T = O(\lg n)$, $P = n$ and $W = O(n \lg n)$, $W_2 = O(n)$. Without loss of generality, let n be a power of two.*

Proof. The algorithm is shown in Figure 3.3. We assume n is a power of two and the number of processors P is equal to n. □

```
1  //  pid() : ID of the processor issuing the call
2  //  P == n
3      void Broadcast(memory M, datum d) {
4        i = 0 ; j = pid() ;  M[0] = d ;
5        while ( 2**i < P ) {
6           if ( j < 2**i )
7              M [ j + 2**i ] = M[ j ] ;
8           i= i + 1 ;
9        }
10       Processor j reads d from M[j];
11     }
```

Figure 3.3: Broadcasting on an EREW PRAM

Exercise 3.4. *Broadcasting on a hypercube and a butterfly (Hint: Base your solution on the* `Broadcast` *algorithm).*

Exercise 3.5. *Suppose (in some strange model) we can copy in a single cycle not once but twice), that is, $M[0]$ can be copied to $M[1]$ and $M[2]$ in a single time-step. Can you have a faster broadcasting? What if t copies per cycle are allowed? Explain.*

3.14 PRAM Algorithm: Parallel Prefix

Problem 3.3 (*PPF Sum*)**.**
Input. *Array/Sequence* $x[0\ldots n-1]$ *of n values and a unit cost associative operator* $\oplus$.
Output. *Evaluate* $\sum_{i=0}^{j} x[i] = x[0] \oplus \ldots \oplus x[j]$, *for all* $j = 0,\ldots n-1$.

Solution.
Parallel prefix is also called *prefix sums* if the associate operation is addition or *scan* in general. Note that $+$ is an operator, that implies operation addition; the $+$ operator in this context is a binary operator requiring a left and right operand. The operator $\oplus$ can be replaced with any other associative operator such as $\times$, min, max or more complex operators (e.g. matrix multiply or matrix add). In our case we will treat $\oplus$ as if it is $+$. A sequential algorithm solves this problem by performing $n-1$ o-additions. The cost of an operation is going to be assumed to be constant (in fact unit time). This can be relaxed accordingly. An algorithm is presented below for the EREW PRAM. For a PRAM implementation, value x_i is initially stored in shared memory cell $M[i]$. It has many uses in parallel computing such as in load-balancing, the work assigned to processors and compacting data structures such as arrays. □

We shall prove that computing ALL THE SUMS is no more (asymptotically) difficult than computing the single sum $x_0 + \ldots x_{n-1}$. An algorithm for parallel prefix on an EREW PRAM would require $\lg n$ phases. In phase i, processor j reads the contents of cells j and $j - 2^i$ (if it exists) combines them and stores the result in cell j. We first present an non-optimal divide and conquer approach that only works on a CREW PRAM or on an EREW PRAM with a significant slowdown.

3.14.1 PPF sums: a CREW approach

Proposition 3.3 (*PPF sums on a CREW PRAM*)**.** *The prefix sums*

$$\sum_{i=0}^{j} x[i] = x[0] \oplus \ldots \oplus x[j],$$

for all $j = 0,\ldots n-1$, *can be computed in a CREW PRAM in* $T = O(\lg n)$, $P = n$ *and* $W = O(n \lg n)$, $W_2 = O(n)$. *Without loss of generality, let n be a power of two.*

Proof. Figure 3.4 shows a straightforward divide-and-conquer approach to solving the parallel prefix problem. The only problem with this approach is that the $n/2$-nd parallel sum of the lower-half of the input needs to be concurrently read by all computing elements (processors) of the upper-half. This can either be done simultaneously (i.e. a concurrent-read PRAM is required) or in $n/2$ delayed step (i.e. a slow resulting algorithm).

□

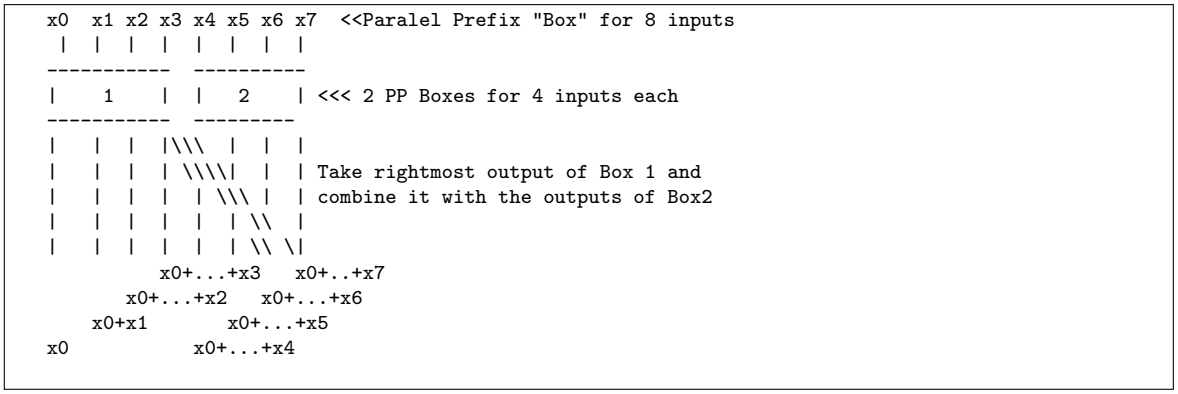


Figure 3.4: Paralle Prefix: Schedule for a CREW solution

3.14.2 PPF sum : an EREW time-optimal solution

Proposition 3.4 (*PPF sums on a EREW PRAM*)**.** *The prefix sums*

$$\sum_{i=0}^{j} x[i] = x[0] \oplus \ldots \oplus x[j],$$

for all $j = 0, \ldots n-1$*, can be computed in an EREW PRAM in* $T = O(\lg n)$*,* $P = n$ *and* $W = O(n \lg n)$*,* $W_2 = O(n)$*. Without loss of generality, let n be a power of two.*

```
1  //   [i:j] = X[i]+X[i+1]+...+X[j] ; [i:i] = X[i]
2  // Input   M[j] = x[j]     where j=0 .. n-1
3  // Output  M[j] = x[0:j]   where j=0 .. n-1
4     void  ParallelPrefix(memory M, int n) {
5        i=1; j=pid() ;
6        while ( i < n ) {
7          if (j > 2**(i-1) ) {
8  // Currently M[j]           = [j+1-2**(i-1):j]
9  // Currently M[j-2**(i-1)] = [j-2**(i-1)+1-2**(i-1):j-2**(i-1)]
10           a = M[ j ] ;
11           b = M[ j- 2**(i-1) ] ;
12           M[j] = a + b ;
13         }
14         i = i + 1 ;
15       }
16    }
```

Figure 3.5: Parallel Prefix: EREW PRAM

Proof. Let $x[j]$ be assigned to $M[j]$. A single processor j is assigned to that memory location and will computer $\sum_{k=0}^{j} x_k$. The algorithm depicted in Figure 3.5 solves the problem in $O(\lg n)$ steps. The iteration index is i and in the i-th iteration, a processor assigned to memory location $M[j]$, reads the element in a location 2^{i-1} position away. If such location does not exist (index less than 0 or greater than n the read is not performed). To this read location processor j combines its content with $M[j]$ so that $M[j-2^{i-1}] + M[j]$ is computed and stored back in $M[j]$. For visualization purposes, the second step is written in two different lines in the computational schedule outline depicted in Figure 3.6.

□

```
   [i:j] to denote xi + ... + xj  ; [i:i] is x[i]
   * :  indicates value of previous row remains the same
   [i:j,j+1,k] means [i,j] op [j+1,k]
0. x0        x1        x2        x3        x4        x5        x6        x7
0. [0:0]     [1:1]     [2:2]     [3:3]     [4:4]     [5:5]     [6:6]     [7:7]
1. *      [0:0,1:1] [1:1,2:2] [2:2,3:3] [3:3,4:4] [4:4,5:5] [5:5,6:6] [6:6,7:7]
1. *         [0:1]     [1:2]     [2:3]     [3:4]     [4:5]     [5:6]     [6:7]
2. *         *      [0:0,1:2] [0:1,2:3] [1:2,3:4] [2:3,4:5] [3:4,5:6] [4:5,6:7]
2. *         *         [0:2]     [0:3]     [1:4]     [2:5]     [3:6]     [4:7]
3. *         *         *         *      [0:0,1:4] [0:1,2:5] [0:2,3:6] [0:3,4:7]
3. *         *         *         *         [0:4]     [0:5]     [0:6]     [0:7]
  [0:0]     [0:1]     [0:2]     [0:3]     [0:4]     [0:5]     [0:6]     [0:7]
  x0      x0+x1     x0+x1+x2  x0+...+x3 x0+...+x4 x0+...+x5 x0+...+x6 x0+...+x7
```

Figure 3.6: Computation schedule for the algorithm of Figure 3.5

3.14.3 PPF sum: Tree-based computation

Proposition 3.5 (*PPF sums on a EREW PRAM*)**.** *The prefix sums*

$$\sum_{i=0}^{j} x[i] = x[0] \oplus \ldots \oplus x[j],$$

for all $j = 0, \ldots n-1$, can be computed on a binary tree $T = O(\lg n)$, $P = n$ and $W = O(n \lg n)$, $W_2 = O(n)$. Without loss of generality, let n be a power of two.

Consider the following variation of parallel prefix on n inputs that works on a complete binary tree with n leaves (assume n is a power of two). It is depicted in Figure 3.7. Action by nodes

1. Non-leaf : If it receives l and r from left and right children, computes l + r and sends it up and send down to its right child the l.
2. Root : Step [1] except nothing is sent up.
3. Non-leaf : If it gets p from parent it transmits it to its left/right children.
4. Leaf : If it holds l and receives p from its parent it sets l = p + l (this order) [note p is the left argument, l is the right one, order matters]

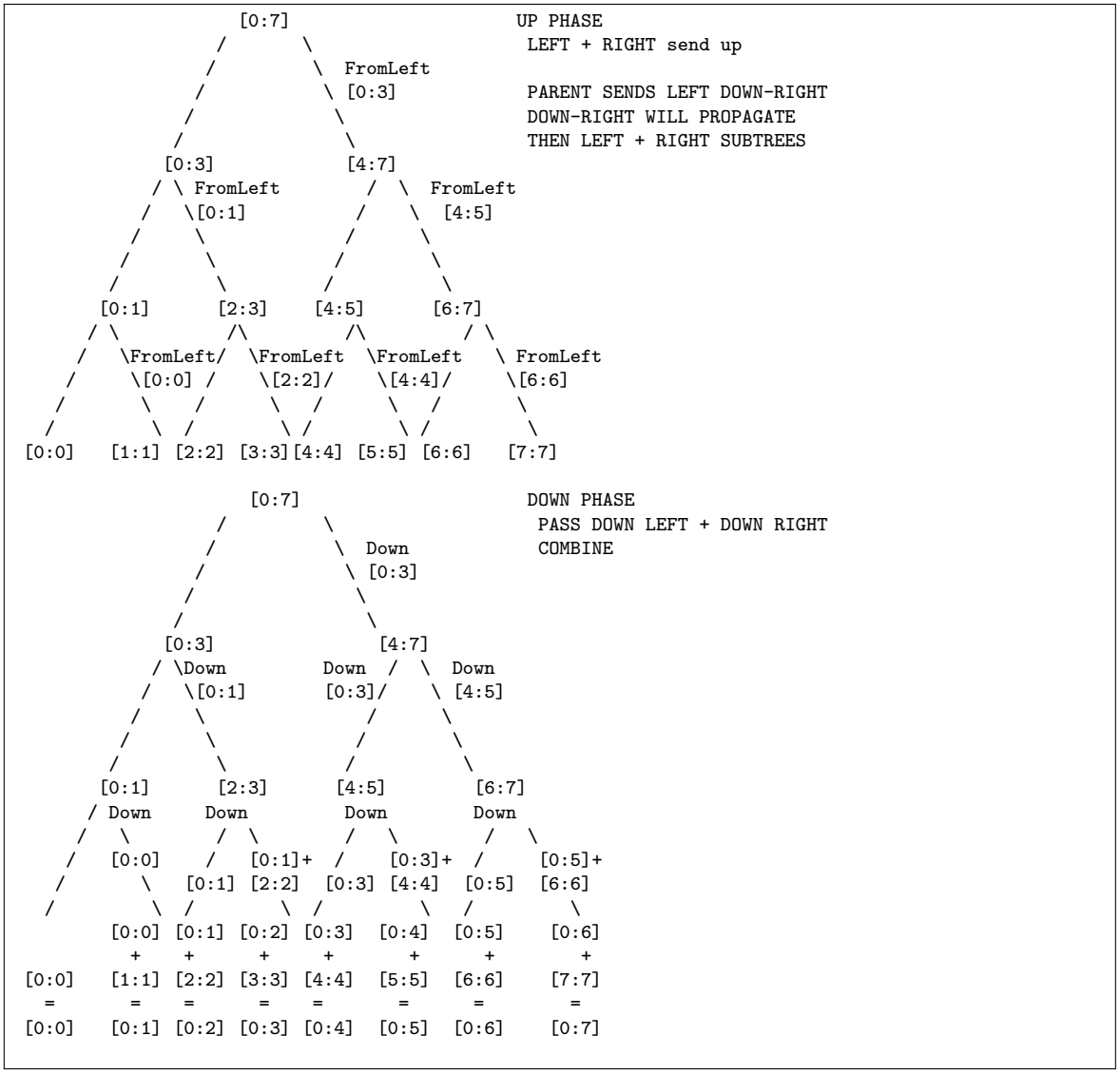


Figure 3.7: PPF sum: optimal tree computation

3.14.4 PPF sum: a recursive version

The parallel prefix algorithm of the previous page (tree-based) depicted in Figure 3.7 requires about $2\lg n+1$ parallel steps, $P=n$ processors and work $W=\Theta(n\lg n)$, and $W_2=\Theta(n)$. One could describe that version due to Ladner and Fischer as follows in Figure 3.8 By rescheduling the computation and using $P=n/\lg n$ processors, the work can be reduced to linear.

Proposition 3.6 (*PPF sums on a EREW PRAM*)**.** *The prefix sums*

$$\sum_{i=0}^{j} x[i] = x[0] \oplus \ldots \oplus x[j],$$

for all $j=0,\ldots n-1$, can be computed on a binary tree $T=O(\lg n)$, $P=n/\lg n$ and $W=W_2=O(n)$. Without loss of generality, let n be a power of two.

```
1   void PPFrecursive(X[0..n-1],Out[0..n-1],n) {
2     Out[0] = X[0] ;
3     if ( n > 1 ) {
4      for( i=0 ; i < n ; i++) in par {
5         a = (2*i   < n) ? X[2*i]   : 0 ;
6         b = (2*i+1 < n) ? X[2*i+1] : 0 ;
7         Temp[i] = a + b ;
8      }
9      Y = PPFRecursive(Temp[0..n/2-1],  Y [0..n/2-1],n/2);
10     for( i=0 ; i < n/2-1 ; i++) in par {
11        Out [2i+1] = Y[i] ;
12     }
13     for( i=1 ; i < n/2-1 ; i++) in par {
14        a = (i-1   < n) ? Y[i-1]   : 0 ;
15        b = (2*i   < n) ? A[2*i]   : 0 ;
16        Out [2i] = a + b ;
17     }
18
19    }
20  }
```

Figure 3.8: A recursive PPF sum algorithm)

3.14.5 PPF sum : an iterative version

An iterative version of the algorithm of Figure 3.8 is depicted in Figure 3.9.

```
1   void PPFiterative(X[0..n-1],Out[0..n-1],n) {
2     for( i= 0 ; i < n ; i++ )  inpar {
3        T[ 0 , i ] = X[ i ] ;
4     }
5     for( j= 1 ; j<=lg(n) ; j++ ) inpar {
6        for( i=0 ; i <= n/2**j - 1; i++) inpar {
7          T[ j , i ] = T[j-1 , 2*i ] + T[j-1, 2*i + 1] ;
8        }
9     }
10    for( j= lg(n) ; j>=0 ; j-- ) inpar {
11        On pid==0 : V[ j , 0 ] = T[ j , 0 ] ;  //Proc 0 only
12        for( i = 1 ;  i <= n/2**j -1 ; i+= 2 ) inpar {
13           V[ j , i ] = V[j+1 , i/2 ] ;
14        }
15        for( i = 0 ;  i <= n/2**j -1 ; i+= 2 ) inpar {
16           V[ j , i ] = V[j+1 , (i-1) /2 ] + T[ j , i ] ;
17        }
18    }
19    Out[ i ] =  V[ 0 , i ] ;
20  }
```

Figure 3.9: An iterative PPF sum algorithm)

3.14.6 An application of parallel prefix: binary addition

Problem 3.4 (*Binary addition*)**.**
Input. *Two n-bit integer $a[0..n-1]$ and $b[0..n-1]$ in little-endian representation.*
Output. *An $(n+1)$-bit integer $c[0..n]$ such that $c = a+b$.*

Solution.
The straightforward grade school addition requires $O(n)$ bit operations and is inherently sequential. This is because in order to compute the k-th bit, the $k-1$-st carry needs to be computed as well. There exists a non-trivial non-obvious parallel solution for this problem that can be done faster than linear time. We will show a parallelization that can be performed on a complete binary tree with n leaves: a parallel-prefix based approach. Its running time will be $2\lg n + 1$ steps.

We shall try for each bit position to find the carry bit required to complete the corresponding addition so that all bit positions can be added in parallel. We shall show that carry computation takes parallel time $\Theta(\lg n)$ time on a binary tree with a computation that is well-known to us: parallel prefix.
Question. How can we find i-th carry bit?

We answer this question by providing a guided example. In the example of Figure 3.10 the following definitions for the notation s, g, p are used. The symbols appear beneath the a_i and b_i bit of a and b.

- s: symbol s is being used to indicate that a carry bit stops its propagation (becase a_i and b_i are both 0),
- g: symbol g is being used to indicate that a carry bit is generated (because a_i and b_i are both 1), and
- p: symbol p is being used to indicate that the current bit position propagates a carry bit (a_i and b_i are either $0,1$ or $1,0$).

Example 3.4. *Let a and b be two 16-bit (unsigned) integers as shown below. Row 1 delineates the 16 bit of a and b, with an extra right most bit whose position is labeled 0 used as a boundary condition bit. Row 2 provides the 16 bit of a: the leftmost one is a_{16} and the rightmost one a_1. Likewise, Row 3 provides the 16 bit of b. Row 4 generates from a_i and b_i a symbol s, g, p according to the definition provided earlier. Be default the symbol generated for index position 0 is an s. Row 5 performs the elementary calculation $(a+b)_i = a_i \oplus b_i \oplus c_{i-1}$, where $\oplus$ = XOR; It is bit addition not integer addition. Thus the result is one if one bit is 0 and the other is 1; it is zero in any other case (both bit 0 or both bit 1). Row 6 will contain the parallel prefix over the elements of Row 4. Finally Row 8 will contain the bit of $a+b$. The information will be generated from Row 5 and Row 7 (is Row 6 where a g becomes a 1 and an s a 0). The question that needs to be address is how do we generate Row 5/Row 6 first?*

Row1		17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
Row2	a		0	1	0	1	1	1	0	0	1	0	0	1	0	0	1	0	-
Row3	b		0	1	1	0	1	0	0	0	0	1	0	1	1	1	0	0	-
Row4	x		s	g	p	p	g	p	s	s	p	p	s	g	p	p	p	s	s
Row5	PPFgs																		
Row6	PPF10																		
Row7	$a+b$																		
Row8	c																		

Figure 3.10: Binary addition example

We derive the following proposition.

Proposition 3.7 (Row 5 and 6: PPF)**.** *The i-th carry bit is one if the leftmost non-p to the right of the i-th bit is a g.*

$\otimes$	s	p	g
s	s	s	g
p	s	p	g
g	s	g	g

Figure 3.11: Associative operator $\otimes$ for the PPF or Row6

Proof. If to the right of the i-th bit we see an s the s will stop a cary. A $0+0$ in the $(i+1)$-st position will never generate a carry bit of 1 to the i-th possition.

If to the right of the i-th bit we see lots of p eventually followed by an s, the s will stop a cary. A $0+0$ in the $(i+j)$-st position will never generate a carry bit of 1 to the i-th possition as the p bit between position i and $i+j$ would only propagate the 0 carry of the $i+j$-th position.

The only case left is that on the right of the i-th bit we have a g or consecutive p followed immediately by a g. □

The previous observation takes the following algorithmic form.

Method 3.1. *Let the i-th bit position symbol (p,s,g) be denoted by x_i. Note that for an n-bit integer, we have $n+1$ symbols for x. To generate* $\mathrm{PPF}[x_0 = s,\ldots,x_n]$ *it will take $\Theta(n)$ serial time or $O(\lg n)$ parallel time using parallel prefix.*

Proof. In order to generate

$$\mathrm{PPF}[x_0 = s,\ldots,x_n]$$

we use for the prefix operation the associative operator $\otimes$ defined in Figure 3.11. □

We revisit the example of Figure 3.10 by providing a solution to it.

Solution 3.1. *We perform the PPF operation with associative operator $\otimes$ to fill Row 5. Row 6 is easy an s in Row 5 in the i-th position gives a 0, a g gives a 1. Row 7 requires a simple XOR operation on the a_i and b_i of Row2 and Row3 respectively. To complete Row 8 we look at Row 7 position i and Row 6 position $i-1$ and compute an XOR of those two bit.*

Row1		17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
Row2	a		0	1	0	1	1	1	0	0	1	0	0	1	0	0	1	0	-
Row3	b		0	1	1	0	1	0	0	0	0	1	0	1	1	1	0	0	-
Row4	x		s	g	p	p	g	p	s	s	p	p	s	g	p	p	p	s	s
Row5	PPFgs		s	g	g	g	g	s	s	s	s	s	s	g	s	s	s	s	s
Row6	PPF10		0	1	1	1	1	0	0	0	0	0	0	1	0	0	0	0	0
Row7	$a+b$		0	0	1	1	0	1	0	0	1	1	0	0	1	1	1	0	
Row8	c		1	1	0	0	0	1	0	0	1	1	1	0	1	1	1	0	

Figure 3.12: Binary addition example solved

□

We summarize the algorithm in Figure 3.13.

Theorem 3.4 (Parallel Addition)**.** *Parallel Addition, using paralle prefix for the carry bit computation requires approximately $T(n) = O(\lg n)$ times step. Processor size is $P = 2n-1$.*

```
1   void BinaryAdd(a[1..n], b[1..n], c[1..n+1], n ) {
2     int i;
3     alloca   x[0..n] ;
4     alloca ppf[0..n] ;
5     alloca apb[0..n] ;
6     /* Row 2  is a[1..n] */
7     /* Row 3  is b[1..n] */
8     /* Row 4  is x[1..n]. Computed below */
9     x[0] = s;
10    for( i = 1 ; i <= n ; i++ ) {
11      if ( a[i] == 1 && b[i] == 1 )
12        x[i] = g ;
13      else
14        if ( a[i] == 0 && b[i] == 0 )
15          x[i] = s ;
16        else
17          x[i] = p ;
18    }
19    /* Row 5 is computed; otimes as in proof */
20    ppf = PPF(x[0..n], otimes );
21    /* Row 6 is computed */
22    for( i = 0 ; i <= n ; i++ ) {
23       ppf[i] = (ppf[i]==g ? 1 : 0 );
24    }
25    /* Row 7 is computed */
26    for( i = 0 ; i <= n ; i++ ) {
27       apb[i] = XOR ( a[ i ] , b[ i ] ) ;
28    }
29    /* Row 8 is computed */
30    for( i = 0 ; i <= n ; i++ ) {
31       c[ i ] = apb[ i ] + ppf [ i-1 ] ;
32    }
33    c[ n ] = ppf[ i ] ;
34    /* DONE */
35    return( c[0..n] ) ;
36  }
```

Figure 3.13: An iterative PPF sum algorithm)

Proof. We used a parallelized version of the algorithm in Figure 3.13. The loop of lines 10-18 requires one parallel step i.e. $O(1)$ time with $n+1$ processors with processor 0 dealing with line 9 and processors $1,\ldots n$ with the steps of the loop. Likewise the loop of lines 22-24 requires $O(1)$ time. Likewise the loop of lines 26-28, lines 30-33 require $O(1)$ time. The PPF of line 20 requires $O(\lg n)$ time. The proof is completed. □

3.15 Segmented Parallel Prefix

A segmented prefix (scan) computation consists of a sequence of disjoint prefix computations. Let the x_{ij} below take values from a set X and let $\oplus$ be an associative operator defined on the elements of set X. Then the segmented prefix computation for

$$x_{11}x_{12}\ldots x_{1k_1} \mid x_{21}x_{22}\ldots x_{2k_2} \mid \ldots \mid x_{m1}x_{m2}\ldots x_{mk_m} \mid$$

requires the computation of all

$$p_{ij} = x_{i1} \oplus x_{i2} \oplus \ldots \oplus x_{ij} \;\; \forall\, 1 \leq i \leq m,\; 1 \leq j \leq k_i$$

In brief the segment separator $\mid$ terminates one prefix operation and starts another one.
One way to deal with a segmented prefix computation in parallel is to extend $(X, \oplus)$ into $(X', \otimes)$ so that

$$X' = X \cup \{|\} \cup \{|x : x \in X\}$$

i.e. X' has more than twice the elements of X: it has all the elements of X, the segment separator $\mid$ and a new element $|x$ which consists of the segment separator and x. The new operator $\otimes$ is associative if we define it as follows.

$\mid \otimes \mid = \mid$,	$\mid \otimes x = \mid x$,	$\mid \otimes \mid x = \mid x$,
$x \otimes \mid = \mid$,	$\mid x \otimes \mid = \mid$,	$x \otimes y = x \oplus y$
$\mid x \otimes y = \mid (x \oplus y)$	$x \otimes \mid y = \mid y$	$\mid x \otimes \mid y = \mid y$

Now, if the length of the segmented prefix formula is n we can assign n processors to solve the problem with parallel prefix in asymptotically the same time. Note that an element in X' requires for its representation no more than 2 extra bits of the storage size of an element of X. If an $\oplus$ computation takes $O(1)$ time (see Table 3.15) so does an $\otimes$ computation (see Table 3.15).

$\oplus$	b
a	$(a \oplus b)$

Then, the new operator $\otimes$ extends $\oplus$ as follows.

$\otimes$	b	$\mid b$
a	$(a \oplus b)$	b
$\mid a$	$\mid (a \oplus b)$	b

3.15.1 Segmented Parallel Prefix: Example and Refinements

Example 3.5. *Let us have three segments.*

$$\{2,3\}\ \{1,7,2\}\ \{1,3,6\}.$$

We create three segments with two barriers in between.

$$2\ 3\ |1\ 7\ 2|1\ 3\ 6.$$

The barrier | pipe symbol is treated as a symbol for the definition of the associative operator by extending the semigroup of numbers. The ppf operation then resolves as follows.

$$2\ 5|1\ 8\ 10|1\ 4\ 10.$$

We can refine the previous algorithm as follows.

Let the operator be defined as $\oplus$ that operates on operands a, b. Then for the segmented parallel prefix problem, we extend operator $\oplus$ to operate on a, b and on extension of the operands that include $|$. Thus we double the input inverse by including as inputs for any a, b the augmented $|a, |b$.

3.16 Matrix Multiplication

Input: Two $n \times n$ matrices $A = (a_{ij})$ and $B = (b_{ij})$.
Output: The product $C = A \times B$, where $C = (c_{ij})$, with $c_{ij} = \sum_k a_{ik} b_{kj}$.

A simple algorithm for multiplying two $n \times n$ matrices on a CREW PRAM with time complexity $T = O(\lg n)$ and $P = n^3$ follows.

For convenience, processors are indexed as triples (i, j, k), where $i, j, k = 1, \ldots, n$. In the first step processor (i, j, k) concurrently reads a_{ij} and b_{jk} and performs the multiplication $a_{ij} b_{jk}$. In the following steps, for all i, k the results $(i, *, k)$ are combined, using the parallel sum algorithm to form $c_{ik} = \sum_j a_{ij} b_{jk}$. After $\lg n$ steps, the result c_{ik} is thus computed.

The same algorithm also works on the EREW PRAM with the same time and processor complexity. The first step of the CREW algorithm need to be changed only. We avoid concurrency by broadcasting element a_{ij} to processors $(i, j, *)$ using the broadcasting algorithm of the EREW PRAM in $O(\lg n)$ steps. Similarly, b_{jk} is broadcast to processors $(*, j, k)$.

The above algorithm also shows how an n-processor EREW PRAM can simulate an n-processor CREW PRAM with an $O(\lg n)$ slowdown.

```
                                          CREW    EREW
1. aij to all            (i,j,*) procs    O(1)    O(lgn)
   bjk to all            (*,j,k) procs    O(1)    O(lgn)
2. aij*bjk at            (i,j,k) proc     O(1)    O(1)
3. parallel sum aij *bjk (i,*,k) procs    O(lgn)  O(lgn)   n procs participate
            j
4. cik = sum aij*bjk                      O(1)    O(1)
          j


              3       3               3
 T=O(lgn),P=O(n ) W=O( n  lgn)   W  = O(n )
                                  2
```

Figure 3.14: PRAM matrix multiplication : concurrent and exclusive-read approaches

3.17 Logical AND operation

We define logical AND as follows.

Problem 3.5 (*Logical AND*)**.**
Input. *Let $X[0\ldots n-1]$ be an array of length n of binary (boolean) values.*
Output. *Evaluate $x = X[0] \wedge X[1] \wedge \ldots \wedge X[n-1]$.*

3.17.1 Sequential logical AND

The sequential problem accepts a $P = 1, T = O(n), W = O(n)$ direct solution.

3.17.2 EREW PRAM logical AND algorithm

Theorem 3.5 (*EREW logical AND*)**.** *By using the parallel sum and associate operator AND ($\wedge$) instead of $+$ the time to compute $x = X[0] \wedge X[1] \wedge \ldots \wedge X[n-1]$. is the time of parallel sum.*

An EREW PRAM algorithm solution for this problem works the same way as the parallel sum algorithm and its performance is $P = O(n)$, $T = O(\lg n)$, $W = O(n \lg n)$ along with the improvements in P and W mentioned for the parallel sum algorithm.

3.17.3 CRCW PRAM

Theorem 3.6 (*CRCW logical AND*)**.** *On a CRCW PRAM we can compute $x = X[0] \wedge X[1] \wedge \ldots \wedge X[n-1]$. in time $T = O(1)$, with $P = n$ and $W = \Theta(n)$.*

Solution.
The solution is outlined in the pseudocode of Figure 3.15. The CRCW PRAM capability required is that of a common PRAM. Processor 0 first writes an 1 in the shared memory cell associated with variable x that will hold the result. If $X_i = 0$, processor i writes a 0 in memory cell x. If no processor writes a 0 it means all X_i are equal to one and conjunction is indeed 1 as initialized by processor 0. If however an X_i is equal to 0 the processor i will read it and write a 0 into x. The conjunction is indeed 0 then as one of the X_i is 0. The correct result x is then made available in this memory cell.

```
1       void logicalAND( X[0..n-1]) {
2         if   pid()  == 0   x=1 ;
3         if X[pid()] == 0   x=0 ;
4         /* implicit return(x) */
5       }
```

Figure 3.15: Logical AND

□

Exercise 3.6. *Give an $O(1)$ CRCW algorithm for LOGICAL OR.*

3.18 Maximum finding

Problem 3.6 (*MAX operation*)**.**
Input. *Let $X[0 \ldots n-1]$ be an array of length n of (comparable) values (defining a total order).*
Output. *Evaluate $x = \max\{X[0], X[1], \ldots, X[n-1]\}$.*

3.18.1 Sequential MAX

The sequential problem accepts a $P = 1, T = O(n), W = O(n)$ direct solution. In fact we need $n-1$ comparisons to find the MAX of n keys.

3.18.2 EREW PRAM MAX algorithm

Theorem 3.7 (*EREW MAX*)**.** *By using the parallel sum and associate operator MAX instead of + the time to compute $x = \max\{X[0], X[1], \ldots, X[n-1]\}$. Its performance is $P = O(n)$, $T = O(\lg n)$, $W = O(n \lg n)$ along with the improvements in P of the parallel sum algorithm.*

Proof.
We may view the PRAM algorithm of Figure 3.1 as a binary tree operating algorithm. Consider the schedule depicted in Figure 3.2. View it as an upside down binary tree; view a + as a MAX operation. The n key values are laid out along the $t = 0$ line. Each one of the $n/2$ key values of the $t = 1$ line is the maximum (though the figure indicates the sum) of two keys of the $t = 0$ lines. At $t = 3$ or in general $t = \lg n$ we end up with one value the MAX (or sum) of the n leaf values.

The EREW PRAM algorithm solution for this problem works the same way as the parallel sum algorithm and its performance is $P = O(n)$, $T = O(\lg n)$, $W = O(n \lg n)$ along with the improvements in P and W mentioned for the parallel sum algorithm. □

3.18.3 CRCW PRAM max algorithm: MAX1

In the remainder we will investigate a CRCW PRAM algorithm. Let binary value Y_i reside in the local memory of processor i or be associated with i.

The CRCW PRAM algorithm `MAX1` to be presented has performance $T = O(1)$, $P = O(N^2)$, and work $W_2 = W = O(N^2)$.

The second algorithm to be presented in the following pages utilizes what is called a doubly-logarithmic depth tree and achieves $T = O(\lg\lg N)$, $P = O(N)$ and $W = W_2 = O(N \lg\lg N)$.

The third algorithm is a combination of the EREW PRAM algorithm and the CRCW doubly-logarithmic depth tree-based algorithm and requires $T = O(\lg\lg N)$, $P = O(N)$ and $W_2 = O(N)$.

Theorem 3.8 (*MAX1*). *On a CRCW PRAM algorithm MAX1 computes* $x = \max\{X[0], X[1], \ldots, X[n-1]\}$. *in time* $T = O(1)$, *with* $P = O(n^2)$ *and work* $W_2 = W = O(n^2)$.

Solution.
The setup requires the use of n^2 processors. Each processor can have an ID from 0 to $n^2 - 1$. However we prefer to number them with a pair of values (i, j) where $0 \le i, j \le n-1$. We assume without loss of generality all keys are distinct. The if statement of line 7 compares $X[i]$ with $X[j]$ by one processor and an 1 is stored in $M[i][j]$ if $X[i] > X[j]$ and 0 otherwise. If $X[t]$ is the MAX then row t of M i.e. all values $M[t][*]$ would contain an 1, since $X[t] > X[j]$ for all j except for $j = t$ and for $j = t$ $X[t] \ge X[t]$ so no harm.

In line 9, we calculate the logical AND of $M[i][0..n-1]$ for all $i = 0, \ldots, n-1$. Only for $i = t$ all the $M[t][j]$ values would be one due to $M[t]$ being the maximum. Thus $Y[t]$ would be 1 where as $Y[i], i \ne t$ would be $Y[i] = 0$.

In line 10 processor t realizes $Y[t] = 1$ and thus write $X[t]$ the maximum value into res that is then returned as the MAX value.

```
1        void  max1 ( X[ 0 .. n-1 ] ) {
2        // Processor pid() in 0.. n**2 -1
3        // maps to  pid().i and pid().j,
4        // where pid().i = pid() / n and
5        //       pid().j = pid() % n.
6
7        if X[pid().i] >=  X[pid().j] M[pid().i][pid().j]=1 ;
8           else                      M[pid().i][pid().j]=0 ;
9
10       Y[i] = logicalAND(M[i][0..n-1]) ;
11       if Y[pid()] == 1  res= X[pid()] ;
12       return( res ) ;
13       }
```

Figure 3.16: Algorithm MAX1

□

3.18.4 CRCW PRAM max algorithm : MAX2

In preparation of algorithm `MAX2` we introduce a **doubly logarithmic-depth tree.**

Definition 3.1. *Let $n = 2^{2^m}$, for some integer $m > 0$. A* doubly logarithmic-depth tree *of order n (dldt-n) has n leaves and is defined as follows.*

(1) The root of the tree at level 0 has $2^{2^{m-1}} = n^{1/2}$ children in level 1.

(2) Each node at level 1 has $2^{2^{m-2}} = n^{1/2^2}$ children in level 2.

(3) Each node of level i has $2^{2^{m-(i+1)}} = n^{1/2^{i+1}}$ children in level $i+1$.

(4) Each node of level $m-1$ (the level before the last) has $2^{2^{m-m}} = n^{1/2^m} = 2$ children in level $m = \lg\lg n$.

(5) The nodes of level m are the **leaves** *of the tree.*

Some properties of a *doubly logarithmic-depth* tree are listed below.

- The **height** of the tree is $m = \lg\lg n$.
- A node of level i has $2^{2^{m-(i+1)}}$ children in level $i+1$.
- The TOTAL number of level i nodes is $2^{2^{m-1}}2^{2^{m-2}}\ldots 2^{2^{m-i}} = 2^{2^m-2^{m-i}}$.
- The product $(2^{2^{m-i-1}})^2 \times 2^{2^m-2^{m-i}}$ is $O(2^{2^m}) = O(n)$.

Theorem 3.9 (*MAX2*)**.** *On a CRCW PRAM algorithm MAX2 computes $x = \max\{X[0], X[1], \ldots, X[n-1]\}$. in time $T = O(\lg\lg n)$, with $P = O(n)$ and work $W_2 = W = O(n\lg\lg n)$.*

Proof.
Algorithm MAX2 achieves better work performance than algorithm MAX1, even if it is slower. It exhibits the following performance: $T = O(\lg\lg n)$, $P = O(n)$, and $W = W_2 = O(n\lg\lg n)$.

Algorithm `Max2` works as follows:

In line 2 we arrange the n keys on the leaves of a doubly logarithmic tree with n leaves and depth $m = \lg\lg n$.

The algorithm works bottom to top level by level starting from level $m-1$ (the leaves that hold the keys are at level m).

A node at level i, call it u, will compute the MAX of its children by utilizing algorithm MAX1 (line 6-7). If the number of children of a node u at level i is $q = 2^{2^{m-(i+1)}}$ the number of processors assigned to computing at u the maximum of its children values would be $(2^{2^{m-i-1}})^2$ (line 7). The number of nodes at level i is $2^{2^{m-1}}2^{2^{m-2}}\ldots 2^{2^{m-i}} = 2^{2^m-2^{m-i}}$ and thus the total number of processors used on those nodes at level i would be the product

$$(2^{2^{m-i-1}})^2 \times 2^{2^m-2^{m-i}} = n$$

We have exactly n processors available for level i and every level from level $m-1$ all the way to the root of level 0.

Moving upwards we eventually reach the root of level 0 that will hold the MAX of the n keys (line 9).

□

```
1       void  max2 ( X[ 0 .. n-1 ] ) {
2         Arrange X[0..n-1] on  leaves(dldt(n));
3         for(i=m-1 ; i >= 0 ; i-- )  {
4           // Using (2**(2**(n-i-1)))**2 processors
5           for each u of dldt(n) of level i
6            u = max1( children(u) )
7                utilizing #processors = square(level i+1 children);
8         }
9         return(root(dldt(n));  // contains MAX
10      }
```

Figure 3.17: Algorithm MAX2

3.18.5 CRCW PRAM max algorithm : MAX3

Theorem 3.10 (*MAX3*). *On a CRCW PRAM algorithm MAX3 computes* $x = \max\{X[0], X[1], \ldots, X[n-1]\}$. *in time* $T = O(\lg\lg n)$, *with* $P = O(n/\lg\lg n)$ *and work* $W_2 = W = O(n)$.

Proof.
Algorithm MAX3 has two rounds of computation.

In round 1, each one of $n/\lg\lg n$ processors is assigned $\lg\lg n$ keys each. In $\lg\lg n$ time each processor finds a partial MAX among its $\lg\lg n$ keys. There are $n/\lg\lg n$ partial leftovers.

In round 2, algorithm MAX2 is run on dldt($n/\lg\lg n$), a doubly logarithmic depth tree of the remaining $n/\lg\lg n$ partial maxima.

The maximum is then on the root of the dldt($n/\lg\lg n$) at the completion of round 2.

With regard to the code of Figure 3.18 lines 2-4 map to round 1. The running time of these steps is $T_1 = \lg\lg n$ with $P_1 = n/\lg\lg n$, and $W = W_2 = \Theta(n)$.

Lines 5-11 is a copy of MAX2 but on a dldt($n/\lg\lg n$). Time is $T_2 = \lg\lg n$, and $P = n/\lg\lg n$ with $W = W_2 = \Theta(n)$.

```
1       void  max3 ( X[ 0 .. n-1 ] ) {
2         Assign lg(lg(n)) keys per processor among X[0..n-1]
3         In parallel  n/lgg(n) partial maxima are determined;
4         Results in X[0.. n/lg(lg(n))-1] ;
5         Arrange X[ 0..n/lg(lg(n))-1 ] on  leaves(dldt(n/lg(lg(n))));
6         for(i=m-lg(lg(lg(n)))-1 ; i >= 0  ; i-- )  {
7           // Using (2**(2**(n-i-1)))**2 processors
8           for each u of dldt(n) of level i
9            u = max1( children(u) )
10               utilizing #processors = square(level i+1 children);
11        }
12        return(root(dldt(n));  // contains MAX
13      }
```

Figure 3.18: Algorithm MAX3

□

Question 3.3. *Is there a* $p \leq n$ *processor CRCW PRAM algorithm that finds the maximum of N keys faster than* `Max2` *or* `Max3`*?*

3.18.6 A matching Lower bound and Turan's Theorem

Definition 3.2. *On an undirected graph $G = (V,E)$, an* **independent set** *is a set of vertices such that no two vertices are connected by an edge.*

Definition 3.3. *On an undirected graph $G = (V,E)$, a* **clique** *is a set of vertices such any two vertices are connected by an edge.*

A clique on n vertices is the "complement" of an independent set on the same n vertices. The complete graph on n vertices is a clique on n vertices by default. A clique of n vertices has $n(n-1)/2$ edges.

Theorem 3.11 (Turan)**.** *Let $G = (V,E)$ be an undirected graph, where $|V| = n$ and $|E| = m$. Graph G has an independent set of size at least $n^2/(2m+n)$.*

Another formulation of Turan's Theorem is the following one.

Theorem 3.12 (Alternative Turan)**.** *Let $G = (V,E)$ be an undirected graph, where $|V| = n$ and $|E| = m$. If graph G has no p clique then it has at most $(1-1/(p-1))n^2/2$ edges.*

Proof. If $n \leq p-1$ then G does not have obviously a p-clique and G has at most $n(n-1)/2$ edges. It is obvious that $n(n-1)/2 \leq (1-1/(p-1))n^2/2$ by elementary calculation.

Thus the interesting case left is $n \geq p$. If graph G has the maximum number of edges but does not have a p-clique it must have a $(p-1)$-clique. This is because otherwise we could add edges to G to create such a $(p-1)$-clique; this would contradict the maximality of edges of G. Call C a $(p-1)$-clique of G. C has $(p-1)(p-2)/2$ edges. Call G' the graph G without C, i.e. $G' = G-C$. Graph G' has m' edges. Let k be the number of edges that go from G' to C. By induction on G' we have that $m' \leq (1-1/(p-1))(n-p+1)^2/2$. Since G does not have a p-clique every vertex of G' is connected to at most $p-2$ vertices of C (since if it were connected to all the vertices of C a p-clique would have been formed). Thus $k \leq (p-2)(n-p+1)$.

Therefore the number of edges m of G is

$$m \leq (p-1)(p-2)/2 + (1-1/(p-1))(n-p+1)^2/2 + (p-2)(n-p+1) \leq (1-1/(p-1))n^2/2.$$

□

If we solve this inequality for p we get

$$p-1 \geq \frac{n^2}{n^2-2m}$$

Therefore an equivalent formulation is that G has a $p-1$ clique of size at least $\frac{n^2}{n^2-2m}$.

Now take the complementary graph (where an edge becomes a non-edge and a non-edge becomes an edge). The complement of G has $N = n$ vertices and $M = n(n-1)/2 - m$ edges. From the latter we get that $2m = n(n-1) - 2M = n^2 - n - 2M$. A $p-1$ clique in G becomes an independent set in its complement whose size is at least $\frac{n^2}{n^2-2m} = \frac{N^2}{N^2-2m} = \frac{N^2}{N^2-n^2+n+2M} = \frac{N^2}{N+2M}$, noting that $N = n$. This latter bound is the expression in the first version of Turan's theorem.

An easy corollary is that a graph with n/k vertices, $k \geq 1$, and n edges has an independent set of size at least $n/4k^2$.

3.18.7 Lower bounds for MAX finding

We are going to show some lower bounds on finding the MAXIMUM of n keys. The model of computation we are going to use is the decision tree model, and in fact the **parallel decision tree model** where we allow p processors to work at any time step each one performing a single comparison between two keys. The decision tree model in the case $p = 1$ was used to prove the lower bound $\Omega(n \lg n)$ for comparison-based sorting. Note that this model deals with **information gathering** to compute the MAXIMUM. One also needs to process this information to derive the MAXIMUM. The model assume that processing is **for free**. We can do so because we plan to establish a lower-bound (minimum possible running time) not to establish an algorithm for realizing this bound.

We know from the sequential setting that finding the MAXIMUM of n keys requires at least $n-1$ comparisons.

Theorem 3.13. *MAX can be found in ONE parallel step with $n(n-1)/2$ processors.*

Proof. n keys allow $n(n-1)/2$ pairs and thus comparisons to be realized to obtain all the information required to find the MAXIMUM (one needs to do some additional processing eg. establishing the rank but that it is for free!). If we have that many processors each one responsible for one comparison, this concludes the proof. □

Theorem 3.14. *In order to find MAX in ONE parallel step we need at least $n(n-1)/2$ processors.*

Proof. In order to prove a lower bound, we use a proof by contradiction. Suppose we can do it with one fewer processor $P = n(n-1)/2 - 1$. Since there are $n(n-1)/2$ pairs of keys to compares, one such pair is not compared. Call the keys of the pair x, y. What we are going to show, by playing the role of an adversary, is that we can set up the values of the n input keys so that the missing comparison of the x, y is the crucial one to establish the maximum. We thus play the role of an adversary whose only mission is to make the algorithm that uses P processors to fail. To do so, we set the results of the comparisons in such a way that x, y are the MAX and SECOND MAX keys. Thus the comparison between x, y (that is not being performed) is CRUCIAL in determining the MAXIMUM of the n keys. Since it is not performed we cannot find the MAX with P processors. Contradiction is established. □

Question. What can you prove about the SECOND MAX key?

An interesting question is how many processors one needs to use to find the MAX of n keys not in one parallel step but in two parallel steps.

Problem 1. Find MAX in two steps with $O(n^{3/2})$ processors.

Hint. Split n keys into groups of $\sqrt{n}$. Compute MAX of each group in first step, and MAX of MAXes in second step.

Problem 2. Find MAX in two steps with $O(n^{4/3})$ processors.

Hint. Optimize the splitting. $\sqrt{n}$ might not be optimal.

Problem 3. Show that MAX in two steps requires $\Omega(n^{4/3})$ processors.

The Proof is a repetition of the arguments of the proof of Theorem 1.

3.18.8 A matching lower bound

Theorem 3.15 (Valiant)**.** *Computing the maximum of n keys requires at least* $\lg\lg n$ *parallel steps with $p \leq n$ processors.*

Proof. (By induction) It is proved by what we call **an adversary argument** through induction. An **adversary** for this problem is allowed to choose the input keys by modifying their values in such a way so as to force the

algorithm to run for at least $\lg\lg n$ steps. These modifications should not invalidate, however, the operations of the algorithm already performed.

View step i as a graph $G_i = (V_i, E_i)$. The vertices are the keys and the edges are the comparisons performed at step i.

Consider initially graph G which becomes G_1. Since any max finding algorithm can perform no mores than $p \leq n$ comparisons at any time step, $|E_1| \leq n$ and $|V_1| = n$. So by Turan's theorem G_1 has an independent set of size $n/4$. Call this set of keys I_1. Consider now the computation where all the I_1 keys are the larger than anything in $V_1 - I_1$. In this case every node in I_1 wins in the comparisons performed and is a candidate for the maximum. Since I_1 is an independent set we have no information on the relative order of the keys in I_1.

Consider now G_2 and take the intersection of G_2 and I_1. It has $\leq n$ edges (since $p \leq n$ can be performed at a time) and at least $n/4$ vertices (lower bound for I_1 size. So the graph G_2 intersected by I_1 has an independent set of size at least $n/64$ and call it I_2. Repeating the same thing for G_i and I_{i-1} we end up with an independent set of size $n/2^{2^{i+1}-2}$. So if we repeat this procedure about $\Omega(\lg\lg n)$ times the independent set will drop below 2 and the maximum will be established. This takes however $\Omega(\lg\lg n)$ parallel steps.

□

Let us prove Problem 3.

Proof. (Problem 3). We set up a graph, just as in Theorem 1, with n vertices. Let us have p processors. In one step they can force p comparisons. By Turan's theorem the graph on n vertices and p edges must have an independent set of size at least $k = n^2/(n+2p)$. We can set the values of the keys or equivalently determine the output of the comparisons so that the keys of the named independent set are all candidates for the MAX. Since they form an independent set none has been compared to any other key of the set. Thus in the second round we can find the MAX among these k keys in $k(k-1)/2$ comparisons/processors using Lemma 1, if and only if we can afford to do so i.e. $p \geq k(k-1)/2$, which leads to $p = \Omega(n^{4/3})$. □

Problem 4. Is there an algorithm that finds the MAX in $O(\lg\lg n)$ parallel steps using n processors? What is the work of the algorithm?

Long Hint. Consider n keys. Splits into $n/3$ groups of 3 keys each. For each group we can detemine the MAX in $3(3-1)/2 = 3$ comparisons using 3 procs per group. Total number of processors used is $n/3 \cdot 3 = n$. Thus we are left with determining the max of $n/3$ keys.

Take the $n/3$ Maxima, and split them into groups of 7. We have $n/(3 \cdot 7)$ groups of 7 keys. Each group requires $7(7-1)/2 = 21$ processors to find the MAX of the group. Total processors used is $n/21 \cdot 21 = n$, that we can afford to. Thus after the second second it suffices to find the max of $n/21$ keys to determine the MAX of the original n keys.

How do we split the $n/21$ keys next? What is the pattern?

Say we at some point we end up having n/s keys. We split them into groups of t so that $t(t-1)/2$ comparisons/processors per group. You can fill in the details to show that this way $\lg\lg n$ can be achieved.

3.19 PRAM Integer Sorting: Count-Sort

We introduce a special case of sorting n keys that are integers whose values are in the range of $[0..\lg n-1]$, where $\lg n$ we assume it is also an integer. This is the well-known problem of count-sort. Sorting n keys in the range $0..k-1$ requires sequential time $O(n+k)$. We show below that sorting n integer keys in the specified range with $P = n/\lg n$ processors can be done in time $T = O(\lg n)$ using $W = W_2 = \Theta(n)$. Note that count-sort does not sort in place i.e. we are going to use different arrays for input and output. Let $M[1..n-1]$ be the input and $N[1...n-1]$ the output arrays.

The idea is to assign $\lg n$ keys per processor; for example if keys are in $M[1..n-1]$, processor $i = 0,\ldots,p-1$ deals with keys $i\lg n + j + 1$, where $j = 0,\ldots,\lg n-1$. The P processors collectively create an $P \times \lg n$ array C initialized to 0. We assign to each processor a single row of the array. Thus initialization take $O(\lg n)$ steps to zero the entries of row i assigned to processor i. Entry (i, j) of the table would indicate how many keys with value j processor i is assigned to. This information can be collected easily: processor i scans its keys and for each key it updates the counters of row i of C. Total time is $O(\lg n)$. Then a parallel prefix operation is formed. It consists of the first column, the second column and so on the last column. The purpose is to count all the keys with values 0 before the ones with value 1, before those with value 2 and so on. Note that the prefix sequence is of length n. If we have n processors we can work it out in $O(\lg n)$ time. Now that we have only P processors we can invoke Brent's principle to do it in $O(\lg n)$ time as well but with $O(n)$ work. Note that during the prefix operation C is not overwritten; a new array D will hold the results. After the prefix operation if the entry that corresponded initially to the (i, j) element of C has value t, this means that processor i will store the keys assigned to it with value j to consecutive positions ending with memory location t of the output. Thus if for example we have that $C(i, j) = 3$, then N[t-2] N[t-1] and N[t] will hold the three keys with value j of processor i. Processor i, after D becomes available scans its keys and writes them into N as appropriately.

Theorem 3.16. *Sorting n keys in the range $[0..\lg n-1]$ with $P = n/\lg n$ processors can be done in time $T = O(\lg n)$ and work $W = W_2 = \Theta(n)$.*

Applying this theorem t times (i.e. use t rounds of count-sort to obtain a radix-sort algorithm) the following is derived.

Theorem 3.17. *Sorting n keys in the range $[0..\lg^t n-1]$ with $P = n/\lg n$ processors can be done in time $T = O(t\lg n)$ and work $W = W_2 = \Theta(tn)$.*

3.19.1 Parallel Count-Sort: Sequential algorithm

3.19.2 Parallel Count-Sort: An example

```
1   CountSort(I[0..n-1],O[0..n-1],n,k)
2    for(i=0;i<k;i++)
3       C[i]=0;            // Initialize the Counter Array of length k
4    for(j=0;j< n;j++)     // If key I[j] is  m increment C[m] by one
5       C[I[j]] ++;        // Result: C[m] : number of keys with value m
6    for(i=1;i<k;i++)      // C[i] becomes #keys with values at most i.
7      C[i] = C[i]+C[i-1];// C[i]-1 is the last index of O holding an i
8    for(j=n-1;j>=0;j--) {
9       C[I[j]]--;
10      O[C[I[j]]]=I[j
11   }
```

Figure 3.19: Algorithm CountSort

```
1
2  Step 1: Split n keys on p processors ;
3        P:    0          1          2          3
4  ----------------------------------------
5        I: 0 2 2 0   1 3 0 0   1 3 0 3  2 1 3 2 [n=16, lgn=4]
6
7  Step 2: Initialize C;
8        P:    0          1          2          3
9  ----------------------------------------
10     C: 0    0          0          0          0
11        1    0          0          0          0
12        2    0          0          0          0
13        3    0          0          0          0
14
15  Step 3: Count keys + update C accordingly;
16       P:    0          1          2          3
17 ----------------------------------------
18     C: 0    2          2          1          0
19        1    0          1          1          1
20        2    2          0          0          2
21        3    0          1          2          1
22
23  Step 4: Parallel Prefix Sum (row major)
24          [Transpose values before ppf]
25       P:    0          1          2          3
26 ----------------------------------------
27   PPF: 0    2          4          5          5
28        1    5          6          7          8
29        2   10         10         10         12
30        3   12         13         15         16
31
32  Step 5: Output row major
33       P:    0          1          2          3
34 ----------------------------------------
35   PPF: 0    0          0          0          0
36        1    0          1          1          1
37        2    2          2          2          2
38        3    3          3          3          3
```

Figure 3.20: Algorithm CountSort example

3.20 Comparison or comparator networks

A comparison network consists of wires (also called lines or channels) and comparators. The values of the network are atomic: they cannot be subdivided nor duplicated. The values travel in a number of wires also known as channels that will be represented with horizontal lines. (An alternative representation uses vertical lines.) The inputs are presented on the far left (or alternatively top) and after they travel through the network in some permuted order on the far right (or alternatively bottom). The network is notionally divided into a finite number of levels. We may call level 0 the level of the inputs. Each one of subsequent levels contains a number of comparators. Each comparator is placed on two wires. At most one comparator is placed on any wire at any given level. We shall denote a comparator with a vertical line with the contact points indicated by an x in the text figures to be shown. When two wires are interrupted by a comparator, the wires' values are fed into the comparator and the appropriate value are output from the comparator, as to be explained.

For input $x_1,\ldots,x_n$ on wires $1,\ldots,n$ we shall say that wire i transfers value v at level k on $x_1,\ldots,x_n$ if

a. either $k=0$ and $v=x_i$,

b. or $k>0$, there is no comparator on wire i at level k and wire i transfers value v at level $k-1$,

c. or $k>0$, there is a comparator on wire i and wire j at level k, and $j<i$, wire i transfers value v_i at level $k-1$, wire j transfers value v_j at level $k-1$, and $v=\max\{v_i,v_j\}$.

d. or $k>0$, there is a comparator on wire i and wire j at level k, and $j>i$, wire i transfers value v_i at level $k-1$, wire j transfers value v_j at level $k-1$, and $v=\min\{v_i,v_j\}$.

The output is the set of values transferred by the wires at the final level. Cases (c) and (d) describe the action of a comparator as specified in a simple form below.

comparator is a box with two inputs a and b, fed through wires and two outputs c and d feeding into wires. The comparator compares a and b and outputs through c the minimum of a and b and through d the maximum of a and b.

The box of Fig. 1 in most cases is simplified into the vertical line of Fig. 2 Each comparator takes $O(1)$ time steps to compare the two keys and output them in proper order (from now one 1 step).

A comparison network consists of wires and comparators. A **comparator** is a box with two inputs x and y and two outputs a and b. The comparator compares x and y and outputs through a the minimum of x and y and through b the maximum of x and y. The box of Fig 1. in most cases is simplified into the vertical line of Fig 2. Each comparator takes $O(1)$ time steps to compare the two keys and output them in proper order. The wires transmit values as a whole (think of parallel communication rather than serial) from left to right. If multiple comparators are attached to a horizontal line that transfers the input values the result of a preceding comparator must become available before a succeeding comparator starts performing the comparison required. The **depth** of a sorting network is the maximum number of comparators attached in a path (not line) from an input to an output. Thus if all input lines are of depth 0, and the input wires to a comparator are of depth d_1 and d_2 then the two output wires of the comparator are of the same depth $\max\{d_1,d_2\}+1$.

3.21 Sorting network

Definition 3.4. *A sorting network is a comparison network that always sorts its inputs. That is the inputs appear in ascending (non-decreasing) order, smallest at top, largest at bottom on the right side which is the final (last) level of the network.*

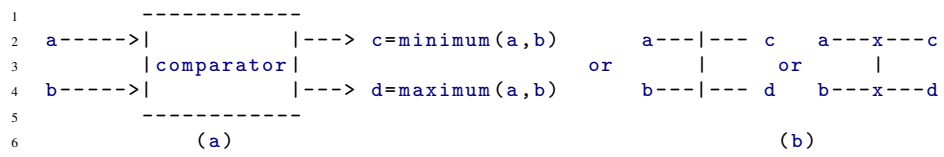


Figure 3.21: Comparator

A sorting network is a (comparison) network that always sorts its inputs by performing comparisons only (i.e. countsort, radix sort can not be implemented in such networks). Sorting networks are special cases of the broader class of networks known as comparison networks.

A sorting network is a comparison network whose output lines generate the input sequence in monotonically increasing order top-to-bottom (i.e. smallest element appear on top-right output line and largest in bottom-right).

Note 1. This means that countsort, radix sort can not be implemented in sorting networks that uses comparators.

Note 2. Sorting networks are thus special cases of the broader class of networks known as comparison networks.

Note 3. The wires transmit values as a whole (think of parallel port communication rather than serial port communication) from left to right.

Note 4. If multiple comparators are attached to a horizontal line that transfers the input values the result of a preceding comparator must become available before a succeeding comparator starts performing the comparison required.

The **depth** of a comparison network is the number of its levels. It is thus the maximum number of comparators attached in a path (not line/wire) from an input to an output. Thus if all input are of level 0 implying also a depth 0, and the input wires to a comparator are of level d_1 and d_2 then the two output wires of the comparator are of the same depth $\max\{d_1, d_2\} + 1$.

The depth of a sorting network is the depth of the corresponding comparison network.

The **size** of a sorting network is the number of its comparators i.e. the size of the corresponding comparison network.

The comparator as define is sometimes called a MIN-MAX comparator. Like-wise one can define a MAX-MIN comparator where the top line generator the maximum of a and b. One could even build a network of MIN-MAX and MAX-MIN comparators. A result due to Floyd and Knuth states that a network of MIN-MAX and MAX-MIN comparators can result into a network of MIN-MAX comparators of the same depth and size.

A sorting network can be transformed into a serial (sequential) sorting algorithm if we describe in a code the appropriate sequence of comparisons generated or performed by the sorting network. The 'running time' is derived from the size of the network. Moreover a sorting network can be transformed into a parallel sorting algorithm if we extract from the network structure of the sorting network the necessary parallelism. The 'parallel runing time' is then the depth of the network.

The sorting algorithm implied by a sorting network is an **oblivious sorting algorithm**. An oblivious algorithm is an algorithm whose actions are always the same and independent of the input and output: in an oblivious sorting algorithm the sequence of comparisons performed is the same for all inputs and outputs.

The sorting algorithm implied by a sorting network is an **oblivious sorting algorithm**. An oblivious algorithm is an algorithm whose actions are always the same and independent of the input and output: in an oblivious sorting algorithm the sequence of comparisons performed is the same for all inputs and outputs.

Odd-even transposition sort is an oblivious sorting algorithm. In an even round the keys indexed 0-1, 2-3,

4-5, 6-7 are compared (and swapped as needed). In an odd round the keys indexed 1-2, 3-4, 5-6 are compared and swapped. (One can call odd-even transposition sort an unoptimized version of bubble sort.)

3.21.1 Finding the maximum

We present first some comparison networks for simple operations. Later on we build on them in creatin a sorting network of increasing complexity.

Suppose we are interested in Finding the maximum of n values.

Suppose we are interested in Finding the maximum of n values. We shall use the term values interchangeably with the term keys. Note that keys (or values) are comparable i.e. a total order is defined on them. Thus for two keys x and y a comparator comp(x,y) yields the MIN(x,y) on the top line and MAX(x,y) on the bottom line. (In case of equality the input order is preserved.)

Let n be a power of two.

One way to build a comparison network for finding the maximum is divide and conquer. Suppose that a comparison network with n input lines and n output lines finds the maximum of n values fed through the input lines and this maximum appears through the bottom wire of the output. We can construct this network recursively by first constructing two networks, one that finds the maximum of the first $n/2$ values and one that finds the maximum of the remaining $n/2$ values. Then, a single comparison determines the maximum of the two maxima, i.e. the maximum of the n values.

We want to build a comparison network for MAX called MAX(n) that would consist of n input wires containing n input keys (value), and n output wires, where the last (bottom-most) contains the MAX of the n input and the other $n-1$ wires are irrelevant (eg they contain the remaining $n-1$ inputs. For the sake of this exposition let n be a power of two.

A recursive construction: We can use divide-and-conquer in constructing the MAX(n) network that computes the maximum of n input values $x_0, \ldots x_{n-1}$.

One way to construct MAX(n) of $x_0, \ldots x_{n-1}$ is by using divide-and-conquer to:

(a) first constructing a MAX(n/2) of $x_0, \ldots, x_{n/2-1}$ that computes the MAX of the first $n/2$ values in the bottom-most output wire,

(b) then constructing a second MAX(n/2) of $x_{n/2}, \ldots, x_{n-1}$, that computes the MAX of the next $n/2$ values in the bottom-most output wire, and then

(c) combining by comparing the bottom-most output wires of both networks to compute the MAX of the n values as the maximum of the two generated partial maxima.

Let $S(n)$ be the size of $MAX(n)$. Then, a simple recursive formulation shows

$$S(n) = 2S(n/2) + 1$$

We have as a base case $S(2) = 1$. A solution for $S(n)$ is thus $S(n) = n - 1$

Let $D(n)$ be the depth/delay of $MAX(n)$. Then,

$$D(n) = D(n/2) + 1$$

We have $D(2) = 1$. A solution for $D(n)$ is $D(n) = \lg n$.

3.21.2 Sorters

Figure 3.23 describes comparator networks for sorting a fixed number of inputs.

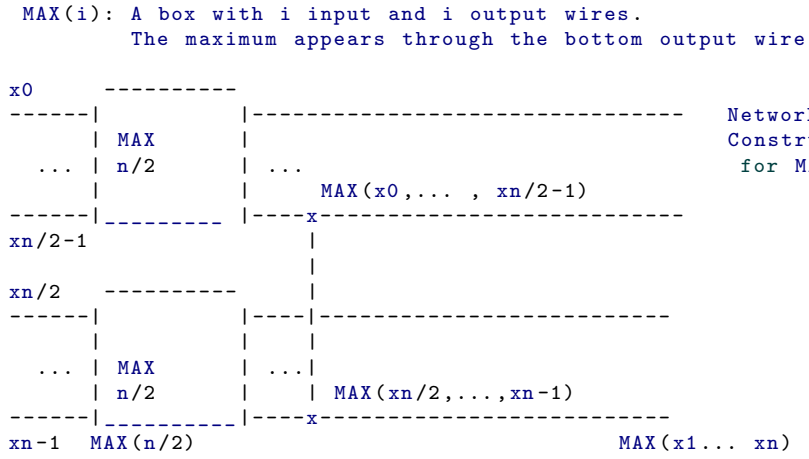


Figure 3.22: Comparator network for MAX

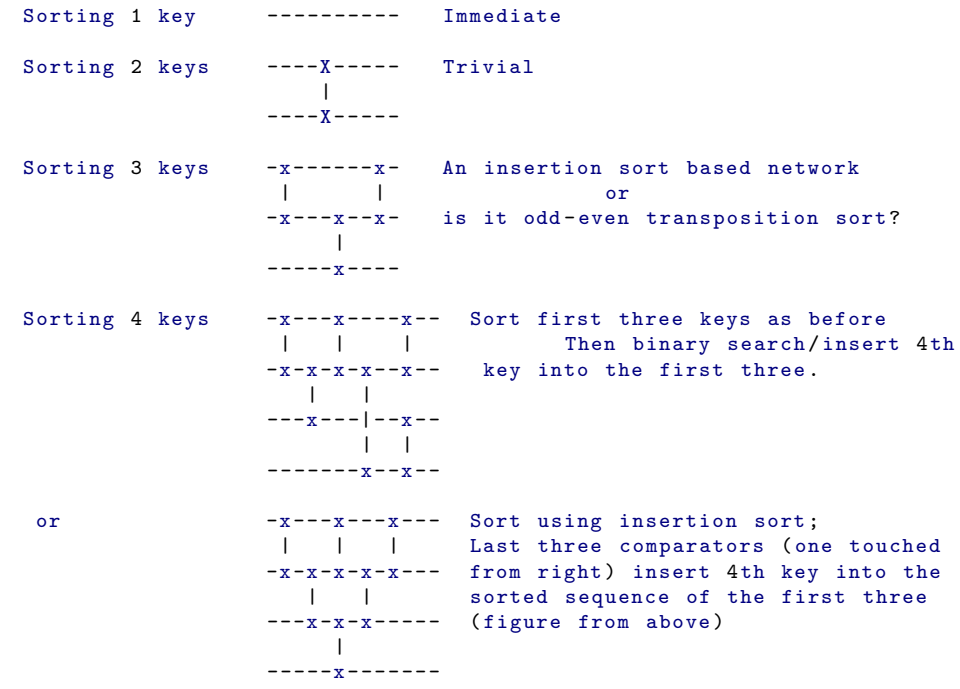


Figure 3.23: Comparator networks for sorting 1,2,3,4 keys

3.21.3 0-1 Sorting Lemma

The 0-1 Sorting Lemma also referred to as 0-1 (Sorting) principle can be used to show that a comparison network is indeed a sorting network. The 0-1 sorting lemma says that if a sorting network sorts all inputs of zeroes and ones correctly, then it can sort any type of inputs as well, be them floating point numbers, integers, etc. Note that the lemma applies to sorting "algorithms" that are oblivious and use only comparisons and do not inspect the values of the input.

Lemma 3.1 (Lemma MonotonicP)**.** *If a comparison network transforms $a = \langle a_0, \ldots, a_{n-1} \rangle$ into a $b = \langle b_0, \ldots, b_{n-1} \rangle$, then for any monotonically increasing function f, the network transforms, $f(a) = \langle f(a_0), \ldots, f(a_{n-1}) \rangle$, into $f(b) = \langle f(b_0), \ldots, f(b_{n-1}) \rangle$.*

Proof. **MonotonicP.** A proof of the monotonic property will be by induction.
Base case. A single comparator has the monotonic property. Suppose that the inputs to a comparator are x and y. If $x \leq y$, then $f(x) \leq f(y)$ by the monotonicity of f. In the former case x is output on the top output line and in the latter case $f(x)$ is output on the top output line. A similar observation applies to the case $x \geq y$, where $y, f(y)$ are on the top output line. Therefore $\min(f(x), f(y)) = f(\min(x,y))$, and $\max(f(x), f(y)) = f(\max(x,y))$.
Inductive step (on depth). A comparator C at depth d has input lines that are of depth strictly less that d. If these input lines carry a_i and a_j when the input is a, they will carry, by the induction hypothesis $f(a_i)$ and $f(a_j)$, when the input is $f(a)$. This completes the induction, as C would produce, $f(\min(a_i, a_j))$, $f(\max(a_i, a_j))$ by the base case. □

Lemma 3.2 (0-1 Sorting Lemma)**.** *If a comparison network with n inputs sorts all 2^n possible input sequences of zeroes and ones, then it sorts all sequences of arbitrary numbers correctly.*

Proof. **0-1SL.** Suppose that the comparison network sorts all 2^n binary n inputs, but fails to sort a sequence s of arbitrary numbers. Then this means that in sequence s there are two numbers s_i and s_j that are out of order, i.e. $s_i < s_j$ but the network places s_j before s_i. We then define the following monotonically increasing sequence.

$$f(x) = \begin{cases} 0 & \text{if } x \leq s_i \\ 1 & \text{if } x \geq s_i \end{cases}$$

Since the network places s_j before s_i then given a monotonically increasing f it would place $f(s_j) = 1$ before $f(s_i) = 0$ in the output sequence, i.e. it would place an 1 before a 0. Then this specific input sequence of 0's and 1's (implied by f) would not be sorted contradicting the assumption that all binary sequences are sorted correctly. □

3.22 Arbitrary input sorting networks

We will propose the construction of two sorting networks both proposed by K. Batcher. One is known as the bitonic-based sorting network and the other is known as the odd-even merge sort-based network.

We first discuss the bitonic-based sorting network implying a bitonic-based sorting algorithm.

In order to build a sorting network, we will proceed in steps.

Step 1. We first show how to sort some structured sequences called bitonic sequences. We call the network that achieves that a bitonic sorting network. The network for sorting bitonic sequences of length n would be denoted by FC(n).

Step 2. We then show how to merge two sorted sequences and thus how to construct a merging sorting network using Step 2. The network for merging two sorted sequences is a variation of FC(n), and is denoted by $FC'(n)$.

Step 3. We finally show how to sort an arbitrary sequence by using a merging-based approach: to sort a sequence of length n, we recursively sort the two halves i.e. sequences of length $n/2$ and then we merge them using $FC'(n)$. We call the latter network $BS(n)$.

Because in Step 3 the algorithm uses Step 1, it is referred to as bitonic-based sorting or just bitonic sorting.

3.22.1 Bitonic sequence

A **bitonic sequence** is a sequence that

S1 monotonically increases and then monotonically decreases or

S2 is a circular shift (rotation) of a sequence described in case S1.

Example 3.6. *For example* $\langle 4,7,8,6,5,3,2,1\rangle$ *is a bitonic sequence ; this is because* $4 \nearrow 7 \nearrow 8 \searrow 6 \searrow \ldots \searrow 1$.

Example 3.7. *The sequence* $\langle 7,8,6,5,3,2,1,4\rangle$ *is also a bitonic sequence. It results from shifting/rotating the sequence* $\langle 4,7,8,6,5,3,2,1\rangle$ *one position to the left.*

In the remainder we will assume that n is a power of two. If this is not the case, we can pad keys in the sequence so that its length becomes a power of two.

3.22.2 Properties of bitonic sequences

Theorem 3.18 (Bitonic sequences)**.** *The number of 0-1 bitonic sequences is* $n^2 - n + 2$.

Proof. 0-1 bitonic sequences of length n are of the form $0^i1^j0^k$ where $1 < i, j < n$ or $1^i0^j1^k$ or 0^n or 1^n. The last two are special cases of the former two with $j = k = 0$ and $i = n$.

Consider the former sequence $0^i1^j0^k$.

```
0   1   2   3   4   5                     n-3          n
  x   x   x   x   x   ....    x   x   x   ....  x   x   x
```

We can place two separators A, B on any two of the $x's$ above: first A is placed anywhere and then B on the remaining x's between A and the rightmost x. Everything on the left of A will be 0 and associated with i, between A, B will be 1's and associated with j and on the right of B will be a 0's as well associated with k. How many choices do we have for A? Given that $i > 1$, we have $n-1$ choices for A/i from 1 to $n-1$, and $n-i$ choices for B/j from $i+1$ through n i.e. a total of $n-(i+1)+1 = n-i$. This also fixes the choices for k which will be n minus the value of j.

Thus the number of $0^i1^j0^k$ sequences is $\sum_{i=0}^{n-1}(n-i) = n(n-1)/2$. Similarly the number of $1^i0^j1^k$ sequences is $\sum_{i=0}^{n-1}(n-i) = n(n-1)/2$.

If we add the 0^n and 1^n we have $n(n-1)/2 + n(n-1)/2 + 2 = n^2 - n + 2$. □

Theorem 3.19 (0-1 Principle for bitonic sequences)**.** *If a comparator network sorts all 0-1 bitonic sequences, then it sorts arbitrary bitonic sequences.*

Proof. The proof is identical to the 0-1 Sorting Lemma.

a. We assume that a network sorts all 0-1 bitonic sequences but does not sort a generic bitonic sequence A. For this to happen an input a_j appears "on top" of an input a_i even if it should have been the other way around, i.e. it appears that even $a_i < a_j$ the output indicates that $a_j < a_i$, i.e. it is incorrect.

b. We use a monotonically increasing function $f(.)$ to convert the generic bitonic sequence A into a bitonic sequence of 0-1s. Let that be $f(A)$. By the proof of the 0-1 Monotonic Lemma, we have that the output instead of a_i, a_j is $f(a_i)$ and $f(a_j)$, i.e. it is an output of 0-1s. Given that f is constructed so that $f(a_i)$ is 0 and $f(a_j)$ is an 1, the appearance of $f(a_j)$ on top of $f(a_i)$ indicates that $f(a_j) < f(a_i)$, a contradiction.

The function f as noted above is the same as used in the 0-1 Sorting Lemma. The only thing that needs to be proved is that f converts a generic bitonic sequence A into a bitonic sequence $f(A)$ of 0-1s. If the latter $f(A)$ input sequence was not bitonic, no contradiction could be be proven as we don't know the behavior of the bitonic sorting network on non-bitonic input sequences generic or otherwise. Thus the only thing we need to prove is that if $A = \langle a_1, \ldots, a_n \rangle$ is a bitonic sequence then the sequence $f(A) = \langle f(a_1), \ldots, f(a_n) \rangle$ is also a 0-1 bitonic sequence, where $f(.)$ is a monotonically increasing function defined by

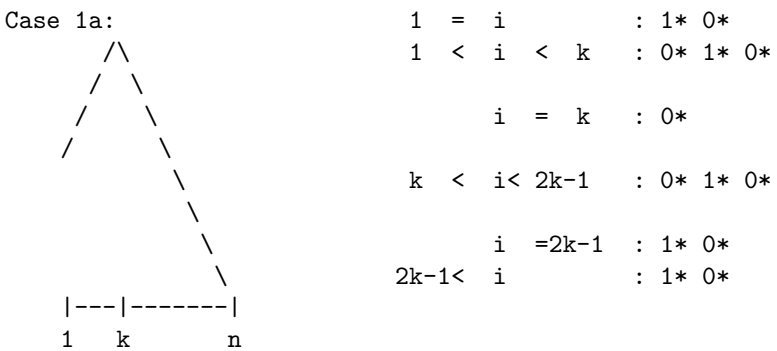
$$f(x) = \begin{cases} 0 & x \leq a_i \\ 1 & x > a_i \end{cases}$$

Since $A = \langle a_1, \ldots, a_n \rangle$ is bitonic, then A is $\nearrow\searrow$ or a rotation of it. We prove below that $f(A)$ is also bitonic.

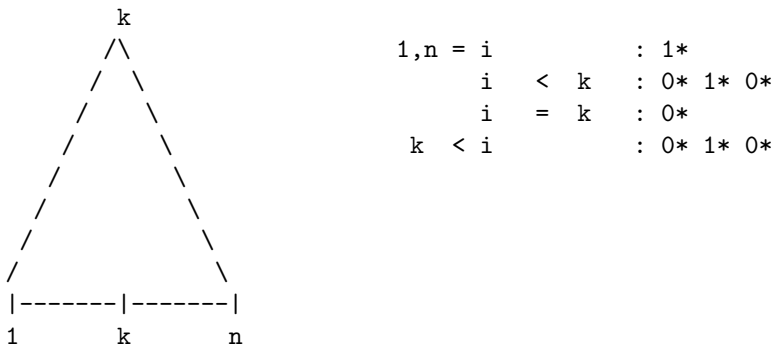
If A is $\nearrow\searrow$ we show that $f(A)$ is also of that form. If A is a rotation, then let the rotated sequence which is of the form $\nearrow\searrow$ be $R = \langle a_t, \ldots, a_n, a_1, \ldots a_{t-1} \rangle = s(A)$. We then prove that $f(R)$ is also of the $\nearrow\searrow$ form and thus A is a bitonic sequence.

Case 1: A is an $\nearrow\searrow$ sequence. For this case we distinguish three subcases as shown below. In all three cases the corresponding subcases yield a bitonic 0-1 sequence for $f(A)$, as shown.

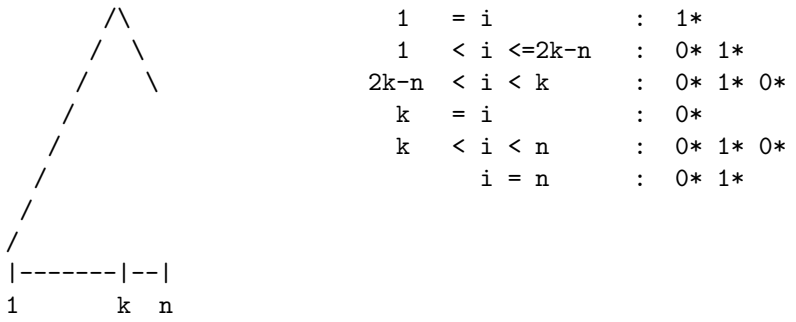
Case 1a: A is $\nearrow\searrow$ with a left-point higher than the right-point.



Case 1b: A is $\nearrow\searrow$ with a left-point and right-point at the same level/value.



Case 1c: A is $\nearrow\searrow$ with a left-point lower than the right-point.



Case 2: A is not a $\nearrow\searrow$ sequence, but R is

If $R = s(A)$, a circular shift of A, is a $\nearrow\searrow$ sequence, then $f(R)$ is a 0-1 bitonic sequence from Case 1 above, and thus $A = s^{-1}(R)$ is also a bitonic sequence of 0-1s. □

3.22.3 Bitonic sequence sorting using FC(n)

We first show how to sort a bitonic sequence of length n.
Input. A bitonic sequence of length n

Output. The bitonic sequence sorted (in non-decreasing order).

In the remainder we will assume that n is a power of two. If this is not the case, we can pad keys in the sequence so that its length becomes a power of two.

Idea 1: Half-cleaner HC(n). We construct a comparison network where line i is connected to line $i+n/2$, $0 \leq i < n/2$. Such a network turns a bitonic sequence of 0's and 1's into two bitonic sequences of half the original size, one of which is clean (all output lines contain one kind of input either all 0 or all 1). We call such a network a half-cleaner and we denote it with HC(n). (Proof of correctness is deferred to 6-5-5.)

Idea 2: Full-cleaner: bitonic sequence sorting done. We use the idea of half-cleaning recursively until we "clean" all the lines/wires. The first application of half-cleaning cleans half the input (which becomes a

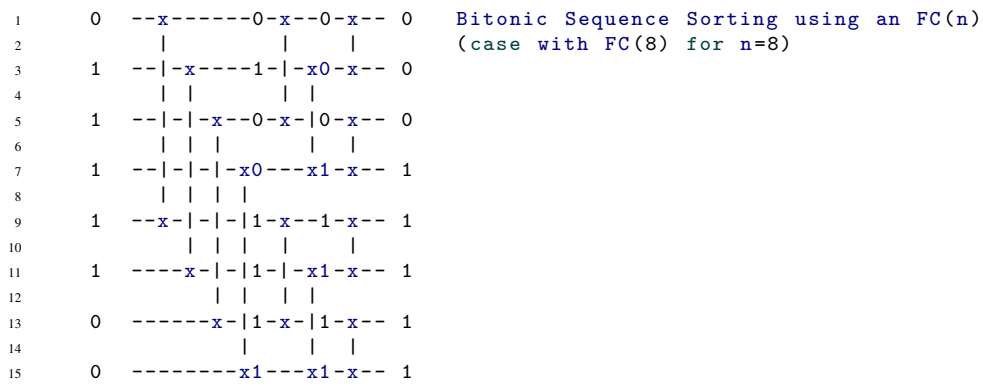


Figure 3.24: Bitonic sequence sorter FC(n)

trivial bitonic sequence) and turns the other half into a bitonic sequence whose elements are smaller than (or equal to) the cleaned ones if it is the top-half and at least the cleaned elements if it is the bottom half.

Bitonic sequence sorting: Full-cleaner. A full cleaner results by using half cleaners for smaller and smaller line size networks. We start with HC(n) a half-cleaner for n wires. It is then followed by two HC(n/2), followed by four HC(n/4) and so on. This becomes a FC(n). An HC(n) uses $n/2$ comparators and its depth is one. Thus

$$D_{HC}(n) = 1, \qquad S_{HC}(n) = n/2.$$

Bitonic sequence sorter FC(n). An FC(n) uses $\lg n$ levels of HC(.) Each level consists of $n/2$ comparators. The depth and size of the network are respectively

$$D_{FC}(n) = \lg n \qquad S_{FC}(n) = n \lg n/2.$$

Bitonic sequence sorting: Algorithm Correctness

In order to prove the correctness of this scheme we distinguish eight cases for a bitonic sequence of zeroes and ones. The first four of those cases are of the form $\bar{0}\bar{1}\bar{0}$, where $\bar{1}, \bar{0}$ denote a block of 1's and 0's of arbitrary (potentially not equal) size respectively. The remaining symmetric cases are for $\bar{1}\bar{0}\bar{1}$ and treated analogously.

(a) $\bar{1}$ does not include the midpoint and falls in top half.

(b) $\bar{1}$ does not include the midpoint and falls in bottom half.

(c) $\bar{1}$ includes midpoint and length less than $n/2$.

(d) $\bar{1}$ includes midpoint and length not less than $n/2$.

In case (a) a block $\bar{0}$ of length at least $n/2$ results in the top half (i.e it is clean). Same for case (b). In case (c) we have the same. In case d we have a block $\bar{1}$ of length $n/2$ in the bottom/upper half, and a bitonic sequence in the bottom half.

In all cases we obtain at least one clean half, two bitonic sequences, and every element in the top half is less than or equal to every element in the bottom half.

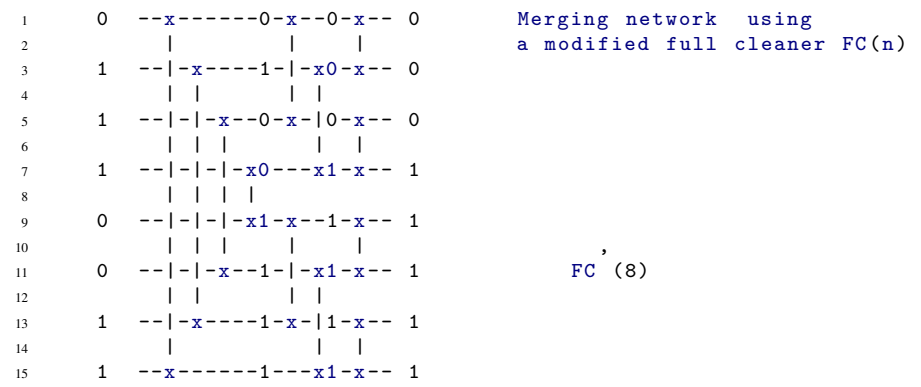


Figure 3.25: Bitonic sequence sorter $FC'(n)$

3.23 Merging

The problem of merging deals with merging two sorted sequences into one sorted sequence. The problem of merging is defined as follows.

MERGING(X[1..k],Y[1..l],Z[1..n=k+l],k,l,n=k+l) of two sorted sequences.
Input. Two sorted sequences $X[1..k]$ and $Y[1..l]$
Output. Merge X and Y into a sorted sequence $Z[1..n]$ keys, where $n = k+l$.

In the remainder we will assume that $k = l = n/2$.

The input $\langle X,Y\rangle$ consisting of two sorted sequences can become a bitonic sequence. For two sorted sequences X and Y we can form first sequence $\langle X,Y\rangle$. Afterwards we form $\langle X,rev(Y)\rangle$, where $rev(Y)$ is sequence Y reversed. The latter is a bitonic sequence.

Merging two sorted sequencs $\langle X,Y\rangle$ is equivalent to sorting the bitonic sequence $\langle X,rev(Y)\rangle$. The only problem that needs to be resolved is transforming $\langle X,Y\rangle$, two sorted sequences, into a bitonic sequence. This transformation can be incorporated in the first level of the bitonic sequence sorting network described in previous pages.

We recall that in the first phase of the bitonic sequence sorting involving FC(n) network line i is compared to $i+n/2$ as part of an HC(n) that is at the first level of FC(n).

Projecting X and Y as inputs to FC(n), the key of Y in line $i+n/2$ is the i-th largest key of Y. If Y was to be reversed then the key would move to line $n-i+1$.

Thus in order to build a merging-network out of FC(n) we need to make sure that the input is a bitonic sequence rather than two sorted sequences! This means that we need to have $HC'(n)$ at the first level where wire i and $n-i+1$ are fed into a comparator rather than i and $i+n/2$. (After the first phase is completed we get two bitonic sequences for the top and bottom half and thus the remaining phase of the bitonic sorter need not be modified.)

An $FC'(n)$ is an $FC(n)$ where the first level HC(n) has been replaced with an $HC'(n)$.

3.23.1 Sorting bitonically

A sorting network that is bitonic-based and utilizes $FC'(n)$ will be presented.

We are now ready to introduce the algorithm for sorting, i.e. build a sorting network that sorts its arbitrarily ordered input. The algorithm is merge-sort based. In order to sort n keys we produce two sorted

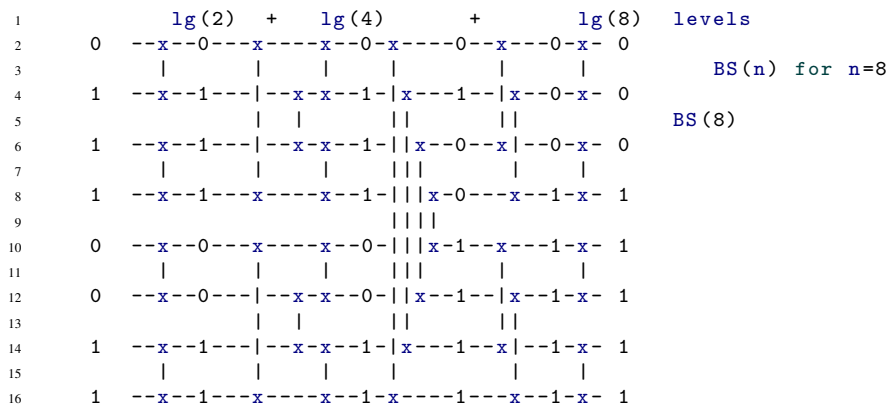


Figure 3.26: Bitonic-based sorter $BS(n)$

sequences recursively (divide and conquer step) and merge these sorted sequences , by the n-line merging network $FC'(n)$ of the previous section.

The base case of the recursive decomposition is easy. A sequence consisting of a single key is already sorted.

Therefore using this divide and conquer construction of a sorting network we build $BS(n)$ and can conclude the following.

A bitonic-based sorter $BS(n)$ consists of two $BS(n/2)$ that sort the first $n/2$ and the last $n/2$ keys respectively. At the output we have two sorted sequences. This structure is then followed by an $FC'(n)$ that merges the two $n/2$-long sorted sequences. Building block wise

$$BS(n) = BS(n/2) + BS(n/2) + FC'(n).$$

The depth of $BS(n)$ can be derived from the recursive construction above. $D_{BS}(n) = D_{BS}(n/2) + D_{FC'}(n)$. Therefore

$$D_{BS}(n) = D_{BS}(n/2) + \lg n \Rightarrow D_{BS}(n) = \lg n(\lg n + 1)/2.$$

The size of $BS(n)$ can be derived from the recursive construction above. $S_{BS}(n) = 2S_{BS}(n/2) + S_{FC'}(n)$. Therefore

$$S_{BS}(n) = 2S_{BS}(n/2) + (n/2)\lg n \Rightarrow S_{BS}(n) = n\lg n(\lg n + 1)/4.$$

3.23.2 Odd-even merge sort-based sorting network

We present the second sorting network that is known as the odd-even merge sort-based network. Building steps to constructing an odd-even merge sort-based sorting network are as follows.

Step 1. We first show how to merge two sorted sequences of length $n/2$ each, by using a method known as odd-even merge-sort. The network for merging two such sequences of total length n would be denoted by OM(n).

Step 2. We finally show how to sort an arbitrary sequence by using a merging-based approach: to sort a sequence of length n, we recursively sort the two halves i.e. sequences of length $n/2$ and then we merge them using $OM(n)$. We call the latter network $OS(n)$.

```
1       MergeSort(A,n)
2         MS(A,0,n-1);
3
4       MS(A, l, r)
5         if (l > 1)
6           Divide A[l..r] into two halves A[l..m] and A[m+1..r].
7           Sort A[l..m] and A[m+1..r] recursively.
8           Merge A[l..m] and A[m+1..r] into A[l..r]
```

Figure 3.27: MergeSort $MS(n)$

Theorem 3.20. *Properties of OS(n): $D_{OS}(n)$ and $S_{OS}(n)$. The new sorting network has $D_{OS}(n) = \lg n(\lg n - 1)/2$ and $S_{OS}(n) = n\lg n(\lg n - 1)/4$.*

The idea behind it is just plain merge-sort as well. In the remainder we assume that n is a power of two.

Question 3.4. *How many recursive rounds implied in Step 3 above?*

Answer: $O(\lg n)$.

Question 3.5. *How do you realize Step 2 above?*

Answer: OS(n) using a recursive construction.

Question 3.6. *How do you realize Step 3 above?*

Answer: OM(n) using a recursive construction and Odd-Even merging.

Question 3.7. *What are the properties of the sorting network?*

Answer: $D_{OM}(n) = \lg n + 1$. $D_{OS}(n) = \lg n(\lg n - 1)/2$. The depth recurrence is as follows.

$$D_{OS}(n) = D_{OS}(n/2) + D_{OM}(n)$$

where $D_{OS}(n)$ is the depth of the sorting network for sorting n arbitrary keys, $D_{OM}(n)$ is the depth of the odd-even merging network for merging two sorted sequence of $n/2$ keys each.

The only question left unaswered is: relates to odd-even merging that works miraculously well.

3.23.3 Odd-even merging (Batcher's original method)

OddEvenMergeB($A = \langle a_0 \dots a_{n/2-1} \rangle$, $B = \langle b_0 \dots b_{n/2-1} \rangle$, n)
Input. Two sorted sequences A and B as stated with a total of n keys, where n is a power of two. Each sequence contains $m = n/2$ keys.
Output. Merge A and B into a sorted sequence of n keys: the first half appears in A and the second half in B.

```
1  void  OddEvenMergeB(keys A[0..m-1] , keys B[0..m-1], int n ) {
2  /* This is the original Batcher method. Note m=n/2=> n=2*m  */
3    if $(n>2) {
4       Even(A)     = [A[0], A[2], ... , A[m-2]] ;
5       Odd(A)      = [A[1], A[3], ... , A[m-1]] ;
6       Even(B)     = [B[0], B[2], ... , B[m-2]] ;
7       Odd(B)      = [B[1], B[3], ... , B[m-1]] ;
8       C = OddEvenMergeB ( Even(A), Even(B), m/2) ;
9       D = OddEvenMergeB ( Odd (A), Odd (B), m/2) ;
10      M = [c[0], d[0], c[1], d[1], c[2], d[2],..., c[m-1], d[m-1]] ;
11 /* Below x:y denotes comparator applied to inputs x,y *\
12      L = [c[0], d[0]:c[1], d[1]:c[2],..., d[m-2]:c[m-1],  d[m-1]] ;
13 /* Perform a comparison side-by-side between d[i] and c[i+1] */
14   }
15   else CompareSwap( a[0], b[0]) ;
16   Return(L) ;
17 }
```

Figure 3.28: Odd Even merge

begin ODDEVENMERGEB ($A = \langle a_0 \ldots a_{m-1} \rangle$,$B = \langle b_0 \ldots b_{m-1} \rangle$,2m)
// This is the original Batcher method. Note $m = n/2$
0. if $(n > 2)$
1a. Even(A) $= \langle a_0, a_2, \ldots, a_{m-2} \rangle$;
1b. Odd(A) $= \langle a_1, a_3, \ldots, a_{m-1} \rangle$;
1c. Even(B) $= \langle b_0, b_2, \ldots, b_{m-2} \rangle$;
1d. Odd(B) $= \langle b_1, b_3, \ldots, b_{m-1} \rangle$;
2a. C= ODDEVENMERGEB(Even(A),Even(B),m/2);
2b. D= ODDEVENMERGEB(Odd (A),Odd (B),m/2);
3. $M = \langle c_0, d_0, c_1, d_1, c_2, d_2, \ldots, c_{m-1}, d_{m-1} \rangle$;
// Below x:y denotes comparator : and two input x,y
4. L $= \langle c_0, d_0 : c_1, d_1 : c_2, \ldots, d_{m-2} : c_{m-1}, d_{m-1} \rangle$;
//Perform a comparison side-by-side between d_i and c_{i+1}.
5. else CompareSwap(a_0, b_0);
6. Return(L);
end ODDEVENMERGEB

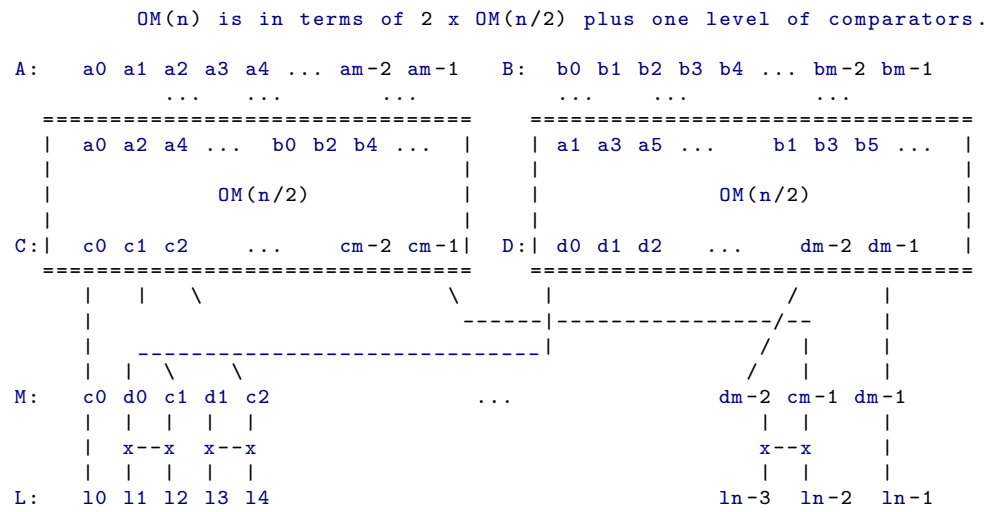


Figure 3.29: Batcher's odd-even merging network

3.23.4 Example

```
An Example of the Batcher's method
    A= < 1  5  7  8 >
    B= < 2  3  4  6 >
   even(A) = < 1 7 >   odd(A) = < 5  8 >
   even(B) = < 2 4 >   odd(B) = < 3  6 >
   C    = OddEvenMerge(even(A), even(B)) = < 1 2 4 7>
   D    = OddEvenMerge( odd(A),  odd(B)) = < 3 5 6 8>
   M    = < 1 3:2 5:4 6:7 8>
   L    = < 1 2 3 4 5 6 7 8>
```

3.23.5 Batcher's odd-even merging network: a recursive structuring

Lemma 3.3. *Algorithm* OddEvenMergeB *works as claimed. Consequently network OM(n) works as claimed as well.*

Proof. We use the 0-1 Sorting Lemma.
Base case $m = 2$**.** OM(2) consists of one comparator. OM(4) is true by inspection.
Inductive step. Let us suppose that OM(n/2) and OM(n/2) merge correctly. We are going to use the 0-1 Merging Lemma which is the merging analogue of the 0-1 Sorting Lemma. Let A consists of a zeroes and B of b zeroes. Thus the number of ones is $m-a$ and $m-b$ respectively. Then C contains $c = \lceil a/2 \rceil + \lceil b/2 \rceil$ zeroes followed by $m - \lceil a/2 \rceil - \lceil b/2 \rceil = m - c$ ones. Likewise D contains $d = \lfloor a/2 \rfloor + \lfloor b/2 \rfloor$ zeroes followed by $m - \lfloor a/2 \rfloor - \lfloor b/2 \rfloor = m - d$ ones. Note that c can be two more than d and no more.

For sequence M subsequently formed there are three cases to consider.
Case 1. C has the same number of zeroes as D and $a = b = 0$. The output is trivially sorted.
Case 2. C has the same number of zeroes as D and $a, b > 1$. Then M looks like and it is sorted and thus L would remain sorted.
Case 3. C has one more zero than D. Then M looks like as follows. L would be sorted anyway.

c_0	d_0	c_1	d_1	c_2	d_2	c_3	d_3	$c4$
0	0	0	0	0	0	1	1	1

c_0	d_0	c_1	d_1	c_2	d_2	c_3	d_3	$c4$
0	0	0	0	0	1	1	1	1

Case 4. C has two more zeroes than D. Then M looks like. The comparison involving $d_1 : c_2$ would turn L sorted. This completes the proof. □

c_0	d_0	c_1	d_1	c_2	d_2	c_3	d_3	$c4$
0	0	0	1	0	1	1	1	1

Theorem 3.21 (*Depth*)**.** *The depth $D_{OM}(n) = D_{OM}(n/2)+1$ where the $+1$ is the extra level to generate M from L. Moreover $D_{OM}(2) = 1$. For n a power of two we obtain*

$$D_{OM}(n) = \lg n.$$

Theorem 3.22 (*Size*)**.** *The size $S_{OM}(n) = 2S_{OM}(n/2)+n/2-1$ where the number of comparators from M to L is $(n-2)/2 = n/2-1$. Moreover $S_{OM}(2) = 1$. For n a power of two we obtain*

$$S_{OM}(n) = (n/2)\lg(n/2)+1 = n\lg n/2 - n/2 + 1.$$

3.23.6 Yet another variant

A variant of odd-even merging of Batcher's original method is shown below.

OddEvenMergeV($A = \langle a_0 \ldots a_{n/2-1}\rangle$**,** $B = \langle b_0 \ldots b_{n/2-1}\rangle$**, n)**
Input. Two sorted sequences A and B as stated with a total of n keys, where n is a power of two. Each sequence contains $m = n/2$ keys.
Output. Merge A and B into a sorted sequence of n keys: the first half appears in A and the second half in B.

The variant given here slightly differs from Batcher's original method shown on the previous page. Last page Odd(A) with Odd(B) and Even(A) with Even(B) are merged. That however, requires the comparison of d_i with c_{i+1} in line 4 instead, with c_0 and d_{m-1} not participating in that step (left unchanged). In this variant the comparisons are natural c_i with d_i.

3.23.7 An example

```
An Example of Batcher's variant.
    A= < 1  5  7  8 >
    B= < 2  3  4  6 >
   even(A) = < 1 7 >   odd(A) = < 5  8 >
   even(B) = < 2 4 >   odd(B) = < 3  6 >
   C    = OddEvenMerge(even(A), odd(B)) = < 1 3 6 7>
   D    = OddEvenMerge(odd (A),even(B)) = < 2 4 5 8>
   M    = < 1:2 3:4 6:5 7:8>
   L    = < 1 2 3 4 5 6 7 8>
```

Lemma 3.4. *Algorithm* ODDEVENMERGEV *works as claimed. Consequently network $OM'(n)$ works as claimed as well.*

```
void  OddEvenMergeV(keys A[0..m-1] , keys B[0..m-1], int n ) {
/* This is a variant of Batcher method. Note m=n/2=> n=2*m  */
  if $(n>2) {
     Even(A)    = [A[0], A[2], ... , A[m-2]] ;
     Odd(A)     = [A[1], A[3], ... , A[m-1]] ;
     Even(B)    = [B[0], B[2], ... , B[m-2]] ;
     Odd(B)     = [B[1], B[3], ... , B[m-1]] ;
     C = OddEvenMergeV ( Even(A),  Odd(B), m/2) ;
     D = OddEvenMergeV ( Odd (A), Even(B), m/2) ;
     M = [c[0], d[0], c[1], d[1], c[2], d[2],..., c[m-1], d[m-1]] ;
/* Below x:y denotes comparator applied to inputs x,y *\
     L = [c[0]:d[0], c[1]:d[1], c[2]:d[2],...,    c[m-1] :d[m-1]] ;
/* Perform a comparison side-by-side between d[i] and c[i+1] */
  }
  else CompareSwap( a[0], b[0]) ;
  Return(L) ;
}
```

Figure 3.30: Odd Even merge variant of Batcher's method

begin ODDEVENMERGEV ($A = \langle a_0 \ldots a_{m-1}\rangle$, $B = \langle b_0 \ldots b_{m-1}\rangle$,2m)
// This is a variant of Batcher's method. Note $m = n/2$
0. if $n > 2$
1a. Even(A) $= \langle a_0, a_2, \ldots, a_{m-2}\rangle$;
1b. Odd(A) $= \langle a_1, a_3, \ldots, a_{m-1}\rangle$;
1c. Even(B) $= \langle b_0, b_2, \ldots, b_{m-2}\rangle$;
1d. Odd(B) $= \langle b_1, b_3, \ldots, b_{m-1}\rangle$;
2a. C= ODDEVENMERGEV(Even(A),Odd (B),m/2);
2b. D= ODDEVENMERGEV(Odd (A),Even(B),m/2);
3. $M = \langle c_0, d_0, c_1, d_1, c_2, d_2, \ldots, c_{m-1}, d_{m-1}\rangle$;
// Below x:y denotes comparator : and two input x,y
4. L $= \langle c_0 : d_0, c_1 : d_1, c_2 : d_2, \ldots, c_{m-1} : d_{m-1}\rangle$;
5. else CompareSwap(a_0, b_0);
6. Return(L);
end ODDEVENMERGEV

Proof. We use the 0-1 Sorting Lemma.
Base case $m = 2$. $OM'(2)$ consists of one comparator. $OM'(4)$ is true by inspection. **Inductive step.** Let us suppose that $OM'(n/2)$ and $OM'(n/2)$ merge correctly. We are going to use the 0-1 Merging Lemma which is the merging analogue of the 0-1 Sorting Lemma. Let A consists of a zeroes and B of b zeroes. Thus the number of ones is $m-a$ and $m-b$ respectively. Then C contains $c = \lceil a/2\rceil + \lfloor b/2\rfloor$ zeroes followed by $m - \lceil a/2\rceil - \lfloor b/2\rfloor = m - c$ ones. Likewise D contains $d = \lfloor a/2\rfloor + \lceil b/2\rceil$ zeroes followed by $m - \lfloor a/2\rfloor - \lceil b/2\rceil = m - d$ ones. Note that now c and d can be equal or one is off by one from the other in other words $\mid c - d \mid \leq 1$.

For sequence M subsequently formed there are four cases to consider.
Case 1. C has the same number of zeroes as D and $a = b = 0$. The output is trivially sorted.
Case 2. C has the same number of zeroes as D and $a, b > 1$. Then M looks like and it is sorted and thus L would remain sorted.
Case 3. C has one more zero than D. Then M looks like as follows. L would be sorted anyway. **Case 4.** C

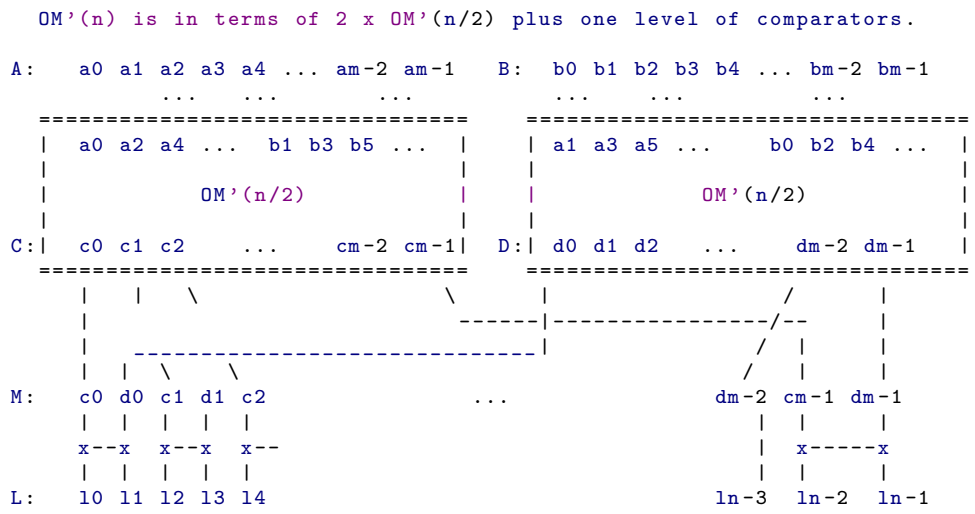


Figure 3.31: Batcher's odd-even merging variant network

c_0	d_0	c_1	d_1	c_2	d_2	c_3	d_3	$c4$
0	0	0	0	0	0	1	1	1

has one fewer zero than D. Then M looks like. The comparison involving $c_i : d_i$ (in the example $c_2 : d_2$ would turn L sorted. □

Theorem 3.23 (*Depth*)**.** *The depth $D_{OM'}(n) = D_{OM'}(n/2) + 1$ where the $+1$ is the extra level to generate M from L. Moreover $D_{OM'}(2) = 1$. For n a power of two we obtain*

$$D_{OM'}(n) = \lg n.$$

Theorem 3.24 (*Size*)**.** *The size $S_{OM'}(n) = 2S_{OM'}(n/2) + n/2$ where the number of comparators from M to L is $n/2$. Moreover $S_{OM'}(2) = 1$. For n a power of two we obtain*

$$S_{OM'}(n) = n \lg n/2$$

3.23.8 From merging to sorting

The interesting point about OddEvenMergeB or OddEvenMergeV is that they are the preparatory steps of a sorting algorithm or network OS(n). The algorithm implies by OS(n) is called OddEvenMergeSort and is described below for completion. It sorts a sequence of n keys.

Theorem 3.25 (*Sorting using Batcher's odd even merge*)**.** *We analyze Batcher's original version with respect to depth. The same analysis applies to the variant. Note that $D_{OS}(2) = 1$ or equivalently $D_{OS}(1) = 0$. For other values of $n > 1$ we have*

$$\begin{aligned} D_{OS}(n) &= D_{OS}(n/2) + D_{OM}(n/2) \\ D_{OS}(n) &= D_{OS}(n/2) + \lg n \\ D_{OS}(n) &= \lg n \left(\lg n + 1\right)/2 \end{aligned}$$

c_0	d_0	c_1	d_1	c_2	d_2	c_3	d_3	$c4$
0	0	0	0	0	1	1	1	1

c_0	d_0	c_1	d_1	c_2	d_2	c_3	d_3	$c4$
0	0	0	0	1	0	1	1	1

Size-wise we have $S_{OM}(n/2) = n\lg n/2 - n/2 + 1$ *and* $S_{OM}(1) = 0$ *for Batcher's odd-even sorter. Generating the recurrence we have.*

$$\begin{aligned} S_{OS}(n) &= 2S_{OS}(n/2) + S_{OM}(n/2) \\ S_{OS}(n) &= 2\frac{n\lg n\,(\lg n - 1)}{4} + n - 1. \end{aligned}$$

we obtain the indicated solution. The running time is the size of the network $OS(n)$.

The analysis of the size of the variant $OS'(n)$ and is left as an exercise to show $S_{OS'}(n) = n\lg n(\lg n + 1)/4$.

3.23.9 Conclusion

Theorem 3.26 (*Odd-even merge sorting*)**.** *Odd-even merge-sort works as claimed.*

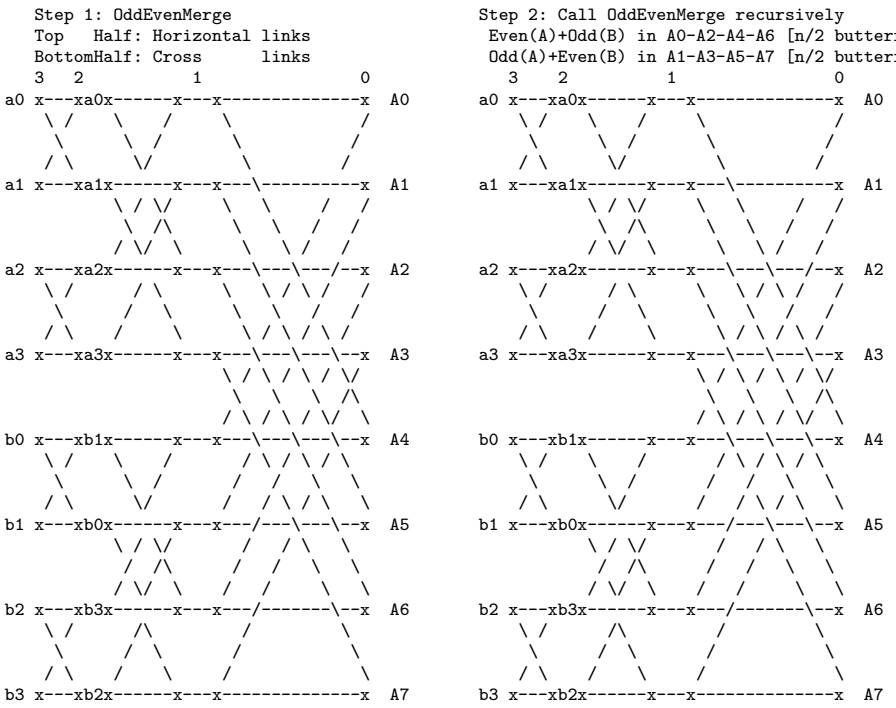
Proof. A proof that OE merge-sort works utilizes the 0-1 sorting lemma. By induction let us assume that OE merge-sort works for sizes less than or equal to $n-1$.

Therefore in order to sort n keys, we split them into 2 halves of size $n/2$ each. By the inductive hypothesis, o-e merge-sort sorts independently the two halves. It remains to be shown that the merging algorithm so described merges the two sorted sequences and the theorem is proved. The latter has been proven earlier. □

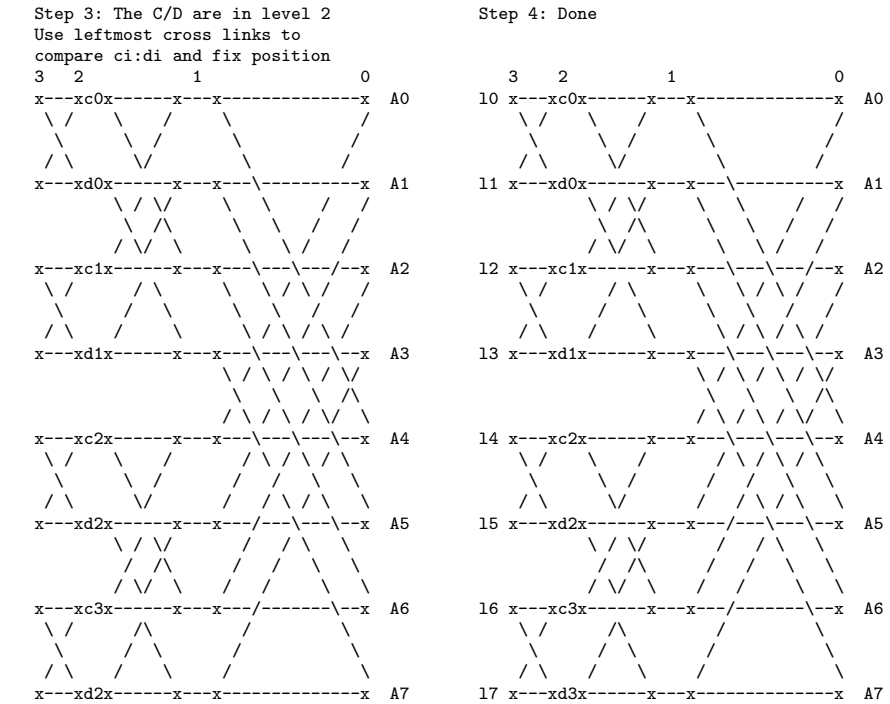
3.23.10 AKS sorting network

There are asymptotically faster sorting networks today (eg. the AKS network due to Ajtai-Komlos-Szemeredi, 1984) that have depth $O(\lg n)$; however they suffer from large constants hidden in the big-Oh notation and they are not practical for small values of n.

```
begin OddEvenMergeSort (X = ⟨x_0 ... x_{n-1}⟩,n)
1a.     Left(A)  = ⟨x_0, x_1, ..., x_{n/2-1}⟩;
1b.     Right(A)= ⟨x_{n/2}, x_{n/2+1}, ..., x_{n-1}⟩;
2a.     L(A) = OddEvenMergeSort(Left (A),n/2);
2b.     R(A) = OddEvenMergeSort(Right(A),n/2);
3.      Y= OddEvenMerge(L(A),R(A),n/2);
5.      Return(Y);
end OddEvenMergeSort
```



Step 1: Level 3 to Level 2 for 8-butterfly: Prepare odd-even sequences.
Step 2: Level 2 to Level 0 back to Level 2 : Recursive merging.



3.24 Realistic Parallel Abstraction: BSP model

Reading. L.G.Valiant. A bridging model for parallel computation. Communications of the ACM, 33(8):103-111, August 1990, accessible through the Course Web-page.

Recommended Reading. Culler, Karp, Patterson et al. LogP: Towards a Realistic model of parallel computation. Proceedings of the 4th ACM SIGPLAN Symposium on Principles and practice of parallel programming, San Diego, CA, USA. Pages 1-12. 1993.

3.25 The bulk-synchronous parallel model (BSP Model)

Since the introduction and adaptation of the von-Neumann model for sequential computing, the effects of computer revolution on society have been pretty significant. A general purpose computer performs well on computer programs written on a variety of standardized programming languages like C, Fortran, Cobol or Lisp and the same computer program can be easily ported on other platforms.

It has always been realized that parallel computers will eventually supersede sequential machines. This has yet to happen despite advances in computer technology and the fact that chip technology seems to have reached physical limitations; nowadays, fast machines are not much faster than the slowest ones which are as fast (or perhaps faster) as a supercomputer of twenty years ago. Small incremental improvements that may lead to stagnation of sequential computing seem inevitable. Despite these shortcomings of sequential computing, there has been no significant spread of use of parallel computers and few companies have realized that their future may rely on parallel platforms. The main reason for this has been that parallel computer platforms are built in such a way that are too hardware specific, programs written for them exhibit poor performance unless the programmer fine-tunes its code to take into consideration features of the particular architecture. Not only the code is non-portable but scalability comes at a high cost as well. On the other hand parallel algorithms designed and analyzed by the theorists work on parallel models that usually ignore communication and/or synchronization issues, like the PRAM and its variants, and work only under unlimited parallelism assumptions.

One of the earliest attempts to model a parallel computer has been the Parallel Random Access Machine (PRAM) which is one of the most widely studied abstract parallel models. A PRAM consists of a collection of processors which work synchronously and which communicate with a global shared random access memory which can access in unit time. There are many different types of PRAMs which are distinguished from the way they access the shared memory (eg CRCW, EREW PRAMs). Numerous parallel algorithms have been developed for this parallel computer model.

More realistic models of parallel computing view a parallel computer as a collection of sequential processors, each one having its own local memory (*distributed-memory model*). The processors are interconnected by a network which allows them to communicate by sending and receiving messages. Constraints such as the maximum number of pins on a chip, or the maximum width of a data bus, limit the capacity of a processor to communicate with any other processor. It is only possible for a single processor to communicate directly with few others, in most cases those physically close to it. If a message needs to be sent to a distant processor it is relayed through a number of intermediate processors.

As it has already been mentioned, the parallel machines built in the 1980s and early 90s failed to garner general acceptance mainly because of the lack of a stable, unified and bridging parallel programming model. These deficiencies made programming of such machines difficult (cf. assembly vs higher level programming languages), time consuming, non-portable and architecture-specific. Recently, the introduction of realistic parallel computer models such as the Bulk-Synchronous Parallel (BSP) model of computation by L.G. Valiant comes to address these limitations of parallel computing. Our hope is that further architectural convergences will occur with the goal of writing software that will be portable and run with high performance on a variety

of architectures from networks/clusters of workstations (NOW/COW) to parallel supercomputers.

The *Bulk-Synchronous Parallel* (BSP) model of computation has been proposed by L.G. Valiant, as a **unified** framework for the design, analysis and programming of general purpose parallel computing systems. It allows the design of algorithms that are both **scalable and portable**.

In a BSP program, processors jointly advance through its various phases called **supersteps** with the required computations and remote communication occurring between them; at the end of a superstep processors check themselves in order to proceed to the following superstep.

The BSP model consists of three parts:

(1) a collection of *processor-memory components*,

(2) a *communication network* that can deliver messages point-to-point among the components, and

(3) a *facility for global synchronization*, in barrier style, of all or a subset of the components.

A time step (as opposed to a CPU instruction or cycle) would refer to the time needed to perform a local computation (such as a fetch from memory and a floating-point operation followed by a store operation).

It should be noted that, although the model stresses global barrier-style synchronization, pairs of processing units may synchronize pairwise by sending messages to and from an agreed memory location. However, such message exchanges should respect the superstep rules.

As mentioned, computation on the BSP model proceeds in a succession of *supersteps*. A superstep may be thought of as a segment of computation during which each processor performs a given task using data already available there locally before the start of the superstep. Such a task may include (i) local computations, (ii) message transmissions, and (iii) message receipts.

3.25.1 BSP Model: Parameter p, L, g

The tuple (p, L, g) characterizes the behavior and performance of a BSP computer.

- p is the number of components available.
- L is the minimum time between successive synchronization operations,
- g is the ratio of the total throughput of the whole system (in a steady state, i.e. in a state of continuous message usage) in terms of basic computational operations, to the throughput of the communication network in terms of words of information delivered.

- A lower bound on the value of L is the time for a remote memory operation/message dispatch to become effective and is thus dependent on say, the diameter of the interconnection network.
- The time for barrier synchronization also poses a lower bound on the effective value of L.
- An upper bound on L is application specific and expressed in terms of problem size n as well.
- The value of g is measured while the network is in a steady state, i.e. latency issues become insignificant in the measurement of communication time; parameter L is large enough for the theoretical bound on g to be realizable.

The theoretical definition of g relates to the routing of h-relations; when each processor sends or receives at most h messages (of the same size) an h-relation is realized and the cost assigned to this communication is gh provided that $h \geq h_0$, where h_0 is a machine dependent parameter.

Otherwise the cost of communication is L.

This way latency issues associated with small message size/messages are taken into consideration.

Machine	Mflop ($\geq$)	p	L ($\geq$) in flops/μsec	g in flops/word/μsec/word
SGI Power Challenge	80	4	2000/26	9.3/0.13
MultiProc Sun	10	4	120/12	4.1/0.41
IBM SP2	30	4	3600/140	8.0/0.30
Digital Alpha Farm	10	4	47000/4664	81/8.10
PCs in Cluster	200	4	180000/900	65/0.325

Figure 3.32: Estimated BSP parameters for a variety of platforms

In practice, and in various libraries implementing the BSP model message requests issued/accepted by processors are of variable size. The h of an h-relation relates then to the total size of communicated data and g is expressed in terms of basic computational operations (sometimes, floating-point operations) or absolute time (seconds) per data-unit (a byte or word of information).

In practice, the parameter L of the BSP model not only hides the cost of communication when each processor sends or receives a small number of messages but also the cost of communication where each processor may send or receive a large number of messages but each one is of small size.

For any BSP computer, the values of L and g are likely to be a non-decreasing functions of p.

The use of L and g to characterize the communication and synchronization performance of a BSP computer is important because such issues are abstracted in only two parameters thus allowing considerations to be moved from a local level to a global one.

For the sake of an example, the values for L and g for some abstract network topologies are as follows. A ring has $L = O(p), g = O(p)$, a 2d-array (mesh) $L = O(\sqrt{p}), g = O(\sqrt{p})$, a butterfly $L = O(\log p), g = O(\log p)$, and a hypercube $L = O(\log p), g = O(1)$. For the case of a hypercube, as $g = O(1)$ the cost of routing a permutation, i.e. an one-relation is not $1 \cdot g$ but L. Thus h_0 for the hypercube is such that $h_0 = \Theta(\log p)$.

If in a superstep

(1) an h-relation is realized, and

(2) x computational operations are performed,

then, the cost of the superstep is given by the following formula.

(1) $\max\{L, x + gh\}$.

Alternative costs are $\max\{L, x, gh\}$ and $L + x + gh$.

The maximum size of a superstep depends on the problem in hand. Under the BSP programming paradigm the objective is to maximize the size of supersteps, decrease their number and increase processor utilization.

The description of BSP programs can be simplified by separating computation and communication and assuming that each superstep contains either local computations or communication.

3.26 Optimality of Algorithms under the BSP model

Two modes of programming on the BSP model were envisaged: **automatic mode** where programs are written, say PRAM style, in a high level language that hides memory distribution from the user and **direct mode** where the programmer retains control of memory allocation. In the **direct mode** of programming small

multiplicative constant factors in runtime are important. It can be shown that the automatic mode achieves optimality within constant factors by simulating say PRAM algorithms on the BSP.

The term **slack** in the context of algorithm design refers to the ratio of the problem size over the processor number of the BSP machine. The question is whether the direct mode can be beneficial in circumstances where:

- small multiplicative constant factors in runtime are important,
- where smaller problem instances can be run more efficiently in direct mode (less slack is required) than in automatic mode,
- where the available machine has high g (automatic mode requires g to be constant), and
- L is high for direct but not for automatic mode for the problem instance in hand.

Although it is difficult to measure performance to high accuracy, since the operations that are counted have to be carefully defined, we can make such measures meaningful by measuring ratios between runtimes on pairs of models that have the same set of local instructions.

The performance of a BSP algorithm P is thus described in three parts.

- A sequential algorithm S with which we are comparing P is first specified.
- The model of computation used for both algorithms is then defined and the basic computational operations that will be counted in both P and S are also described and the charging policy is made explicit.
- Second, two ratios π and μ are specified.
    - π, is the ratio between the computation time C_P, of the BSP algorithm, over the time C_S of the comparing sequential algorithm divided by p, i.e., $\pi = pC_P/C_S$.
    - μ, is the ratio between the communication time M_P required by the communication supersteps of the BSP algorithm and the computation time of S divided by p, i.e., $\mu = pM_P/C_S$.
    - When communication time is described, it is necessary that the amount of information that can be conveyed in a single message be made explicit.
    - Finally, conditions on n, p, L and g are specified that are sufficient for the algorithm to be plausible and the claimed bounds on π and μ to be valid.

Corollaries describe sufficient conditions for the most interesting optimality criteria, such as c-optimality, i.e., $\pi = c + o(1)$ and $\mu = o(1)$. All asymptotic bounds refer to the problem size as $n \to \infty$.

3.27 Software Support under the BSP model

The BSP model, unlike other models of parallel computation, is not just an architectural-oriented theoretical model; it can also serve as a paradigm for programming parallel computers.

(1) The fundamental concept introduced by the BSP model is the notion of the superstep, and that all remote memory accesses occur between supersteps as part of a global operation among the processors the results of these accesses become effective at the end of the current superstep.

Although it may have been apparent why some consider the BSP as a satisfactory unifying and bridging model for parallel computation, one may ask the question of how successful it has been as a practical model and what level of software support there is for BSP. The BSP model has been realized as a library of functions for process creation and destruction, remote memory access and message passing, and global, barrier-style, synchronization The abstraction offered by the BSP model is such that any library offering such facilities can be used for programming according to the BSP programming paradigm. The Oxford BSP Library that supports Direct Remote Memory Access (DRMA) for parallel programs, the Green BSP library that supports message passing and the Oxford BSP Toolset that supports both DRMA and message passing are some of the libraries that specifically allow programming according to the BSP paradigm. Just as the von-Neumann model encompasses various programming language paradigms eg. functional, logic programming, the BSP does not dictate a particular mode of programming as well. All three libraries present a particular set of choices to the user. Some of the elements of this set are:

- **Data Parallel Program Structure**. It allows large-scale parallelism by splitting data and concurrently working on the individual pieces.
- **SPMD programs.** A Single Program Multiple Data programming style is used as perhaps implied by the structure of supersteps.
- **Direct mode of global memory management.** The programmer has direct access to memory allocation and determines how data are partitioned.
- **Static processor allocation.** The number of participating processors is determined in the beginning of the execution and cannot vary during the computation dynamically.

3.28 Performance vs Running Time Prediction under the BSP model

Whereas performance of a particular BSP algorithm can be reliably predicted, one *should not* expect the BSP cost model to accurately predict the running time and behavior of a particular implementation. Accurate prediction is difficult even for sequential algorithms due to the existence of varying memory hierarchies on real machines; adding parallelism and the side-effects of communication introduces two more difficulties.

There have been various attempts to more accurately predict parallel performance by extending the BSP model. The E-BSP model is one such approach and is more explicit and specific about the communication network of a particular hardware platform, and the patterns of communication involved in routing. The attractiveness of the BSP cost model is its simplicity and generality; introducing more parameters to describe the performance of a communication network under various patterns of communication increases the complexity of describing the performance of all but the simplest algorithms with perhaps only small gains in prediction accuracy. Such an approach may also be problematic as it is the main reason parallel computing failed in the past: the attempt to realize for a particular algorithm those patterns of communication that are optimal for a given platform (and communication network), whereas they may lead in significant degradation in performance, if utilized in other platforms (portability vs. efficiency). Another variant of the BSP model introduces one more parameter, B, related to message size, and associates g with that message size, enforcing this way coarse-grained communication of messages of size equal to B. The original BSP model does not elaborate in detail between fine-grained and coarse-grained communication. If small h-relations are communicated (where "small" is to mean less than some parameter h_0, usually assumed to be equal to L/g) a cost L is assigned to such a communication; no mention of message size is inferred. Presumably, for BSP to be an abstract and general-purpose model, details of how communication is performed efficiently are left to the BSP library implementor. In practice, the generality of the BSP model works well if one interprets the cost

model so as to absorb in L not only "small" h-relations but also "small" messages; therefore, the value of B is reflected in the choice of L and g (as is, h_0 as well) without the need of introducing an extra parameter.

3.28.1 Traditional vs Architecture Independent Parallel Al gorithm Design

As an example of how traditional PRAM algorithm design differs from architecture independent parallel algorithm design, example algorithm for broadcasting in a parallel machine is introduced.

Problem: In a parallel machine with p processors numbered $0,\ldots,p-1$, one of them, say processor 0, holds a one-word message The problem of *broadcasting* involves the dissemination of this message to the local memory of the remaining $p-1$ processors.

The performance of a well-known exclusive PRAM algorithm for broadcasting is analyzed below in two ways under the assumption that no concurrent operations are allowed. One follows the traditional (PRAM) analysis that minimizes parallel running time. The other takes into consideration the issues of communication and synchronization as viewed under the BSP model. This leads to a modification of the PRAM-based algorithm to derive an architecture independent algorithm for broadcasting whose performance is consistent with observations of broadcasting operations on real parallel machines.

3.28.2 Broadcasting: PRAM-1

Algorithm. Without loss of generality let us assume that p is a power of two. The message is broadcast in $\lg p$ rounds of communication by binary replication. In round $i = 1,\ldots,\lg p$, each processor j with index $j < 2^{i-1}$ sends the message it currently holds to processor $j+2^{i-1}$ (on a shared memory system, this may mean copying information into a cell read by this processor). The number of processors with the message at the end of round i is thus 2^i.

Analysis of Algorithm. Under the PRAM model the algorithm requires $\lg p$ communication rounds and so many parallel steps to complete. This cost, however, ignores synchronization which is for free, as PRAM processors work synchronously. It also ignores communication, as in the PRAM the cost of accessing the shared memory is as small as the cost of accessing local registers of the PRAM.

Under the BSP cost model each communication round is assigned a cost of $\max\{L, g\cdot 1\}$ as each processor in each round sends or receives at most one message containing the one-word message. The BSP cost of the algorithm is $\lg p \cdot \max\{L, g\cdot 1\}$, as there are $\lg p$ rounds of communication. As the communicated information by any processors is small in size, it is likely that latency issues prevail in the transmission time (ie bandwidth based cost $g\cdot 1$ is insignificant compared to the latency/synchronization reflecting term L).

In high latency machines the dominant term would be $L\lg p$ rather than $g\ \lg p$. Even though each communication round would last for at least L time units, only a small fraction g of it is used for actual communication. The remainder is wasted. It makes then sense to increase communication round utilization so that each processor sends the one-word message to as many processors as it can accommodate within a round.

3.28.3 PRAM-2

Input: p processors numbered $0\ldots p-1$. Processor 0 holds a message of length equal to one word.

Output: The problem of *broadcasting* involves the dissemination of this message to the remaining $p-1$ processors.

Algorithm 2. In one superstep, processor 0 sends the message to be broadcast to processors $1,\ldots,p-1$ in turn (a "sequential"-looking algorithm).

Analysis of Algorithm 2.

The communication time of Algorithm 2 is $1 \cdot \max\{L, (p-1) \cdot g\}$ (in a single superstep, the message is replicated $p-1$ times by processor 0).

3.28.4 Broadcasting: Algorithm 3

Algorithm 3. Both Algorithm 1 and Algorithm 2 can be viewed as extreme cases of an Algorithm 3. The main observation is that up to L/g words can be sent in a superstep at a cost of L. Then, It makes sense for each processor to send L/g messages to other processors. Let $k-1$ be the number of messages a processor sends to other processors in a broadcasting step. The number of processors with the message at the end of a broadcasting superstep would be k times larger than that in the start. We call k the degree of replication of the broadcast operation.

Architecture independent Algorithm 3. In each round, every processor sends the message to $k-1$ other processors. In round $i = 0, 1, \ldots$, each processor j with index $j < k^i$ sends the message to $k-1$ distinct processors numbered $j + k^i \cdot l$, where $l = 1, \ldots, k-1$. At the end of round i (the $(i+1)$-st overall round), the message is broadcast to $k^i \cdot (k-1) + k^i = k^{i+1}$ processors. The number of rounds required is the minimum integer r such that $k^r \geq p$, The number of rounds necessary for full dissemination is thus decreased to $\lg_k p$, and the total cost becomes $\lg_k p \max\{L, (k-1)g\}$.

At the end of each superstep the number of processors possessing the message is k times more than that of the previous superstep. During each superstep each processor sends the message to exactly $k-1$ other processors. Algorithm 3 consists of a number of rounds between 1 (and it becomes Algorithm 2) and $\lg p$ (and it becomes Algorithm 1).

```
1   void  Broadcast (message M, int p, int k)  {
2    /* M message to be broadcast to p processors  */
3    /* Degree of replication is k;  PRAMs use k=2 */
4       pid       =  bsp_pid() ;
5       mask_pid =   1         ;
6       while   (mask_pid <  p ) {
7          if ( my_pid < mask\_pid )
8             for ( i=1, j=mask_pid ; i<k ; i++, j+=mask\_pid ) {
9                target_pid = my\_pid + j ;
10               if ( target_pid <  p )
11                  bsp_put (target_pid ,&M,&M,0, sizeof(M)) ;
12            }
13         bsp_sync()} ;
14         mask_pid = mask_pid * k;
15  }
```

Figure 3.33: BSP broadcasting

3.28.5 Broadcasting $n > p$ words: Algorithm 4

Now suppose that the message to be broadcast consists of not a single word but is of size $n > p$. Algorithm 4 may be a better choice than the previous algorithms as one of the processors sends or receives substantially more than n words of information. There is a broadcasting algorithm, call it Algorithm 4, that requires only two communication rounds and is optimal (for the communication model abstracted by L and g) in terms of the amount of information (up to a constant) each processor sends or receives.

Algorithm 4.
Two-phase broadcasting
The idea is to split the message into p pieces, have processor 0 send piece i to processor i in the first round and in the second round processor i replicates the i-th piece $p-1$ times by sending each copy to each of the remaining $p-1$ processors (see attached figure).

3.28.6 PPF

Exercise. What can you say about parallel prefix? Analyze the BSP performance of the PRAM algorithm for parallel prefix. Can you halve its number of supersteps yet maintain the same BSP cost?

The structure of the four algorithms described for broadcasting can also be used to derive algorithms for parallel prefix that require similar number of supersteps (at most twice as many).

Algorithm 1 gives rise to a "sequential"-like parallel prefix algorithm. Algorithm 2 gives rise to the binary tree based algorithm that requires $2\lg n$ supersteps. The corresponding PRAM algorithm, however, (that also runs on the butterfly) requires half as many supersteps and is thus more efficient on the BSP model. Algorithm 3 gives rise to the equivalents of 2 when the number of supersteps needs to be decreased.

We can generalize the prefix problem so that each processor instead of holding a single scalar value, holds a sequence/vector of scalar values n. In the case $n > p$, implementations following the counterparts of Algorithm 1,2 and 3 for broadcasting fail to provide optimal algorithms.

Algorithm 4 gives rise to a two-phase parallel prefix algorithm that is more efficient in architectures with large L for large independent prefix problems n.

3.28.7 Matrix Computations

SPMD program design stipulates that processors executes a single program on different pieces of data. For matrix related computations it makes sense to distribute a matrix evenly among the p processors of a parallel computer. Such a distribution should also take into consideration the storage of the matrix by say the compiler so that locality issues are also taken into consideration (filling cache lines efficiently to speedup computation). The are various ways to divide a matrix. Some of the most common one are described below.

One way to distribute a matrix is by using block distributions. Split an array into blocks of size $n/p_1 \times n/p_2$ so that $p = p_1 \times p_2$ and assign the i-th block to processor i. This distribution is suitable for matrices as long as the amount of work for different elements of the matrix is the same.

The most common block distributions are.

- **column-wise** (block) distribution. Split matrix into p column stri pes so that n/p consecutive columns form the i-th stripe that will be stored in processor i. This is $p_1 = 1$ and $p_2 = p$.
- **row-wise** (block) distribution. Split matrix into p row stripes so that n/p consecutive rows form the i-th stripe that will be stored in processor i. This is $p_1 = p$ and $p_2 = 1$.
- **block** or **square** distribution. This is the case $p_1 = p_2 = \sqrt{p}$, i.e. the blocks are of size $n/\sqrt{p} \times n/\sqrt{p}$ and store block i to processor i.

There are certain cases (eg. LU decomposition, Cholesky factorization), where the amount of work differs for different elements of a matrix. For these cases block distributions are not suitable. In block cyclic distributions the rows (similarly for columns) are split into q groups of n/q consecutive rows per group, where potentially $q > p$, and the i-th group is assigned to a processor in a cyclic fashion.

- **column-cyclic** distribution. This is an one-dimensional cyclic distribution. Split matrix into q column stripes so that n/q consecutive columns form the i-th stripe that will be stored in processor $i\%p$. The symbol % is the mod (remainder of the division) op erator. Usually $q > p$. Sometimes the term **wrapped-around column** distribution is used for the case where $n/q = 1$, i.e. $q = n$.
- **row-cyclic** distribution. This is an one-dimensional cyclic distribution. Split matrix into q row stripes so that n/q consecutive rows form the i-th stripe that will be stored in processor $i\%p$. The symbol % is the mod (remainder of the division) op erator. Usually $q > p$. Sometimes the term **wrapped-around row** distribution is used for the case where $n/q = 1$, i.e. $q = n$.
- **scattered** distribution. Let $p = q_i \cdot q_j$ processors be divided into q_j groups each group P_j consisting of q_i processors. Particularly, $P_j = \{jq_i + l \mid 0 \leq l \leq q_i - 1\}$. Processor $jq_i + l$ is called the l-th processor of group P_j. This way matrix element (i, j), $0 \leq i, j < n$, is assigned to the $(i \bmod q_i)$-th processor of group $P_{(j \bmod q_j)}$. A scattered distribution refers to the special case $q_i = q_j = \sqrt{p}$.

The BSP algorithm for matrix multiplication presented below was presented in the seminal work of Valiant. It works for $p \leq n^2$. Each processor is assigned the task of computing an $n/\sqrt{p} \times n/\sqrt{p}$ submatrix of the product $A \times B$. The input matrices A and B are divided into p block-submatrices, each one of dimension $m \times m$, where $m = n/\sqrt{p}$. We call this distribution of the input among the processors *block* distribution. This way, element $A(i, j)$, $0 \leq i < n, 0 \leq j < n$, belongs to the $(j/m) * \sqrt{p} + (i/m)$-th block that is subsequently assigned to the memory of the same-numbered processor. Let A_i (respectively, B_i) denote the i-th block of A (respectively, B) stored in processor i. With these conventions the algorithm can be described in Figure 3.34. The following Proposition describes the performance of the aforementioned algorithm.

begin MULT_A (C,A,B,n,p)
1. Let $m = n/\sqrt{p}$;
   Each processor is also assigned a unique processor number q;
2. Let $p_i = q \bmod \sqrt{p}$; $p_j = q/\sqrt{p}$; $C_q = 0$;
3. $a_l \leftarrow A_{p_i + l*\sqrt{p}}$, $0 \leq l < \sqrt{p}$;
4. $b_l \leftarrow B_{p_j*\sqrt{p}+l}$, $0 \leq l < \sqrt{p}$;
5. **for** $0 \leq l < \sqrt{p}$ **do**
   $C_q = C_q + a_l \times b_l$;

end MULT_A

Figure 3.34: Procedure MULT_A.

3.28.8 Mult A algorithm

Example-Proposition 3.8. *Algorithm* MULT_A *for multiplying two $n \times n$ matrices A and B stored according to the block distribution requires, for any $p \leq n^2$, computation time $C_{mul}(n)$ that is given by*

$$C_{mul}(n) = \max\{L, \frac{(2n-1)n^2}{p}\},$$

and communication time $M_{mul}(n)$ that is given by the expression

$$M_{mul}(n) = \max\{L, g\frac{2n^2}{\sqrt{p}}\}.$$

One immediately realizes that algorithm MULT_A is not memory efficient since it requires more local memory per processor – by a factor of $\sqrt{p}$ – than the required one. Algorithm MULT_B shown in Figure 3.35 is the memory efficient variant of MULT_A. It is not synchronization efficient though since its number of supersteps is not constant any more; it has been increased by a factor of $\sqrt{p}$. The performance of algorithm MULT_B is summarized in Proposition 3.9.

3.28.9 Mult B algorithm

begin MULT_B (C,A,B,n,p)
1. Let $m = n/\sqrt{p}$;
   Each processor is also assigned a unique processor number q;
2. Let $p_i = q \bmod \sqrt{p}$; $p_j = q/\sqrt{p}$; $C_q = 0$;
3. **for** $0 \leq l < \sqrt{p}$ **do**
   begin
4. $a \leftarrow A_{((p_i+p_j+l) \bmod \sqrt{p})*\sqrt{p}+p_i}$;
5. $b \leftarrow B_{((p_i+p_j+l) \bmod \sqrt{p})+p_j*\sqrt{p}}$;
6. $C_q = C_q + a \times b$;
   end

end MULT_B

Figure 3.35: Procedure MULT_B.

Example-Proposition 3.9. *Algorithm* MULT_B *for multiplying two $n \times n$ matrices A and B stored according to the block distribution requires, for any $p \leq n^2$, computation time $C_{mul}(n)$ that is given by*

$$C_{mul}(n) = \sqrt{p}\ \max\{L, \frac{(2n-1)n^2}{p^{3/2}}\}$$

and communication time $M_{mul}(n)$ that is given by the expression

$$M_{mul}(n) = \sqrt{p}\ \max\{L, g\frac{2n^2}{p}\}$$

3.28.10 Experimental Results

In order to show the efficiency of algorithm design on the BSP model we present some experimental results for matrix multiplication on Cray T3D; additional results can be found in the author's Web page. Algorithm MULTT_B is a variation of MULT_B where in order to multiply A with B, matrix A is first transposed and the loop for matrix multiplication is changed accordingly. This way the access patterns for both A and B are the same (column - column as opposed to row - column) thus improving locality (cache usage), and subsequently program performance.

3.28.11 C

Finally, we outline a matrix multiplication algorithm that is computation, communication and synchronization efficient. It fails, however, to be memory efficient, as its memory requirements are a multiplicative factor $p^{1/3}$ from the optimal. Algorithm MULTT_C is outlined in the remainder of this section.

Algorithm MULT_B								
	$p=1$		$p=4$		$p=16$		$p=64$	
n	Time (sec)	Mfl rate	Time (sec)	Mfl rate	Time (sec)	Mfl rate	Time (sec)	Mfl rate
256	4.1	7.9	1.1	7.8	0.28	7.4	0.03	13.9
512	34.0	7.8	8.4	7.9	2.1	7.7	0.56	7.4
1024	289.8	7.4	68.4	7.8	16.9	7.9	4.3	7.7
2048	-	-	-	-	136.8	7.8	33.8	7.9

Table 3.1: Execution time for MULT_B on the Cray T3D

Algorithm MULTT_B								
	$p=1$		$p=4$		$p=16$		$p=64$	
n	Time (sec)	Mfl rate	Time (sec)	Mfl rate	Time (sec)	Mfl rate	Time (sec)	Mfl rate
256	2.3	14.3	0.58	14.4	0.15	13.7	0.03	15.1
512	20.7	12.9	4.7	14.1	1.16	14.4	0.30	13.5
1024	202.7	10.5	41.7	12.8	9.4	14.1	2.3	14.3
2048	-	-	-	-	83.5	12.8	19.0	14.1

Table 3.2: Execution time for MULTT_B on the Cray T3D

In MULTT_C matrices A and B (and the result C) are split into two ways into submatrices. Each matrix (A, B and the result C) is split into p "physical" block-submatrices, as in the previous algorithms, each of size $n/p^{1/2} \times n/p^{1/2}$. A "physical" block-submatrix indicates the part of the matrix stored in a single physical (processor) location (i.e. block-submatrix A_i is stored in processor i). At the same time, each of the three matrices is split into $p^{2/3}$ "virtual" block-submatrices each of size $n/p^{1/3} \times n/p^{1/3}$. A "virtual" block-submatrix indicates the block geometry that will be used in the matrix multiplication algorithm to be outlined below. The elements of a "virtual" block-submatrix may be stored in more than one physical processors.

Whereas in the first two algorithms "physical" and "virtual" block-submatrices coincided in number and dimension, in this communication efficient algorithm are clearly distinguished.

Let the "virtual" block-submatrices be identified as A_{ij}, B_{ij} and C_{ij}. Matrix multiplication will thus require the computation of all $C_{ij} = \sum_{k=1}^{p^{1/3}} C_{ijk} = \sum_{k=1}^{p^{1/3}} A_{ik}B_{kj}$, where $C_{ijk} = A_{ik}B_{kj}$.

The algorithm consists of the following steps. We name the processors (i,j,k) the way we did in the matrix multiplication algorithm on the hypercubic networks.

Step 1. Processor (i,j,k) gets A_{ik} and B_{kj}. Note that each of these two "virtual" block-submatrices may originate from more than one processors. Each processor sends at most $2n^2/p$ elements (but each one replicated $p^{1/3}$ times) and receives at most $2n^2/p^{2/3}$ elements. The communication cost of Step 1 is $\max\{L, 2gn^2/p^{2/3}\}$. Subsequently, the two submatrices are multiplied as in the sequential case a step requiring at most $\max\{L, 2n^3/p\}$ time. Partial-submatrix C_{ijk} is thus computed on processor (i,j,k). Each element of such a submatrix is a partial sum of an element c_{lm} of the result matrix C.

Step 2. Each element of C_{ijk} is transmitted from (i,j,k) to that physical processor that stores the "physical" block-submatrix of C whose elements will be formed as sums of the receiving elements (partial sums) of C_{ijk}. Note that each (i,j,k) processor may send its elements to more than one physical processors. At the completion of this step, each of the p processors storing a block-submatrix of C of dimension $n/p^{1/2} \times n/p^{1/2}$ receives at most $p^{1/3} \cdot n^2/p$ such elements (partial sums). The complex communication performed in this step requires time $\max\{L, gn^2/p^{2/3}\}$.

Step 3. The received partial sums are added. $p^{1/3}$ partial sums are summed to give an element of C stored at a physical processor, for a total of n^2/p such elements (of a "physical" block-submatrix). The total computation time performed is $\max\{L, n^2/p^{2/3}\}$.

Example-Proposition 3.10. *Algorithm* MULT_C *for multiplying two $n \times n$ matrices A and B stored according to the block distribution requires, for any $p \leq n^2$, computation time $C_{mul}(n)$ that is given by*

$$C_{mul}(n) \leq \max\{L, 2n^3/p\} + \max\{L, n^2/p^{2/3}\},$$

and communication time $M_{mul}(n)$ that is given by the expression

$$M_{mul}(n) = \max\{L, 2g\frac{n^2}{p^{2/3}}\} + \max\{L, g\frac{n^2}{p^{2/3}}\}.$$

The optimality in communication of the algorithm is established by the following result.

Theorem 3.27. *On a model of computation that allows the operations $\{+, *\}$ only, if any processor reads s elements of A and B and computes at most s partial sums of C, it can compute at most $O(s^{3/2})$ multiplicative terms for these sums.*

This way, if a processor reads at most s elements of A and B it can compute at most $O(s^{3/2})$ multiplicative terms of C. Combined, all p processors can compute $p\ O(s^{3/2})$ such terms which must be $\Omega(n^3)$. Therefore $s = \Omega(n^2/p^{2/3})$ and thus algorithm MULT_C is communication optimal.

How can one prove the Theorem? It suffices to show that if A has s 1's in arbitrary posisition (all other positions are 0) and so has B, then the product $C = A \times B$ requires at most $O(s^{3/2})$ non-trivial multiplica tions (i.e. multiplications where both terms are non-zero). This can be proved by considering two sets of rows for A, the small ones having at most $\sqrt{s}$ ones on each such row and the large one. Call A_s the submatrix of A forme d by these small rows, and A_l the submatrix consisting of the large rows having at least

$\sqrt{s}$ ones. We can only have at most $\sqrt{s}$ rows in A_l. Consider $A_l \times B$. The results can contribute at most $s^{3/2}$ non-trivia l multiplications in C. The reason for that is that each row of A_l can contribute s ones when multi plied with B since B has only s ones. There are at most $\sqrt{s}$ in A_l, i.e . claim follows. For $A_s \times B$ just note that each row of A_s has at most $\sqrt{s}$ ones. Since only s terms are computed in C, and a term involves a row of A_s that has at most $\sqrt{s}$ elements, this $A_s \times B$ product can only involve at most $s^{3/2}$ multiplications.

3.29 The LogP Model

The LogP model (Culler, Karp, Patterson, Sahay, Schauser, Santos, Subramonian and von Eicken, 1993) has also been suggested as a realistic model for the design of parallel algorithms that work predictably well on a wide range of parallel machines. It models a parallel machine as a distributed memory multiprocessor in which processors communicate by point-to-point messages. It is an *asynchronous* model that does not enforce synchronization of the processors, as the BSP model does. If processor synchronization is required in a program, the programmer must provide it. A parallel machine under the LogP model can be characterized by the following tuple (p, L, g, o) of parameters. Each parameter is explained in more detail below (note that LogP may use the same notation for some of its parameters as the BSP model but these may have different meaning).

p: The number of processor/memory components (as in the BSP model).

L: an upper bound on the *latency* or delay of the machine, incurred while communicating a message of very small size (one or a small number of words) from a source to a destination component. In other words, such messages are delivered by the router within time L.

o: the *overhead*, defined as the length of time that a processor is engaged in the transmission or receipt of each message (i.e the time required for the submission of a message to the router or acquisition of a received message from it).

g: the *gap*, defined as the minimum time interval between consecutive message transmissions or consecutive message receptions at a processor. $1/g$ gives the bandwidth of the system per processor.

In addition, the communication capacity of the system is limited. This means that at most L/g messages can originate or be destined to any processor at any time. If a processor needs to transmit a message that would violate this condition, it stalls until this transmission is possible (i.e. there is some available capacity). Writing under the LogP model into a remote memory location takes time $L+2o$.

Although BSP and LogP are similarly powered models (one can simulate a BSP computation on the LogP and the other way around), the BSP model seems to be more usable as a theoretical model for the design and analysis of parallel algorithms and as a programming paradigm for writing parallel software that is scalable and transportable (portable and efficient among a variety of hardware platforms).

The importance of BSP is that it introduces a new abstraction for communication in terms of the BSP parameters L and g so that considerations in programming parallel machines move from a local (detailed) level to a global (abstract) one. The two parameters abstract all communication and synchronization issues related to parallel computing and allow the design of software (a unifying property) that work on any machine independent of the underlying architecture (eg. shared memory vs distributed memory).

Under the BSP model, an algorithm designer describes the performance of a parallel algorithm in terms of p, L, g and problem size n. A collection of algorithms that solves a given problem can then be formed with varying performance characteristics (for example, one algorithm may be suitable for machines with small L, another for machines with small g or n, and so on). For a given machine whose BSP parameters are known or are measurable, that algorithm is chosen from the collection whose performance (computation and communication) on the particular machine is the best available.

3.30 Programming models and approaches: Process vs Thread

We write parallel programs usually using the **SPMD** (Single-Program Multiple-Data) approach. A directive-based model can also be used where high-level constructs are used that leaves further details to the compiler. If `SPMD` is used, one program is written that contains all the information needed for execution by a multiplicity of processors. Each processor determines which fraction of the code it will execute based on directives within the code that determine the thread of execution for a given processor. Depending on the used parallel programming library, some other restrictions might also apply. In the general case, one can employ a fully **MIMD** (Multiple-Instruction Multiple-Data) approach by using an **MPMP** (Multiple-Program Multiple-Data) approach. This is not explicitly covered here although several libraries (e.g. Open MPI) could support it.

In the case of an `SPMD` approach, the Single-Program can give rise to a number of processes or threads. Thus we could classify the possible models of programming in three major groups as follows.

a. **Process Model.** The single program spawns multiple processes. All memory is local to the processor. A UNIX-based process approach is used.

b. **Thread Model.** The single program/process spawns multiple threads. All memory is global and can be accessed by all the available threads A Posix-threads API is used. A set of functions is provided for the creation, termination and synchronization of threads. Threads are covered in detail in this work.

c. **Directive-based Model.** One uses high-level compiler directive to specify concurrency. This is the `OpenMP` approach that is not covered here.

d. **Global address space programming languages.** They provide an abstraction that facilitates the use of shared and private data at the same time. The underlying hardware can be typical distributed memory which is not shared. UPC (Unified Parallel C) is an example; we do not cover PC here.

The thread model is rather low-level. Each thread has its own program counter, stack space and register set, and priority. No collective communication operations are available such as those provided by MPI or BSPlib.

Part III

Multi-core computing

Chapter 4

Multi-core computing overview

4.1 What is multi-core computing

In the past 20 years uniprocessor (single core) performance has barely improved. The limitations of CPU clock speeds (around 2-3GHz), power consumption, and heating issues have significantly impacted the improvement in performance through instruction level parallelism. Some minor improvements have due to the increasing size and use of multi-level cached memory hierarchies.

An alternative that has been pursued is the increase of the number of "processors" on a processor die (computer chip). Because we have used the term processor to identify an execution unit inside a single computer (micro)chip, we introduced earlier the term "execution unit" to refer to the "processors" inside processor (or inside a processor's chip). When we deal with multiple execution units within the same chip we usually use the term **core** to refer to them.

Thus if we have two processors with eight execution units each, we say that each processor has eight cores. The total number of execution units is however 16 spanning those two processors.

In the past 20 years in order to increase performance instead or relying to increasing the clock speed of a single processor, we utilize multiple cores that work at the same (or not) clock speed (boost speed), or in several instance at lower clock speeds (regular speed). To support such a multi-core processor architecture, traditional cached memory hierarchies such as the L1 and L2 used by traditional (uniprocessor) architectures are not enough. L1 and L2 caches are usually local to a processor or a single core. A higher memory hierarchy is needed to allow cores to share memory "locally". An L3 cache hierarchy has been made available to support multicore and more recently (2015) L4 cache hierarchies have been tested to support L3 for specific (graphics-related) purposes. The disappearance or lack of dominance of an L4 cached memory shows that such manylevel cache hierarchy approaches have their limitations.

The difference between multicore and manycore is not clearly delineated. One might use the latter term if the number of core is greater than say 50 (as in Intel's now defunct Phi architecture). Such architectures usually sacrifice the L3 cache for more control logic (processors). To allow inter-core communication the L2 caches are linked together to form a sort of L3 cache.

The effective use of multiple cores requires familiarity with multiprocessing development. Traditional parallel programming is not more different in a multicore environment. Performance differences, however, exist. The multiple cores are physically closer, inter-core communication utilizes highly advanced bus/crossbar architectures, and latency considerations in communication become less important than say the case where processors are interconnected with a slow (eg. ethernet-based) interconnection device (eg. switch).

A process can be defined as a program that is being executed. One can define a thread to be a light-weight process. Each process maintains its own state information (eg. open files) and has its own address-space and

interprocessor communication requires facilitation by the operating system itself, whereas a thread shares the same address space with the other threads of a given process.

The term **hyperthreading** refers to the ability to run two or more threads on a single chip/die and is a logical form of viewing a single physical processor as multiple cores. Although certain hardware is replicated to allow for hyperthreading, such replication is not sufficient for the on-chip hardware to be called a `multicore`. In multicore chips nowadays, each core is capable (it has hardware support) to run about two threads at a time. Manycore architectures have more enhanced support for hyperthreading.

There are also **hybrid multicore** architectures that mix multiple processor types in a single chip. One such example, is IBM/Sony/Toshiba's Cell Broadband Engine (CBE) architecture.

4.2 Multi-core programming requirements

The effective use of the extra cores requires parallelization and optimized memory usage since one needs to manage multiple memory hierarchies. Besides the main (eg. system) memory, **cache** memory is available in the hierarchy between the processor and its main memory. While the cache is not as efficient as the on-chip registers, it is faster than main memory; it is more abundant than the register set but its size is not as large as that of main memory. Its existence allows for more effective memory transfer rates thus increasing a processor's performance since small segments of memory are speculatively fetched from main memory on the expectation that a program code will exhibit **temporal** or **spatial** locality.

In multi-core computing, parallelization is important but so is efficient memory utilization through the available memory hierarchies. If all cores must access the same main memory, at the same time problems can arise. To avoid such problems, one needs to utilize the cache hierarchy.

Locality of reference becomes important.

The piece of data that will be accessed in the next core cycle must already be available in the cache nearby (L1 is preferable over L2, and L2 over L3, and all of them are preferable to main memory); if the piece of data still resides in main memory, competition with other cores will occur at a lower transfer rate than cache transfers. Otherwise performance may deteriorate if it is shared by multiple cores.

Chapter 5

GPU computing

5.1 CPU vs GPU CPUs

A CPU (Central Processing Unit) refers to a traditional microprocessor (a.k.a. processor) that can be unicore (rarely nowadays) or multi-core or many-core.

The number of cores is usually (2023) less than 30 (e.g. Intel's Xeon processors). Sometimes many-core processors (e.g. Intel'Phi) are attached to the CPU and work in 'parallel' with the CPU or independetly of it. In such a case a many-core is called a *coprocessor*.

A GPU (Graphics Processing Unit) is used primarily for graphics processing. *CUDA* (Compute Unified Device Architecture) is an application programming interface (API) and programming model created by NVIDIA. It allows CUDA-enabled GPU units to be used for General Purpose processing, sequential or massively parallel. Such GPUs are also known as GPGPU (General Purpose GPU) when provided with an API for general purpose work. With time this evolved (around 2007) into CUDA, when NVIDIA released a software architecture and computational model that allows one to use the CUDA interface with C or C++ programs. CUDA gives programmers access to the GPU's parallel computing elements and instruction set to exploit the high degree or parallelism of the GPU. Other languages and interfaces are also supported.

Definition 5.1 (*Host and Device*)**.** *The terms host and device refer respectively to the traditional CPU (i.e. Intel or AMD) and the CUDA device (i.e. a GPU) attached to the host.*

Each one has its own memory space, host memory and device memory respectively. Separate mechanisms are being used to allocate and free such memory and allow data transfer between any of the two destinations.

Definition 5.2 (*CUDA GPU abstraction*)**.** *A CUDA GPU is using three abstractions.*

- *a hierarchical order of thread groups,*
- *shared memories, and*
- *a mechanism for a barrier synchronization.*

At a higher level CUDA allows a programmer to split an application (problem) by organizing its data and its computational tasks. Data are organized into groups of coarse-grained data, and computational tasks are organized using task parallelism so that they can independently be solved by blocks of threads.

Each task is then split into finer pieces using fine-grained data and thread parallelism by having all threads in a block cooperating in solving a specific task.

Thus at block level the GPU automatically schedules the execution of blocks of threads to one of several available multiprocessors with the details hidden from the user/programmer.

Definition 5.3 (*Hierarchical division: threads, blocks, grids*)**.** *At the lowest level of the CUDA hierarchy there are threads. Threads form groups known as thread blocks, and thread blocks form other groups known as grids.*

At the lowest level of the hierarchy there are threads. Each thread has its own private memory (registers). Threads form groups known as thread blocks, and threads blocks form grids. The threads in a thread block share memory that is visible to them during the lifetime of the block, and also read-only memory. All threads share/have access to the DRAM (global memory).

The use of multithreading serves several reasons.

- Given that global memory is slow (100s of processor cycles), multithreading allows the processor to switch to a thread that has already received data from the global memory and thus is ready for computation.
- It supports fine-grained parallelism for graphics related applications, and
- provides hardware virtualization support for many-threading as each thread has its own registers, program counter and execution state/status and thus can execute an independent code sequence, and this is accomplished with minimal overhead.

5.2 What is a (CUDA) GPU

At the hardware level, a GPU (device) is attached as a PCI3 card in the motherboard chassis of a traditional desktop PC that has a CPU (host).

A GPU consists of a number of **streaming multiprocessors** known in NVIDIA terminology as SM, SMX (Kepler architecture), or SMM (Maxwell architecture) in different architecture generations (Kepler, Maxwell) of NVIDIA.

Definition 5.4 (*Streaming Multiprocessor (SM)*)**.** *A GPU consists of a number of streaming multiprocessors. Each streaming multiprocessor (SM) has hardware support for multiple streaming processors (SM cores or functional units).*

Generically we shall be using the term SM unless we describe a specific architecture and then we might use the specific name for an SM (e.g. SMX for Kepler). Multiple GPUs may be on the same card and can communicate directly with each other (no host utilization).

An SM is similar to a traditional 'CPU core' but has hardware support for multiple streaming processors ('SM cores' or functional units) and thus it is a highly multithreaded coprocessor to the accompanying CPU (host).

An SM executes many threads in parallel on several multiprocessor cores. Thus we may have multiple GPUs containing multiple SMs that contain multiple SM core that execute multiple threads each.

5.2.1 NVIDIA K20X GPU

A K20X GPU Kepler Accelerator (GK110) has 14 SMX (SM of a Kepler architecture), and each one of them has 512 functional units. These functional units include

- 192 ALU (Arithmetic Logic Units),
- 192 FPU (Floating-Point Unit) for single-precision (SP) floating-point operations,
- 64 FPU for double-precision floating-point operations,

- 32 load/store units, and
- 32 SFU (Special Function Units) for transendental functions (log, sqrt, sin, cos etc).

Thus the 192 ALUs with its 192 FPUs can be viewed as 192 cores of each SMX for a total of 2688 cores altogether for a K20X.

If double-precision computations are to be supported the parallelism of an SM is thus limited to 64 cores per SMX.

Multiprocessors execute in parallel and asynchronously. Thus threads residing in one multiprocessor cannot send data into threads of another multiprocessor.

L1 and shared memory of an SM

Each SM/SMX multiprocessor has registers (register file), a data cache (to be called L1 even if it differs from a CPU's L1 cache) and shared memory that is limited in size (tens of kilobytes) and that it is shared by all the cores (e.g. 192) of the SM. In Kepler the same 64KiB of memory can be configured as 16KiB of L1 cache and 48KiB of shared memory, or 32KiB of L1 and 32KiB of shared memory, or 48KiB of L1 and 16KiB of shared memory.

The shared memory is organized in 32, 4B banks. Successive words are accessed through different banks of the shared memory (see example of matrix transposition later on). If multiple threads use the same bank to access different words, serialization takes place If those multiple threads access the same word however, a multicast takes place and the access is completed in one fetch.

In such an architecture, L1 serves as a victim cache/spill memory for the registers.

It is not a very good practice to use the L1 cache of a GPU the same way an L1 is being used in a CPU i.e. to cache (to move into faster memory) a block of memory that is to be used in an imminent computation.

This is because 100s or 1000s of threads spread over all cores of an SM would be competing for the L1 cache of that SM. In fact it would be better to use the shared memory as the latter is shared among all the threads of an SM; moreover no eviction will ever take place from shared memory!

L2 for all SMs

An L2 memory might be (and currently is) available and it is shared among all SM multiprocessors and their cores (and their threads). No cache coherence is available.

If a register does not contain the information but it is in L1, a 128B transfer is initiated; if data is not in L1 but in L2, then a 32B transfer is initiated.

Global memory of the device

Global memory (of the device) is faster than the host CPU memory (2-3 times on the average) but it is 100-300 times slower than the registers of the SM multiprocessors and their cores, and at least 5-30 times slower than L1 cache and the shared memory and 2-3 times slower than the L2 cache.

Communication between host and device is through PCI3 at more than 10GB/s and up to 85GB/s with close to 800ns latency or so.

For NVIDIA architectures and their SM and core configurations see Figure 5.1.

5.2.2 NVIDIA Tesla GPU

An NVIDIA Tesla C2075 GPU has 14 SM each of which has 32 cores, and thus $32 \cdot 14 = 448$ threads may be running simultaneously.

Architecture	Tesla	Fermi	Kepler	Maxwell
Timeframe	2006-2009	2010-2011	2012-2014	2014-2015
CUDA	1.10-1.2	2.0-2.1	3.0-3.5	5.0
Number of SM	16-30	7-16	8-30	4-5
Number of cores/SM	8	32-48	192	128
Number of cores	128-240	336-512	1536-5760	512-640

Figure 5.1: Grids, Blocks, Threads (CUDA 5.0 and higher); typical ranges

The base clock speed is 575MHz but cores run at twice that speed at 1.15GHz. Memory is 384-bit wide and 6GiB in size operating at 3000MT/sec with effective bandwidth of 144GB/sec. Single precision flop rate is 1030.4GFLOPS and double precision is 515.2GFLOPS. TDP (Thermal Design power) is 225W.

By comparison a K20X GPU Kepler Accelerator (GK110), as stated earlier has 14 SMX and 512 functional units thus providing 192 cores per SMX and 2688 cores atogether, has 732MHz base clock rate but cores operate at that speed and not at double of that as they did in Tesla; the same 384-bit bus width is utilized and same device memory, with 2×2600MT/s for 250GB/s bandwidth (ECC enabling might affect performance) .

Its single and double precision performance is 3935 ($= 732 \times 192 \times 14 \times 2$) and 1310 GFLOPS respectively. (This is under the assumption that all instruction are multiply-and-add; for regular operations peak SP performance is half of that at 1967 GFLOPS.) TDP (Thermal Design power) is 235W.

The L1/shared memory is 64KiB per SMX. L2 cache is 1.5MiB. The SRAM used operates at the same 732MHz. The effective bandwidth of L1/shared memory is about 187-1330GB/s and the L2 cache can support 1024B/cycle.

In more recent architecture realizations (Maxwell) the number of cores per SM has been reduced to 128 from 192, and so has the number of SM down from 4-5 from 8-30.

5.2.3 NVIDIA Fermi GPU

An NVIDIA Fermi GPU has 16 SM. They share a common L2 cache. 1.5GHz. Each SM has 1 instruction cache, 2 warp schedulers and 2 dispatch units, a register file of 32768 32-bit registers, 2×16 CUDA cores, 16 LOAD/STORE unit, 4 SFU transcedental function units, 16 double precision units allowing for the same number of MultiplyAndAdd operations per clock, and 32 single precision units.

Each CUDA kernel ca use up to 63 registers.

64KiB L1 cache which is a register spillover memory of individual threads or a shared memory of all threads of a block. It is configured as $(16KiB, 48KiB)$ or $(48KiB, 16KiB)$. It has 10-20 cycle latency and 1600GiB/s throughput.

It interfaces with the L2 cache. Memory that cannot fit into the register can spill over local memory (mapped global memory).

An L2 cache is used as a unified memory for all SM but also as a texture (graphics) memory. It has size 768KiB.

Global memory is also accessible by the host but has high latency (400-800 cycles). 1GHz operation, and 384bit data bus width accommodating $2 \times (384/8) \times 1GHz = 96$GiB/s.

Peak performance is 2 (Multiply and Add) operations per core (32 cores) per GHz float performance; half of it for double performance. Peak performance is $2 \times 32 \times 16 \times 1.5GHz = 1.5$Tflop/s.

5.2.4 NVIDIA Turing GPU

An NVIDIA Turing GPU (2018) has 16-72 SM with 64 cores per SM for a total of up to 4608 cores. L1 cache is 96KiB per SM (1.5-6.75MiB), L2 cache is 1-6MiB and power consumption to 280W.

5.2.5 NVIDIA Volta GPU

An NVIDIA Volta V100 GPU (2018) has 80 SM with 64 cores per SM for a total of 5120 cores. GPU global memory is 32GiB. Memory bandwidth is 900GiB/s. The 5120 cores operate at 1.370GHz. A shared L2 cache has size 6MiB. Warp size remains 32.

Peak performance double is 7.450Tflop/s derived as $1 \times 1370MHz \times 80 \times 64$.

5.2.6 NVIDIA Ampere A100 GPU

An NVIDIA Ampere A100 GPU (2020) has 108 SM with 64 INT32, 64 FP32 (or 32 FP64) cores for a total of 6912 INT32, 6912 FP32 (3456 FP64) cores. Frequency is 765MHz with boost 1410MHz. Thus an SM has 32FP64 cores, 64FP32, and 64 INT32 cores.

Global memory size is 40GiB (1555GiB/s bandwidth). Register file size is 256KiB/SM. Shared memory is at most 164KiB per SM, L1 cache size is 192KiB per SM. L2 cache size is 40MiB for all SM. A 5120bit that is 640B data path leads to the L2 cache allowing for a 1555GiB/s bandwidth. A 32B cache lines allows for 8 floats or 4 doubles. Power dissipation is 400W for 54.2billion transistors.

Peak performance float is 10.575Tflop/s derived as $2 \times 765MHz \times 108 \times 64$. Peak performance double is 5.287Tflop/s derived as $1 \times 765MHz \times 108 \times 64$. Boosted performance is 19.491Tflop/s and 9.746Tflop/s for float and double respectively.

5.2.7 NVIDIA Hopper H100 GPU

An NVIDIA Hopper H100 PCIe has 114 SM per GPU. It operates at 1125MHz with boost speed at 1755MHz. Peak performance float is 51.217Tflop/s and double performance 25.608Tflop/s.

IT has 128 FP32 cores, 64 FP64 cores and 64 INT32 cores. for a total of 14592 FP32, 7296 INT32 cores, and 7296 FP64 cores.

Global memory size is 80GiB (2000GiB/s bandwidth, memory clock 1593MHz, 640B bus width). Share memory size is at most 228KiB and L2 is 50MiB. Register file size is 256KiB with 300-350W dissipation for 80 billion transistors.

5.3 Heterogeneous programming

The device attached to a host operated in conjunction with the host as a coprocessor to it. The compute capability (CUDA) of a device is expressed as a version number X.Y, where X is the version number and Y a minor subversion/revision number. If X is the same it means devices are using the same architecture. Thus a Maxwell architecture has $X = 5$, a Kepler $X = 3$, a Fermi $X = 2$, and a Tesla architecture $X = 1$.

C code can be translated into assembly form (i.e. what is known as PTX code) or binary form (i.e. into a `cubin` object). PTX code using just-in-time compilation is further compiled into binary code by a specific device driver. This way an application for which PTX code is available can be run on new architectures as soon as a specific device driver is provided.

For PTX instruction generation architecture specific requests are using the `-arch` option as in `-arch=sm_30`. Such PTX code can always get converted into a cubin with at least the X and Y values of the PTX code. To enforce binary code output use in the compiler the `-code` option. Thus `-code=sm_30` will produce binary

code for devices with compute capability 3.0. A cubin generated for X.Y will only execute on devices X.Z where $Z \geq Y$.

Device code can be 32-bit or 64-bit and `-m32` or `-m64` selects the corresponding compiler option. A cudeDeviceReset executed by the host destroys the primary context of the device the host thread currently is operating on. Whereas cudaMalloc and cudaFree is the mechanism to allocate linear memory in the device, alternative mechanisms exists for 2D or 3D memory such as cudaMallocPitch() and cudaMalloc3D(). Moreover cudaMemcpy2D() and cudaMemcpy3D() can be used for memory transfer. A thread (t_x, t_y) accessing a 2D array of width `w` at base address a, retrieves the element $a + w * t_y + t_x$. Asynchronous transfers between the host and the device are possible through the explicit use of streams. (See Section 3.2.5.5 of the Cuda C Programming Guigde by NVIDIA, page 33, 2015).

Extensions	File
.cu :	CUDA source file host or device
.h .cuh :	Header files
.c :	C
.cc :	C++
.cpp :	C++
.cxx :	C++
.o :	Linux object
.obj :	Windows object
.a :	Static library Linux
.lib :	Static Library Windows
.so :	Shared Object
.gpu :	Intermediate Compilation File (device)
.ptx :	Portable Device Assembly Format
.cubin :	Cuda Binary for a specific architecture (GPU)

Figure 5.2: CUDA file extensions

Note that single-precision operation should be preferred. Functions `sinf, cosf, tanf, sincosf` and their DP counterparts are expensive and become even more expensive if the function argument is a large value (in magnitude). To do i/n for n a power of 2, it is better to do $i \gg \lg 2(n)$, and $i \pmod n$ is equivalent to $i\&(n-1)$.

Control flow instructions (if, switch, for, while) can cause threads to diverge and the different execution paths will be serialized.

	CUDA 3.0	CUDA 3.5	CUDA 5.0
32-bit +,*, *+	192	192	128
64-bit +,*, *+	8	64	4
32-bit 1/, sqrt	32	32	32
log2,exp2			
32-bit int add,int sub	160	160	128
32-bit int *,int *+	32	mi*	mi*
32-bit shift	32	64	64
cmp,min,max	160	160	64
32-bi OR,AND,XOR	160	160	128
sum—a-b—	32	32	64
type convert 8,16,32	128	128	32
type to/from 64	8	32	4
other conversions	32	32	32
*mi = multiple instructions			

Figure 5.3: Arithmetic Operations (number of ops per clock cycle per SM)

CUDA	OpenCL
Grid	Grid
Block	Work Group
Thread	Work Item
__lobal	*__kernel*
__device	*__global*
__shared	*__local*
__local	*__private*
__syncthreads	barrier()
threadIdx.x	get_local_id(0)
threadIdx.y	get_local_id(1)
threadIdx.z	get_local_id(2)
blockIdx.x	get_group_id(0)
blockIdx.y	get_group_id(1)
blockIdx.z	get_group_id(2)

Figure 5.4: Correspondence between OpenCL and CUDA

Part IV

CUDA SIMT Computing

Chapter 6

CUDA computing

6.1 CPU execution model : SIMD

In SIMD (Single Instruction Multiple Data) execution, a task executes the same instruction as any other task. The difference is that different tasks operate on different data.

Example 6.1. *Let us assume that we have four tasks identified with a task ID number denoted by tid. Let tid values be in the range* 0..3*. The same instruction* $\mathbf{c}[\mathbf{tid}] = \mathbf{a}[\mathbf{tid}] + \mathbf{b}[\mathbf{tid}]$ *operates on different data on the four tasks available.*

$$\begin{aligned}
\textbf{Task0}: &\quad c[0] = a[0] + b[0] \\
\textbf{Task1}: &\quad c[1] = a[1] + b[1] \\
\textbf{Task2}: &\quad c[2] = a[2] + b[2] \\
\textbf{Task3}: &\quad c[3] = a[3] + b[3]
\end{aligned}$$

We can call the instruction a kernel indicated by the __global__ prefix; inside it obtainIDoftask is a variable that makes available the id of the task to variable tid. A task can be a process or a thread.

In the discussion to follow the term kernel does not refer to an operating system's kernel. It refers to the few instructions that several tasks (that will become threads) are going to execute in parallel, thus processing multiple data (e.g. the elements of a vector, matrix, image, packet, etc).

```
  __global__ void add(a,b,c) {
    int tid = obtainIDoftask;
    c[tid]=a[tid]+b[tid];
}
```

Figure 6.1: A kernel

The kernel 6.1 is quite limited; it can deal with arrays of length equal to the number of tasks available.

6.2 GPU CUDA execution model : SIMT

In a GPU (Graphics Processing Unit) available computer we have the host machine (e.g. a PC computer) that has a GPU attached to it in the form of a card inserted into one of the slots of a desktop or laptop PC computer. The PC is known as the host and the GPU as the device. NVIDIA is one company that provides GPU hardware (GPU CPUs). A GPU CPU is a collection of streaming multiprocessors (SM).

In GPU execution, a task is a thread. The same instruction is to be executed by multiple threads on different pieces of data. This is the SIMT (Single Instruction Multiple Thread) model.

For NVIDIA GPUs, the CUDA (Compute Unified Device Architecture) model interprets SIMT execution as follows.

An NVIDIA GPU contains many (streaming) multiprocessors. NVIDIA refers to them as streaming multiprocessors (SM). An SM core (functional execution unit in an SM) is SIMT. CUDA programs are thus SIMT to match the available hardware (SM core). SIMT computing requires the presence of two functionalities: **SI** and **MT**. These are as follows.

SI Availability of the common SI (Single CPU Instruction) to be executed: the single CPU instruction is part of a sequence of CPU instructions that are generated after compilation of a function written in a high-level programming language such as C or C++. The function is known as the **kernel** in GPU CUDA terminology.

MT Availability of MT (Multiple Threads) that each one will execute the CPU instructions of the kernel, one (common) CPU instruction at a time. The threads are organized into blocks (i.e. thread blocks), and blocks are organized into a **grid**. The grid (of blocks) and the block (of threads) can have regular 1D, 2D or 3D shapes.

In order to start a GPU hosted computation we launch a kernel grid. The kernel describes the C/C++ function (SI part) and the grid the group of blocks of threads (MT part) that would execute the kernel.

When a program is written for a CUDA-enabled device (GPU) it is usually written at the host and compiled by the NVIDIA framework/compiler (known as nvcc) that determines what part is going to be executed at the host and thus invokes the host's compiler infrastructure or what part is going to be executed by the CUDA device and thus the device's compiler infrastructure is to be used. The host code executed at the host can include sequential (serial) and parallel code. The device code is a parallel function (kernel) written in CUDA. A program in CUDA thus operates as follows:

(1) data are copied from host (CPU) memory into device (GPU) memory,

(2) a kernel grid is then launched, and the device code is then executed, and while this is being done, device memory is being used and cached on (device) chip, and

(3) when the computation is completed results are then transferred from the device (GPU) memory back to the CPU memory.

6.2.1 Threads in CUDA

Threads in CUDA are organized hierarchically and have their own program counters (PC) and registers (register file).

Definition 6.1 (*CUDA tasks : threads*)**.** *A CUDA task is a thread.*

A thread is an entity that represents the execution of a kernel.

Definition 6.2 (*Kernel*)**.** *A kernel is a high level programming language function that is being invoked by the host to execute on a CUDA device by a specified aggregation of threads referred to as a grid.*

In the CUDA model, threads do not exist independently.

Multiple threads that will be associated with a kernel and execute that kernel are combined into a thread block or just block. A (thread) block is an aggregation of threads that can have a regular 1D (one dimensional), 2D (two-dimensional), or 3D (three-dimensional) shape.

Definition 6.3 (*Threads in a thread block*)**.** *A block of threads (or thread block) contains a number of threads shaped into regular 1D, 2D, or 3D shapes.*

Multiple thread blocks (of the same shape) form a grid. A grid can have a regular 1D, 2D, or 3D shape consisting of blocks (of threads) of the same shape.

Definition 6.4 (*Grid of thread blocks*)**.** *A grid contains one or more thread blocks. A grid executes (or launches) a kernel: each thread of the thread blocks of the grid execute SIMT-style the instructions of the kernel.*

The threads of a grid (consisting of blocks of threads) execute a kernel, which is the common set of CPU instructions that will be thus executed SIMT-style by all the threads (of the thread blocks) of the grid.

Several grids can co-exist and execute a unique (usually different) kernel each.

A running instance of a kernel is executed by the threads of the grid associated with that kernel.

Definition 6.5 (*Multiple Grids*)**.** *In a program multiple grids can launch multiple kernels, but each grid can launch one and only one kernel.*

Definition 6.6 (*Shape of a grid or block*)**.** *The threads of a block or the blocks of a grid can be organized into 1D, 2D, or 3D regular shapes.*

A thread belongs to a block of threads and blocks of threads are organized into a grid. Thus threads are arranged into a one, two, or three dimensional grid of identically shaped blocks. Blocks of the same shape form a grid. Any block of a grid contains the same number of threads as any other block of that same grid.

6.2.2 CUDA kernel grid launch

In the CUDA model a kernel grid or just kernel is launched by the host to be executed on the device.

A kernel is the set of common instructions that will be executed by the threads that form the (thread) blocks of the kernel (grid). The grid is the structure that describes the shape of the thread blocks associated with the execution of the kernel, and the shape of the threads within each block. Each grid maps to a unique kernel. The number of instances (paralel copies) of the kernel is referred to as the number of (thread) blocks. Every (thread) block of the kernel contains the same number of threads. The blocks inside the grid can form arbitrary regular shapes: 1D, 2D, 3D rectangular shapes. So can the threads of a block.

Prior to (or At) the launch of the kernel, the grid of thread blocks and the threads of each block need to be defined and described. Thus we call such a launch either a kernel launch or a kernel grid launch. All the threads of all the blocks of the launched grid are going to perform the same instruction or set of instructions, and these are the instructions of the kernel associated with the launched kernel grid. A Gigathread global scheduler distributes thread blocks to SMs.

Definition 6.7 (*Kernel Launch*)**.** *A kernel is launched according to the prototype of Figure 6.2.* KernelName *is the name of the function describing the set of common instructions to be executed by the threads of each one of the blocks of the grid. Parameter* nblocks *describes the geometry of the grid, and the number of blocks of the grid; parameter* nthrdperB *describes the geometry of a block of the grid and the number of threads per block of the grid.* args *is the argument list of function* KernelName.

A kernel is launched as follows.

$$\text{KernelName} <<< \text{nblocks}, \text{nthrdperB} >>> \text{args};$$

Figure 6.2: A kernel launch

Note that the term SIMT implies a single instruction execution; a kernel might contain several instructions, albeit common to all threads (of the grid). A question that will arise later is what happens if an instruction contains a conditional execution such as that equivalent of an if statement. In an SIMT, execution threads cannot diverge; all threads will execute both the TRUE and FALSE part of an if statement!

Example 6.2. *A kernel launch is specified in Figure 6.3. The name of the kernel that will be launched maps to kernel function* add*. All the threads of the blocks of the grid will execute the same two instructions, as shown in Figure 6.1. The variable nblocks describes the number of launched blocks including their arrangement (geometry) within the (kernel) grid. Variable nthrdperB (for number of threads per block) describes the number of threads per block and their arrangement (geometry) within a thread block.*

$$\text{add} <<< \text{nblocks}, \text{nthrdperB} >>> (a, b, c);$$

Figure 6.3: A kernel launch: Add vector b and c into a

Kernel execution

A single kernel lauch launches a single grid containing at least one and usually several blocks. Each block of the grid has one or more threads. All threads of a grid share the same global memory space.

The threads of a block can cooperate using block-level synchronization and block-local shared memory. Threads from different blocks cannot cooperate easily (unless they use global memory, the RAM of the GPU).

Each thread of a block has its own memory (registers). What cannot fit into a limited number of registers becomes part (spillage) of a shared memory, or becomes part of a local memory (mapped global memory). All the threads of a block have access to limited shared memory (approximately 48KiB for Kepler architectures, growing to approximately 128KiB but at most 256KiB in later architectures). A read-only constant memory is also available for faster access. An L2 memory is also available for global memory access.

All the threads of a block share an instruction stream (SIMT). When there is (instruction stream) divergence the different branches are run sequentially while they are divergent. When they converge, parallelism is restored.

Multiple kernels may be launched on the same GPU. All such kernels share a Global memory.

Definition 6.8 (*Running instance of Kernel*)**.** *A running instance of a kernel is executed by the threads assigned to that kernel, i.e. the threads of the threads blocks of the grid that launched the kernel.*

Definition 6.9 (*An app of several kernels*)**.** *The same application can launch (spawn) and execute several kernels. One grid is associated, launches and executes a unique kernel. A programmer defines the shape of the grid, and the shape of the blocks of the grid.*

6.2.3 Block and Thread identification

Definition 6.10 (*Shape of grid: gridDim*)**.** *The shape of a grid is available through variable* gridDim *that is of type dim3.*

$$\text{gridDim} = (\text{gridDim.x}, \text{gridDim.y}, \text{gridDim.z})$$

For a 2D (two-dimensional) grid $\text{gridDim.z} = 1$*, and for a 1D (one-dimensional) grid* $\text{gridDim.z} = \text{gridDim.y} = 1$*. The number of blocks of a grid is given by the following product.*

$$NumOfBlocks = gridDim.x \times gridDim.y \times gridDim.z.$$

Therefore, variable gridDim provides two pieces of information: (a) the actual shape of a grid, i.e. whether it is one-dimensional (1D), 2D, or 3D, and (b) the number of thread blocks which is the product of the three dimension contributions of the elements of gridDim. A 1D grid has $\text{gridDim.y} = 1, \text{gridDim.z} = 1$, and a 2D grid has $\text{gridDim.z} = 1$, as noted.

Definition 6.11 (*ID of block of a grid: blockIdx*)**.** *Each (thread) block of a grid has a block identification index available through variable* blockIdx *that is of type uint3. For a 3D grid this translates as*

$$\text{blockIdx} = (\text{blockIdx.x}, \text{blockIdx.y}, \text{blockIdx.z})$$

A missing dimension assumes a value zero.

Definition 6.12 (*Shape of a block: blockDim*)**.** *The shape of a (thread) block is available through variable* blockDim *that is of type dim3.*

$$\text{blockDim} = (\text{blockDim.x}, \text{blockDim.y}, \text{blockDim.z})$$

For a 2D block $\text{blockDim.z} = 1$*, and for a 1D block* $\text{blockDim.z} = \text{blockDim.y} = 1$*. The number of threads in a block is given by the following product.*

$$NumOfThreads = blockDim.x \times blockDim.y \times blockDim.z.$$

A missing dimension assumes a value one.

Definition 6.13 (*ID of a thread of a block: threadIdx*)**.** *Each thread of a block has a thread identification index available through variable* threadIdx *that is of type uint3. For a 3D grid this translates as*

$$\text{threadIdx} = (\text{threadIdx.x}, \text{threadIdx.y}, \text{threadIdx.z})$$

A missing dimension assumes a value zero. Note that a threadIdx might uniquely identify a thread in a thread block but not a thread in the grid of the thread blocks.

Definition 6.14 (*ID of a thread in a grid*)**.** *Threads in a grid are uniquely identified by the pair*

$$(\textbf{blockIdx}, \textbf{threadIdx})$$

Note that there are three elements in variable **blockIdx** and three more elements in variable **threadIdx**.

Definition 6.15 (*Number of threads of a grid*)**.** *The number of threads of a grid is given by the product of the number of blocks and the number of threads per such block.*

$$\text{NumofThreads} = \text{gridDim.x} \times \text{gridDim.y} \times \text{gridDim.z} \times \text{blockDim.x} \times \text{blockDim.y} \times \text{blockDim.z}.$$

Implicit to this is the fact that all block of a grid have the same shape and thus contain the same number of threads.

Definition 6.16 (*Warp size*)**.** *The size of a warp is available through variable*

warpSize

and is of type int. It is hardware-based. By default the value is 32.

The value cannot be changed by a programmer. It has been set to 32 for a while and remains so for at least 10 years now (as of 2023).

Example 6.3.

```
dim3 grid(16,8);     /* grid.x=16  grid.y=8   grid.z =1 */
dim3 block(16,2,1);  /* block.x=16 block.y=2  block.z=1 */
/* Num of blocks = 16x8 = 128          */
/* Num of threads/block = 16x2=32      */
/* Num of threads/grid  = 32x128= 4096 */
```

Quoting NVIDIA's manuals the data type of dim3 is an integer vector type based on uint3 that is used to specify dimensions. When defining a variable of type dim3, any vector component left unspecified is initialized to 1. We summarize variable in Figure 6.4.

gridDim	:	to obtain grid size (number of instances of kernel, i.e. number of blocks)
gridDim	:	dimension of grid of blocks
blockDim	:	block size (number of threads within instances,blocks)
blockDim	:	dimension of thread block
args	:	pointer to GPU memory of arguments
blockIdx	:	index of a block in grid
threadIdx	:	index of a thread in thread block
warpSize	:	32 as of 2023 and hardware bound

Figure 6.4: Variables

6.2.4 Thread linearized identification

The unique identification of a thread in a grid through a pair of triplets (or if we flatten it, through a sextuplet) is cumbersome. Several times we need to map a thread ID into a scalar value, i.e. one single number.

The following assigns a row-oriented numbering to the threads of the grid for a 1D or 2D grid composed of 1D or 2D blocks, as applicable.

Definition 6.17 (*1D grid of 1D blocks*)**.** *In a 1D grid containing 1D blocks, a block is identified by* blockIdx.x*, and inside that block, a thread is identified by* threadIdx.x*. A unique threadID for each thread of the grid is then given as follows representing an integer between* 0 *and* NumofThreads $-$ 1*.*

$$\text{threadID} = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}.$$

In case we have a 2D block in the grid, we need to figure out how to assign IDs to the threads of a block. This involves (usually) a row-major assignment. The following assigns IDs to the threads of all preceding blocks of the 1D grid, before assigning IDs to the threads of the next block, row-major way.

Definition 6.18 (*1D grid of 2D blocks*)**.** *In a 1D grid containing 2D blocks, a block is identified by* blockIdx.x*, and inside that block, a thread is identified by* $(\text{threadIdx.x}, \text{threadIdx.y})$*. A unique threadID for each thread is then given as follows representing an integer between* 0 *and* $\text{NumofThreads} - 1$*.*

$$\text{threadID} = \text{blockIdx.x} \times \text{blockDim.x} \times \text{blockDim.y} + \text{threadIdx.y} \times \text{blockDim.x} + \text{threadIdx.x}.$$

The assignment become a bit more complex in the case of a 2D grid. The following assigns IDs to the blocks of the grid in row-major form. Within a 1D block ID are assigned to threads normally and serially.

Definition 6.19 (*2D grid of 1D blocks*)**.** *In a 2D grid containing 1D blocks, a block is identified by* blockIdx.x*,* blockIdx.y*, and inside that block, a thread is identified by* threadIdx.x*. A unique threadID for each thread is then given as follows representing an integer between* 0 *and* $\text{NumofThreads} - 1$*.*

$$\text{threadID} = \text{blockIdx.y} \times \text{gridDim.x} \times \text{blockDim.x} + \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}$$

The following assigns IDs to the threafs of all preceding row blocks of the 2D grid in row-major form (and inside a block in row-major form) before assigning IDs to the next row's block, in row-major form.

Definition 6.20 (*2D grid of 2D blocks*)**.** *In a 1D grid containing 2D blocks, a block is identified by* blockIdx.x*,* blockIdx.y*, and inside that block, a thread is identified by* $(\text{threadIdx.x}, \text{threadIdx.y})$*. A unique threadID for each thread is then given as follows representing an integer between* 0 *and* $\text{NumofThreads} - 1$*.*

$$\begin{aligned} \text{threadID} &= (\text{blockIdx.y} \times \text{gridDim.x} + \text{blockIdx.x}) \times \text{blockDim.x} \times \text{blockDim.y} \\ &+ \text{threadIdx.y} \times \text{blockDim.x} + \text{threadIdx.x}. \end{aligned}$$

Another way is to assing thread IDs to the first block row of the first grid row, in row-major form, thus viewing the grid as a

$$\mathit{gridDim.y} \cdot \mathit{blockDim.y} \times \mathit{gridDim.x} \cdot \mathit{blockDim.x}$$

matrix with $\text{gridDim.y} \times \text{blockDim.y}$ number of rows and $\text{gridDim.x} \times \text{blockDim.x}$ number of columns. Thus the first row of the first row block of the grid that contains $\text{gridDim.x} \times \text{blockDim.x}$ threads will be assigned IDs in turn and serially, then the next row of $\text{gridDim.x} \times \text{blockDim.x}$ threads spanning the first row block of thread blocks and their threads and so on.

Definition 6.21 (*Grid-global row-major 2D grid of 2D blocks*)**.** *In a 1D grid containing 2D blocks, a block is identified by* $\text{blockIdx.x}, \text{blockIdx.y}$*, and inside that block, a thread is identified by* $(\text{threadIdx.x}, \text{threadIdx.y})$*. A unique threadID for each thread is then given as follows representing an integer between* 0 *and* $\text{NumofThreads} - 1$*.*

$$\begin{aligned} \text{threadID} &= (\text{blockIdx.y} \times \text{blockDim.y} + \text{threadIdx.y}) \times \text{gridDim.x} \times \text{blockDim.x} \\ &+ \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}. \end{aligned}$$

Example 6.4. *In Figure 6.5 the layout of a* 2×2 *grid of blocks is shown. The ID of each block in the form* $(\text{blockIdx.x}, \text{blockIdx.y})$*, is shown. Each block is also identified by a number in row-major form:* $B0, \ldots, B3$*. For one specific block the threadIdx values are also shown in Figure 6.6. Each block is a 2D* 4×8 *block containing 32 threads in four rows of 8 threads per row.*

Example 6.5. *In Figure 6.7 the layout of the* 2×2 *grid of blocks utilized in Figure 6.5 and Figure 6.6 is shown.*

B0:(0,0)	B1:(1,0)
B2:(0,1)	B3:(1,1)

Figure 6.5: Grid of Blocks (a 2×2 grid)

B3:	(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)	(6,0)	(7,0)
	(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)	(7,1)
	(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)	(7,2)
	(0,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)	(7,3)

Figure 6.6: Thread blocks (4×8 thread block)

B0:	(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)	(6,0)	(7,0)	B1:	(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)	(6,0)	(7,0)
	(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)	(7,1)		(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)	(7,1)
	(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)	(7,2)		(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)	(7,2)
	(0,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)	(7,3)		(0,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)	(7,3)
B2:	(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)	(6,0)	(7,0)	B3:	(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)	(6,0)	(7,0)
	(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)	(7,1)		(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)	(7,1)
	(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)	(7,2)		(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)	(7,2)
	(0,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)	(7,3)		(0,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)	(7,3)

Figure 6.7: Labelling threads

Example 6.6. *In Figure 6.8 the layout of the 2×2 grid of blocks utilized in Figure 6.5 and Figure 6.6 is shown. Threads for blocks B0 (all threads), B1 (all threads), B2 (first thread row) are labelled according to the 2D grid of 2D block numbering scheme.*

Example 6.7. *In Figure 6.9 the layout of the 2×2 grid of blocks utilized in Figure 6.5 and Figure 6.6 is shown. Threads for blocks B0 (all threads), B1 (all threads), B2 (first thread row) are labelled according to the grid-global row-major 2D grid of 2D block numbering scheme.*

B0:	0	1	2	3	4	5	6	7	B1:	32	33	34	35	36	37	38	39
	8	9	10	11	12	13	14	15		40	41	42	43	44	45	46	47
	16	17	18	19	20	21	22	23		48	49	50	51	52	53	54	55
	23	25	26	27	28	29	30	31		56	57	58	59	60	61	62	63
B2:	64	65	66	67	68	69	70	71	B3:	(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)	(6,0)	(7,0)
	(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)	(7,1)		(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)	(7,1)
	(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)	(7,2)		(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)	(7,2)
	(0,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)	(7,3)		(0,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)	(7,3)

Figure 6.8: Labelling threads (2D grid of 2D blocks)

B0:	0	1	2	3	4	5	6	7	B1:	8	9	10	11	12	13	14	15
	16	17	18	19	20	21	22	23		24	25	26	27	28	29	30	31
	32	33	34	35	36	37	38	39		40	41	42	43	44	45	46	47
	48	49	50	51	52	53	54	55		56	57	58	59	60	61	62	63
B2:	64	65	66	67	68	69	70	71	B3:	(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)	(6,0)	(7,0)
	(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)	(7,1)		(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)	(7,1)
	(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)	(7,2)		(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)	(7,2)
	(0,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)	(7,3)		(0,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)	(7,3)

Figure 6.9: Labelling threads (Grid-blobal row-major)

6.2.5 Launch examples

Definition 6.22 (*Function prefixes*)**.** *In CUDA a function can accept a prefix. This is shown in Figure 6.10.*

Example 6.8. *The launch*

$$add <<< 4,1 >>> (a,b,c);$$

launches a 1D grid of 4 blocks, each block containing 1 thread.

Each block of the 1D grid has an ID: blockIdx.x gives the ID of a block in the 1D grid of blocks. The only thread of any one of the four blocks also has an ID. It is threadIdx.x and it is 0 for all four threads of the four blocks. It also indicates that a unique ID for a thread in this case is the pair $(\text{blockIdx.x}, \text{threadIdx.x})$.

Example 6.9. *The launch*

$$add <<< 1,4 >>> (a,b,c);$$

launches a 1D grid of 1 block, each block containing 4 threads.

Each block of the 1D grid has an ID: blockIdx.x gives the ID of a block in the 1D grid of blocks. The only block of the grid had blockIdx.x equal to 0. Each one of the four threads of the only grid block also has an ID. It is threadIdx.x and it is 0, 1, 2 or 3 for each one of the four threads of the block. It also indicates that a unique ID for a thread in this case is just threadIdx.x or also the pair $(\text{blockIdx.x}, \text{threadIdx.x})$ as before.

Example 6.10. *The launch*

$$\begin{aligned} dim3 \quad & nblocks(4,4); \\ dim3 \quad & nthrdperB(16,16); \\ add <<< & nblocks, nthrdperB >>> (a,b,c); \end{aligned}$$

launches a kernel grid containing 16 blocks in 4×4 *2D arrangement, and each block contains 256 threads also arranged in a* 16×16 *2D arrangement. Thus the total number of threads launched is* $16 \times 256 = 4096$*.*

Prefix	Execution	Callable
$\_\_global\_\_$	On device	Host or Device
$\_\_device\_\_$	On device	Device
$\_\_host\_\_$	On Host	Host (Default, can be omitted)

Figure 6.10: Function Prefixes

6.3 CUDA block execution

A kernel is executed by a grid of thread blocks. Threads form thread blocks, these thread blocks form a grid, and a grid launches and thus executes a kernel.

A Gigathread global scheduler distributes thread blocks to streaming multiprocessors (SMs). A block is thus a group of threads assigned and executed to a specific SM. Thus threads are assigned to SMs in the form of thread blocks rather than individual threads.

Execution-wise a block is split into warps: a single instruction in CUDA is issued for a group of a fixed number of threads at a time. This group of threads is called a warp. A warp is scheduled for execution by a warp scheduler. There can be one or more warp schedulers. Thus threads are assigned to cores in the form of warps (a scheduling unit of currently 32 threads).

Different blocks can be assigned to the same or different SM, can be executed simultaneously (multithreading) or one after another on the same or different SM. But thread blocks are required to execute independently. It must be possible that they can execute in any order. Thus an SM can execute multiple warps, interleaving their execution to hide memory stalls and thread blocks should be multiples of warp size for optimal performance.

When threads in a warp start execution at the same program address they use their own program counter(s) and are thus free to branch and execute independently. Threads in different SMs cannot send data one another or communicate with the shared memory: there is no guarantee that they will be executed on the same SM. The threads in a block are not (necessarily) executed concurrently but sequentially in warps (groups). The threads in a warp are executed in parallel on an SM. Each warp could consist of two half-warps or four quarter-warps.

If ”communication” is needed it might utilize L2 or the global (GPU's main) memory. All threads of all blocks of all SMs have access to the same global memory.

On the other hand, threads within a block can share information through the shared memory of the SM assigned to that block; the lifetime of shared memory is the lifetime of the block. Moreover threads within a block can synchronize themselves using function syncthreads, a lightweight barrier synchronization mechanism requiring approximately 128 cycles. Threads in different blocks or SMs cannot synchronize globally unless they use the global memory for explicit (programmer implemented) synchronization.

If one wants all threads of a set to be synchronized together, then one needs to assign them to a single block. This would limit significantly the SM utilization as one block is assigned to just one SM, and can not be assigned to multiple SMs.

Definition 6.23 (*Warp*)**.** *A warp is the group of threads of a thread block that executes a single instruction in CUDA at any time. Warp size is hardware dependent (and as of this writing in 2023, equal to 32 threads).*

ThreadID in a warp is consecutive and increasing.

Each thread of a block has its own memory (registers). What cannot fit into a limited number of registers becomes a spilling memory (similar to a victim's cache) and part of a private local memory (mapped global memory). All the threads of a block, during their lifetime, have access to limited shared memory (approximately 48KiB for Kepler architectures). Threads of a block have access to shared memory, can perform atomic operations and barrier style synchronization. Threads of different blocks do not interact.

All kernels (and their threads) share a Global memory; a read-only constant memory is also available for faster access. Texture read-only memory is available for graphics applications. An L2 cache is also available.

Whereas register and shared Memory are on the GPU chip, global, read-only constant and read-only texture memory are not; they persist kernel launches of the same application.

Example 6.11 (*Tesla warps*)**.** *At the physical execution level, in Tesla, a 32-thread warp is distributed on 8 (of the 192) cores of an SM with 4 threads assigned to each core. Over 4 clock cycles each of the 4 threads*

executes an instruction and thus the execution of one instruction for the warp of 32 threads gets completed as well.

Example 6.12 (*Multiple Warp Schedulers*)**.** *If more than one scheduler is available (Fermi has 2, Kepler has 4), more than one warps can be executed concurrently. Kepler has four warp scheduler and eight instruction dispatch units. Thus four warps can be executed concurrently with two instructions per warp can be dispatched each cycle.*

6.3.1 Block assignment on SM

A GPU has many SMs. A launched kernel grid has many blocks of threads. These blocks are distributed to several SMs for execution. Terefore several blocks can end up to (execute) on the same SM.

A (single) block is always scheduled to execute on ONLY one SM and stays there until the end of its execution.

A block of threads on a given SM can contain multiple threads. The SM cannot sustain the concurrent execution of an arbitrary number of threads because of limited hardware resources including cores and memory (registers or local memory). Each SM splits blocks assigned to it into one or more groups of threads by forming a warp. A warp contains consecutive threads of the split block. Warp size (as of 2023) is 32.

A warp scheduler picks the next warp to be executed on an SM. A warp scheduler is a local thread scheduler at SM level compared to the Gigathread global scheduler.

Warp size can vary but it has been a fixed number and equal to 32 so far (2023) and is hardware imposed.

All threads in a warp execute the same instruction at the same time (almost) on different data. Threads in a warp start at the same (machine instruction) address; individual threads can have varying behavior. Each thread has its own PC (program counter also known as instruction pointer), register file and register state. It can have its own independent execution path that can cause problems to the threads of a warp of a block. Threads can synchronize within a block; no inter block synchronization can (currently) occur.

Moreover, the warps of a block can be scheduled arbitrarily. No assumption can be made that they will execute in a specific order.

The number of active warps is limited by SM resources. In addition to an active Warp, a Waiting (or Stalled) Warp is possible. See below for more information.

Also no assumption can be made about the order of the execution of the threads of a block.

6.3.2 SM, warps, blocks and threads

Instruction execution latency for math operations is 4-6 clock cycles; they can thus be hidden if 4 to 8 to 16 warps are executing and thus hiding these latencies. Shared memory / L1 size can be adjusted.

Register access is zero cycle per instruction. Read after write however can be penalized from 16 to 24 cycles. Shared memory access can be as fast as register access as long as there is no contention among threads of a block. Shared memory utilizes __shared__ in variable definition. Constant __constant__ works if a warp of threads reads the same memory.

Effect of warpSize

The warpSize is fixed to 32 and several times determines or affects optimality in scheduling threads in thread blocks of a grid.

The maximum number of active warps in an SM is 64 and thus there can be no more than $64 \times 32 = 2048$ threads in an SM.

Moreover in an SM no more than 16 blocks can be active. The maximum number of threads per block is 32 warps i.e. 1024 threads (Kepler K20X) but used to be 512 (in earlier versions of NVIDIA chips). The

Max	NoofKernels	per device	32	CUDA 5.0
Max	NoofThreads	per block	1024	RULE-2
Max	NoofThreads	per block	1536	some recent archs.
Max	NoofBlocks	per SM	16 or 32	RULE-1
Max	NoofWarps	per SM	64	
Max	NoofThreads	per SM	2048	RULE-3
Max	NoofRegister	per SM	65536	(32-bit)
Max	NoofRegister	per block	65536	
Max	NoofRegister	per Thread	255	
Max	dimgrid (x,y,z)		$=(2^{31}-1,65535,65535)$	
Max	dimblock(x,y,z)		$=(1024,1024,64)$	
Max	Shmem	per SM	48KiB	
Max	Shmem	per SM	164KiB	8.0 (Ampere)
Max	Shmem	per SM	100KiB	8.6
Max	Shmem	per SM	96KiB	Volta
Max	Shmem	per block	100KiB	8.0 (Ampere)
Max	Shmem	per block	99KiB	8.6
Max	Shmem	per block	96KiB	
Max	WarpSize	per device	32	
Num	WarpSchedulers	per SM	4	

Figure 6.11: Restrictions

number of 32-bit registers per SM is 65536 but the maximum number of registers per thread is around 255 (and rising to 1023 for some new NVIDIA chips).

For example, If a thread block contains one thread, it is still going to form, during execution, a warp of size 32. One thread of the 32-thread warp will compute, and 31 of the remaining threads of the warp will do nothing. This is inefficient.

If a block contains 33 threads, it is still going to form two warps of size 32. 31 of the 32 threads of the second warp will do nothing. This is also inefficient. It is advisable that a block contains a multiple of 32 number of threads (or more accurately a multiple of warpSize number of threads).

The number of warps per block is given obviously by the following formula.

$$nWarps = ceil(ThreadsPerBlock/warpSize),$$

where warpSize is currently equal to 32.

On Ampere architectures, a typical block can contain 4 warps of 128 threads total. One SM is assigned 8-12 blocks for a total of approximately 2000 threads.

If the threads of a block are arranged in a 1D shape, then 32 consecutive threads form a warp. The assignment of threads to a warp will use the linearized 1D threadID threadID = threadIdx.x. If however the threads of a block are arranged in a 2D shape, then the threads are virtually mapped into identifiers of a 1D shape using the linearized 2D threadIDs,

$$\text{threadID} = \text{threadIdx.y} * \text{blockDim.x} + \text{threadIdx.x},$$

and the virtual threadID is used to form the 32 consecutive threads that will form a warp. For 3D shapes, the linearized 3D threadID,

$$\text{threadID} = \text{threadIdx.z} \times \text{blockDim.x} \times \text{blockDim.y} + \text{threadIdx.y} \times \text{blockDim.x} + \text{threadIdx.x},$$

will then be used.

Variable warpSize is hardware-bound and currently fixed to 32. However there are multiple restrictions imposed by the GPU and its SMs on how many threads or blocks there can be in a system.

SM restrictions

Occupancy issues can restrict further the possible geometries. For a example an SM can have a maximum of 65536 (32-bit) registers. A thread can have a maximum of 255 registers. If the threads of a block use the maximum number of registers (let us call them thick threads) we can have no more than 256 of them in a block. 1 If however they use only 16 registers, we can not have 4096 threads in a block because the limit (for some GPUs) is 1024 threads per block. Likewise we cannot have more than 2048 threads per SM either. Thus block occupancy restrictions and SM restrictions can affect performance.

The number of block maintained by an SM should be kept below its maximum (16 or 32). Given that the maximum number of resident threads per SM is 2048 this means than a block should have no more than 128 threads. An SM has 64 maximum resident warps. This is consistent with $2048/32 = 64$, the thread occupancy of an SM (2048 threads) and warpSize (32).

Each SM unit has 4 warp schedulers. Each warp scheduler can issue one or two instructions as long as they are destined for different functional units (e.g. one to the floating-point unit, and the other to the load-store or the SFU) or are output independent. For example $a = b + c, d = e + c$ are output independent, but $a = b + c, d = e + a$ are not.

Note also that among the 192 cores per SM (Kepler), the fact that there are only 4 warp schedulers limits the number of active threads to $4 \times 32 = 128$. This means that $192 - 128 = 64$ cores are not utilized unless there is a memory stall in the 128 cores. Because of possibly this consideration the number of cores per SM (or SMM) in the Maxwell architecture has dropped to 128 from 192. A code such as the one below executed among the threads of a block would require time which would be the maximum of a() + max $(b())$ + $max(c())$ + d() . It would be better to split into a separate block the threads that execute $b()$ from those executing $c()$.

Moreover if the number of blocks per grid is say 256, and an SM can accommodate 16 blocks, we need a GPU with 16 SM; a K20X has only 14!

As of this writing NVIDIA CUDA has the set of restrictions as described in Figure 6.11.

In order to determine the number of threads per block one might have to satisfy several conditions. It is the minimum as defined below:

$$
\begin{aligned}
\min\{ & \\
& \mathbf{nWarpSchedulers} * \mathbf{WarpSize}, \\
& \mathbf{MaxThreadsPerSM} \\
& \mathbf{SzSharedM}/\mathbf{SharedMPerThread}, \\
& \mathbf{RegsPerBlock}/\mathbf{RegsPerThread}, \\
\} &
\end{aligned}
$$

6.3.3 Warp divergence and stall

Warp divergence is resolved by having all threads executing both branches of say an if-then-else statement with a cost of execution at least the sum of the costs of the branches.

Example 6.13. *Let a kernel instruction be an if-then-else statement such as the one of Figure 6.12. Let for the sake of this example WarpSize is a non-standard 8, and let threads 0,4,5,6 execute the ifstep and threads 1,2,3,7 execute the elsestep of the if-then-else statement. Such an executions, breaks the essence of SIMT that dictates that the threads of a block must execute the same instruction at the same time. Assuming a D means*

```
if (x) {
  ifstep
}
else   {
  elsestep
}
```

Figure 6.12: An if-else statement

	Thread	0	1	2	3	4	5	6	7
x :	ifstep	D	S	S	S	D	D	D	S
!x :	elsestep	S	D	D	D	S	S	S	D

Figure 6.13: Branches

DO and an S means STALL (or DISABLED) the overview of the activity of the 8 threads over the if-then-else statement would look like as follows in Figure 6.13.

The syntax of the last two lines of Figure 6.13 is that of a predication. If x evaluates to true, the ifstep is executed, otherwise the elsestep does.

By default in NVIDIA CPUs each one of the threads would execute both the ifstep and the elsestep part. They will just stall (get disabled) over the part that is of no consequence. (Threads 1,2,3,7 will stall over the ifstep part.) Thus all 8 threads would first evaluate the condition predicate x, and based on the evaluation of predicate x half will stall and half will execute the ifstep, and then likewise half will execute and half will stall over the elsestep.

Thus the execution time of the if-then-else would be the sum of the times of the ifstep and elsestep plus the stall/DISABLE overhead, if any.

NVIDIA GPUs use branch predication to avoid stalling. It works only for small ifstep and elsestep executions.

In a warp maximum efficiency is extracted if all threads of a warp have identical execution path (e.g. they all perform the "if" part). If there is a divergence of paths, both the "if" and then the "else" parts get executed and disabled as needed to resume a common execution path at the end. Thus within a warp we can have **active** and **inactive** threads.

Furthermore, consider the extreme scenario where a block is split into 32 warps of 32 threads per warp. Only one thread per warp will execute the elsestep and the kernel contains only an if-then-else statement. Moreover the ifstep contains two instructions and the elsestep contains 20 instructions. Assuming an SM executes one warp at a time, the 32 warps will take $32 \times (20+2) = 704$ instructions. Now consider rewriting the kernel so that all the elsestep threads are relegated to a single warp. The running time now would be $31 \times 2 + 22 = 86$ instructions total, an 8-fold improvement in performance.

Thus with divergence CUDA gives the illusion of an MIMD architecture even if it is inherently SIMD/SIMT.

6.3.4 Warp scheduler

The warp scheduler(s) at every instruction iteration selects a warp that is ready to execute and issues the instruction to its active threads. Latency is defined as the time it takes for a warp to be ready to execute. A warp might not be ready because its instructions have not received their input operands yet. Off chip memory take 200-400 cycles to be delivered. Thus to hide a latency of l clock cycles we need l cycles for devices with 2.0 compute capability as an SM issue one instruction per warp over two cycles for warps at a time, $2l$ for 2.1 compute capability as an SM issues a pair of instruction per warp over two cycles for two warps at a time,

and 8l for 3.0 as a multiprocessor issues a pair of instructions per warp over one clock cycle for 4 warps at a time.

6.4 Memory

In a CUDA GPU, every thread has its own "local" memory; it is referred to as registers. According to Figure 6.11 the maximum number of registers per thread is 255. Moreover all the registers used by all threads of a block cannot exceed 65536. This is also the number of registers supported by all blocks assigned to an SM.

The threads of a block can cooperate using block-level synchronization and block-local shared memory. Threads from different blocks CANNOT cooperate.

What cannot fit into a limited number of registers becomes part of a spilling local memory (mapped global memory cached in L1), similar to a victim's cache.

All the threads of a block have access to limited shared memory (approximately 48KiB for Kepler architectures).

All kernels (and of course all the threads of their thread blocks) share a Global memory; a read-only constant memory is also available for faster access. Texture read-only memory is available for graphics applications. An L2 cache is also available.

Whereas registers and shared Memory are on the GPU chip, global, read-only constant and read-only texture memory are not; they persist kernel launches of the same application.

Global memory

Writes into a common global memory location by more than one threads of a warp are undefined for non-atomic instructions; for atomic instructions serialization occurs but its order is also undefined. Global memory is 100x slower than L1/shared memory.

Functions malloc and free allocate and release memory in the host's heap. Functions cudaMalloc and cudaFree do the same with device global memory. Function cudaMemcpy is used for the transfer of memory From/To Host To/From the device.

Global memory instructions support read/writes of 1/2/4/8/16B.

"Coalesced" access occurs when the 32 threads of a warp access adjacent memory locations, properly aligned.

"Uncoalesced" access occurs when the memory locations have gaps (i.e. stride is greater than 1).

Since the PASCAL GPU, the use of a unified cudaMallocManaged and cudaDevicesSynchronized eliminated the need of cudaMemcpy as memory shared by host AND device (global memory).

Shared memory

Shared memory is organized in 32 banks equal to the number of threads in a warp. Banks can be accessed simultaneously.

Thus 32 read instructions on 32 addresses of 32 different banks can be served in one step. If they are to the same step, serialization takes place i.e. 32 steps are required.

Type	Location	Access	Scope;Lifetime
Register	on-chip	RW	thread;thread
Local	off-chip	RW	thread;thread
Shared	on-chip	RW	block; block
Global	off-chip	RW	grid; host
Constant	off-chip	R	grid; host

Figure 6.14: Memory type and lifetime

6.5 Floating-point performance

In modern GPUs double (DP) floating-point performance (64-bit) is half the rate of float (FP) floating-point perfomance (32-bit).

In a older generation multiprocessor (Kepler) only 64 FPU-DP units existed and 192 FPU-SP units. That meant that the DP rate was one-third of the SP rate. However a mix of SP and DP operations in the code can sustain higher performance than the DP implied rate. Moreover SFU units are available for transcendental operations (sin, cos, sqrt, etc).

6.5.1 SFU operations and intrinsic functions

$\_\_fdivide(x,y)$	faster than /
rsqrtf(x)	instead of $1.0/sqrt(x)$
$\_\_fadd$	Addition float
$\_\_fsub$	Subtraction float
$\_\_fmul$	Multiplication float
$\_\_fsqrt$	Square root float
$\_\_frsqrt$	Reciprocal square root
$\_\_fmaf$	Multiply and add
$\_\_frcp$	Recirprocal
$\_\_fdiv$	Division float
$\_\_fdividef$	
$\_\_sinf$	Trigonometrix (sin)
$\_\_cosf$	
$\_\_tanf$	
$\_\_sincosf$	sin(cos(.))
$\_\_logf$	Logarithmic float
$\_\_log2f$	Log base two
$\_\_log10f$	Log base ten
$\_\_expf$	Exponential
$\_\_exp10f$	10 to power
$\_\_powf$	General power x**y
$\_\_dadd$	Double addition
$\_\_dsub$	Double subtract
$\_\_dmul$	Double multiply
$\_\_dma$	Double multiply and add
$\_\_ddiv$	Double divide
$\_\_drcp$	Double reciprocal
$\_\_dsqrt$	Double square root

Figure 6.15: SFU Intrinsic functions

Several of the logarithmic and trigonometric functions are three times faster for floats than doubles. Suffixes such as $\_\_rn, \_\_rz, \_\_ru, \_\_rd$ means “round nearest even”, “round towards zero”, “round up”, “round down”.

Chapter 7

NVIDIA CUDA

7.1 Overview

The CUDA model abstraction uses SIMD processors and SIMT execution: massive parallelism is applied to thousands of threads sharing (or not) data at various levels. GPU computing uses fine-grained parallelism and is data intensive in contrast to CPU computing that is more coarse-grained and program control is (or can be) more complicated.

A grid is a logical organization of thread blocks. Variable gridDim provides information about the dimensionality of the grid: thus the number of non-one values among gridDim.x, gridDim.y and gridDim.z determine whether the thread blocks of the grid are structured in a 1D, 2D or 3D shape.

The x dimension grows "horizontally", the y dimension grows "vertically" , and the z dimension grows "depth-wise".

The product of gridDim.x, gridDim.y and gridDim.z determines the number of thread blocks in the grid. (A thread block is also known as CUDA block; from now on we will call it a block droping the thread in front of it.)

Example 7.1. *A* $(5,3)$ *grid implies it is a 2D grid and thus* gridDim.x $= 5$*,* gridDim.y $= 3$ *and* gridDim.z $= 1$ *(implied). The total number of blocks of the grid is* $5 \times 3 = 15$*.*

In matrix notation we would denote a $(5,3)$ grid as an 3×5 matrix of blocks. The x dimension grows "horizontally", along the columns of a matrix, y dimension grows "vertically", along the rows of a matrix, and z dimension grows "depth-wise".

Example 7.2. *A* $(16,32)$ *block implies it is a 2D aggregation of threads and thus* blockDim.x $= 16$*,* blockDim.y $= 32$ *and* blockDim.z $= 1$ *(implied). The total number of threads of the block (and more importantly, of any block of the grid) is* $16 \times 32 = 512$*.*

The location of a block within a grid is retrieved through variable blockIdx. Its location is then blockIdx $=$ (blockIdx.x, blockIdx.y, blockIdx.z)

Example 7.3. *Thus for a (5,3) grid, the grid of the previous example,*

$$\text{blockIdx} = (\text{blockIdx.x} = 2, \text{blockIdx.y} = 1, \text{blockIdx.z} = 0)$$

identifies the middle block of threads.

The location of a thread within a block is retrieved through variable threadIdx. Its location is then threadIdx $=$ (threadIdx.x, threadIdx.y, threadIdx.z) .

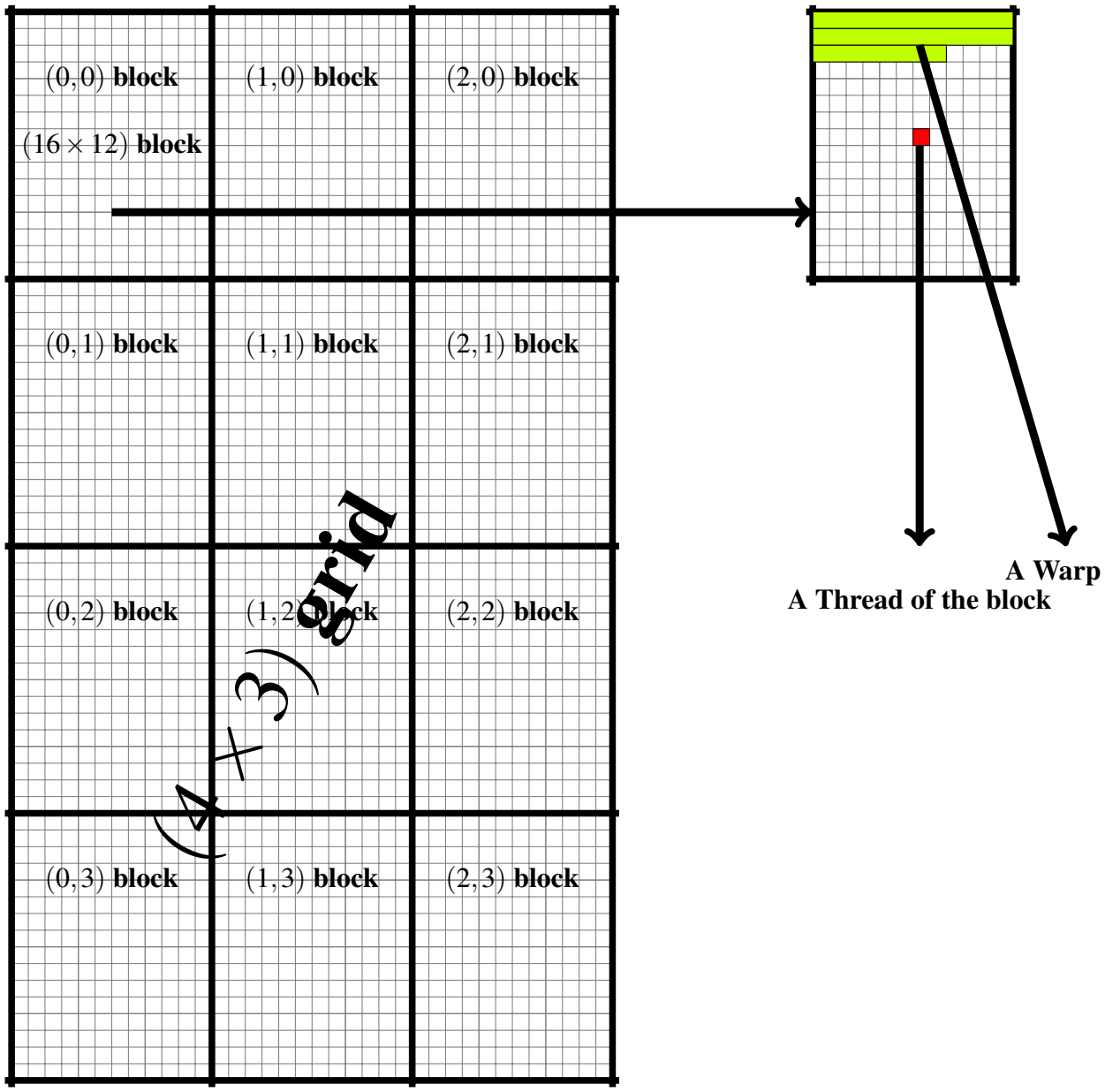


Figure 7.1: Grids, Blocks, Threads and a Warp

Example 7.4. *Figure 7.1 shows a* $(3,4)$ *grid, consisting of 12 blocks. Each block is a* $(12,16)$ *thread block.*

7.2 Execution at the host

Let us see how a HelloWorld program can be written for CUDA. A CUDA program resides in a file ending with the suffix `.cu` and other than that it can be compiled and executed as a regular C program but one needs to the Nvidia C compiler known as `nvcc`. For several activities it serves as a wrapper.

Figure 7.3 looks like a traditional C program. It can be compiled by a traditional C compiler `gcc` or `nvcc`. When the generated executable file (a.out) loads for execution, tt is executed at the host. NVIDIA's CUDA would use only host compiler infrastructure to compile and then the code is to run only on the host!

Figure 7.4 is a variation. It still executes on the host the printf line but NVIDIA's CUDA would use not only the host compiler infrastructure but the NVIDIA's one as well since the executable will execute a piece of code on the device (NVIDIA GPU). The function (kernel) that will be executed is a no code function. The kernel mykern is laynched on a grid consisting one one block; the block contains only one thread.

A traditional C program is an acceptable NVIDIA CUDA program. It executes on the host. See Figure 7.2.

```
1 #include <stdio.h>          /* c7c00.cu */
2 int main(void){             /* stored in c7c00.cu         */
3  printf("Hello world!\n"); /* module load cuda/11.0.2    */
4  return(0);                /* compile with nvcc c7c00.cu */
5 }                           /* run with ./a.out           */
```

Figure 7.2: Hello world in C

```
1 #include <stdio.h>              /* c7c01.cu */
2   int main(void) {              /* % module load cuda/11.0.2 */
3    printf("Hello world\n");     /* %  nvcc c7c01.cu          */
4    return(0);                   /* %  ./a.out                */
5   }                             /*   Hello world             */
```

Figure 7.3: Hello world in C (host launch and execution)

```
1 #include <stdio.h>              /* c7c02.cu */
2 #include <cuda_runtime.h>
3 __global__ void mykern (void) {
4            /* empty */ ;
5  }
6   int main(void) {              /* % module load cuda/11.0.2 */
7    mykern<<< 1, 1 >>>();        /* %  nvcc c7c02.cu          */
8    printf("Hello world\n");     /* %  ./a.out                */
9    return(0);                   /*   Hello world             */
10  }
```

Figure 7.4: Hello world in C (host launch and execution, device usage)

7.3 Execution at the device

We need to move to the code of Figure 7.5 to be able to execute on the device. Compilation is similar to the previous pieces of code; instructions are thus omitted. If we have information about the GPU architecture we might include an architecture directive such as arch = sm_30, as needed.

```
#include <stdio.h>      /* c7c03.cu */
#include <cuda.h>
#include <cuda_runtime.h>
__global__ void mykern (void) {
    printf("Hello world : (%d , %d )\n",
          blockIdx.x,threadIdx.x);
 }
  int main(void) {
    mykern<<< 1, 4 >>>();
    cudaDeviceSynchronize();
    return(0);
  }
```

Figure 7.5: Hello world in C (host launch, device execution)

Figure 7.6 for a change defines an $(1,2)$ grid. Each of the two blocks of the grid has 8 threads arranges as a $(2,4)$ thread block. Identical information about the shape of the grid will be printed by each one of its 16 threads. Likewise about the shape of a block. Identical BlockIdx information will be printed by each one of the 8 threads of each of the two blocks. We need to reach the output of threadIdx to be able to get unique information per thread.

```
#include <stdio.h>  /* c7c04.cu */
#include <cuda.h>
#include <cuda_runtime.h>
__global__ void newkern (void) {
    printf("Grid : %d %d %d\n",gridDim.x, gridDim.y,gridDim.z);
    printf("Block: %d %d %d\n",blockDim.x, blockDim.y,blockDim.z);
    printf("BlockIdx  %d %d\n",blockIdx.x, blockIdx.y);
    printf("ThreadIdx %d %d\n",threadIdx.x, threadIdx.y);
 }
  int main(void) {
    dim3 grid(1,2);
    dim3 blck(2,4);
    newkern<<< grid, blck >>>();
    cudaDeviceSynchronize();
    return(0);
  }
```

Figure 7.6: Grid and Block information

We can reduce the output by a factor of one quarter if we decide that only threads with threadIdx.y $= 2$ will print something. THis is Figure 7.7.

Let us provide some information about the structure of a CUDA program. The kernel is defined by using the prefix `__global__`. It is callable on the device or the host, but execution always takes place on the device. If it is omitted, a `__host__` would then be implied that is callable on the host and execution takes place on the host. Thus Figure 7.8 would generate a compilation error because kernel newkern cannot be launched, because it is not a kernel, but a host function (a regular C function) though its contents and never attempted

```
#include <stdio.h>  /* c7c05.cu */
#include <cuda.h>
#include <cuda_runtime.h>
__global__ void newkern (int tidy) {
if (tidy== threadIdx.y){
    printf("Grid : %d %d %d\n",gridDim.x, gridDim.y,gridDim.z);
    printf("Block: %d %d %d\n",blockDim.x, blockDim.y,blockDim.z);
    printf("BlockIdx  %d %d\n",blockIdx.x, blockIdx.y);
    printf("ThreadIdx %d %d\n",threadIdx.x, threadIdx.y);
}
 }
  int main(void) {
    dim3 grid(1,2);
    dim3 blck(2,4);
    newkern<<< grid, blck >>>(2);
    cudaDeviceSynchronize();
    return(0);
  }
```

Figure 7.7: Argument passing (host to device)

launch might indicate otherwise. Nor using the prefix `__device__` would change anything. Note that there are two underscores preceding and two following the global, host and device keywords.

```
#include <stdio.h>  /* c7c06.cu */
#include <cuda.h>
#include <cuda_runtime.h>
           void newkern (int tidy) {
if (tidy== threadIdx.y){
    /* stripped down */
    printf("ThreadIdx %d %d\n",threadIdx.x, threadIdx.y);
}
 }
  int main(void) {
    dim3 grid(1,2),blck(2,4); /* condensed */
    newkern<<<grid, blck >>>(2);
    cudaDeviceSynchronize();
    return(0);
  }
```

Figure 7.8: Erroneous prefix usage for kernel (implied host))

The following refers to Figure 7.6 even if it applies to other compilable code e.g. Figure 7.5, and Figure 7.7. The code to be run on the device is known as the **kernel** and is the unit of work that is offloaded by the host to be executed on the device. This code is executed in parallel by all threads. The function `newkern` gets launched by the host and executed in the device. The launch (function invocation) of the function from the host is the third line of `main`. The first line of main defines a grid named `grid`. It is a $(1,2)$ grid of (two) blocks. The second line of main defines the shape of a thread block as being $(2,4)$ thus establishing 8 threads in a block and 16 threads for the grid. The launched kernel would thus be executed by 16 threads in total belonging to two blocks. In the simple launch no arguments are pass and this explains the empty set of parentheses in the third line of main. Figure 7.7 has an example of parameter passing from the host to the device. Thus a successful and well-designed launch must include

- the shape of the grid of blocks,

```
#include <stdio.h>   /* c7c07.cu */
#include <cuda.h>
#include <cuda_runtime.h>
__device__ void newkern (int tidy) {
if (tidy== threadIdx.y){
    /* stripped down */
    printf("ThreadIdx %d %d\n",threadIdx.x, threadIdx.y);
}
 }
  int main(void) {
    dim3 grid(1,2),blck(2,4); /* condensed */
    newkern<<<grid, blck >>>(2);
    cudaDeviceSynchronize();
    return(0);
  }
```

Figure 7.9: Erroneous prefix usage for kernel (device))

- the shape of each block of threads of the grid,
- the kernel prefixed with `__global__` , and
- appropriate parameters passed from host to device.

Code that is **called and run** on the device only can have the CUDA C keyword `__device__`. This is not the case for the launched kernel that is called on the host. This explains the compilation error of Figure 7.8 (to be called on host, to run on host, but expected to launch a kernel that should run on the device) and also of Figure 7.9 (to be called on device, to run on device, but expected to launch a kernel that should be called on host)!

It is the responsibility of the `nvcc` compiler to separate host and device code and device code is then processed by the NVIDIA compiler but host code by the host compiler (eg. gcc). This separation is impossible for Figure 7.8 and Figure 7.9. Moreover the number of threads, grid geometry and block geometry that will be used, is derived from information used in the launch invocation.

The simple launch used in Figure 7.4 and Figure 7.5 that does not use the dim3 data type directly, generates a 1D grid and 1D block(s). We need the setup of Figure 7.6 to be able to generate a 2D grid and 2D blocks. This setup can be used to generate 3D grids and 3D blocks, or mix 1D, or 2D, or 3D grids with 1D, or 2D, or 3D blocks.

A kernel launch in addition to `grid` and `blck` as the first and second argument can have two additional arguments. Thus $\lll grid, blck, N, S \ggg$ defines a grid `grid` with block shape and geometry `blck` but allocated N byte of shared memory (N must of type `size_t`) that is to be dynamically allocated to each block (default is 0 bytes), and also generates a number S of streams (default is also a 0).

An application can also launch by altering bounds that are specified using `__launch_bounds__`. The latter accepts as a first argument maxThreadsPerBlock and a second argument MinBlocksPerMultiprocessor. Observe Figure 7.10

```
#include <stdio.h>  /* c7c08.cu */
#include <cuda.h>
#include <cuda_runtime.h>
__global__ void  __launch_bounds__ (8,2) newkern (int tidy) {
if (tidy== threadIdx.y){
    printf("Grid : %d %d %d\n",gridDim.x, gridDim.y,gridDim.z);
    printf("Block: %d %d %d\n",blockDim.x, blockDim.y,blockDim.z);
    printf("BlockIdx  %d %d\n",blockIdx.x, blockIdx.y);
    printf("ThreadIdx %d %d\n",threadIdx.x, threadIdx.y);
}
}
  int main(void) {
    dim3 grid(1,2);
    dim3 blck(2,4);
    newkern<<< grid, blck >>>(2);
    cudaDeviceSynchronize();
    return(0);
  }
```

Figure 7.10: Launch bounds __launch_bounds__

7.4 Configuration information

A `cudaGetDeviceCount` can provide information about the number of devices available and `cudaGetDeviceProperties` about their properties. The output of the execution of the code in Figure 7.12 is shown in Figure 7.13 and it is the platform of the benchmark experiments run on the following chapters.

```
struct cudaDeviceProp {   /* c7c09.cuh */
  char name[256];                 // Device Name
  size_t totalGlobalMem;          // Global Device Memory in B
  size_t sharedMemPerBlock;       // Max shared memory per block
  int regsPerBlock;               // 32-bit registers per block
  int warpSize;                   // number of threads in a warp
  size_t memPitch;                // max pitch for memory copies in B
  int maxThreadsPerBlock;
  int maxThreadsDim[3];
  int maxGridSize[3];
  size_t totalConstMem;
  int major;                      // major  revision number
  int minor;                      // minor  revision number
  int clockRate;
  size_t textureAlignment;
  int deviceOverlap;              // cudaMemcpy and kernel exec ovrlp
  int multiProcessorCount;        // SM in device
  int kernelExecTimeoutEnabled;
  int integrated;
  int canMapHostMemory;
  int computeMode;
  int concurrentKernels;
  int ECCEnabled;
  int pciBusID;
  int pciDeviceID;
  int tccDriver;
 }

/* For information Purposes re c7c09.cu */
```

Figure 7.11: cudaDeviceProp

```
#include <stdio.h>  /* c7c09.cu */
#include <cuda.h>
#include <cuda_runtime.h>
  int main(void) {
   int i,count;
   cudaGetDeviceCount(&count);
   cudaDeviceProp  prop;
   memset(&prop, 0, sizeof(cudaDeviceProp));
   printf("Number of Devices is %d\n",count);
   for (i=0;i<count;i++) {
       cudaGetDeviceProperties(&prop,i);
       printf("Device Number is %d \n",i);
       printf("Device Name   is %s \n",prop.name);
       printf("Total Global M : %d \n",prop.totalGlobalMem);
       printf("Shared Mem/Block %d \n",prop.sharedMemPerBlock);
       printf("Registers /Block %d \n",prop.regsPerBlock);
       printf("WarpSize         %d \n",prop.warpSize);
       printf("memPitch         %d \n",prop.memPitch);
       printf("maxThreads/Block %d \n",prop.maxThreadsPerBlock);
       printf("maxThreads Dim   %d %d %d \n",prop.maxGridSize[0],
                                             prop.maxGridSize[1],
                                             prop.maxGridSize[2]);
       printf("TotConst  Memory %d \n",prop.totalConstMem);
       printf("Major Revision   %d \n",prop.major);
       printf("Minor Revision   %d \n",prop.minor);
       printf("Clock Rate       %d \n",prop.clockRate);
       printf("deviceOverlap    %d \n",prop.deviceOverlap);
       printf("SM in device     %d \n",prop.multiProcessorCount);
       printf("Compute  Mode    %d \n",prop.computeMode);
       printf("Concurrent Kernl %d \n",prop.concurrentKernels);
  }
    cudaDeviceSynchronize();
    return(0);
  }
```

Figure 7.12: cudaGetDeviceProperties

```
Number of Devices is 1
Device Number is 0
Device Name   is Quadro P600
Total Global M : 2095841280
Shared Mem/Block 49152
Registers /Block 65536
WarpSize         32
memPitch         2147483647
maxThreads/Block 1024
maxThreads Dim   2147483647 65535 65535
TotConst  Memory 65536
Major Revision   6
Minor Revision   1
Clock Rate       1556500
deviceOverlap    1
SM in device     3
Compute  Mode    0
Concurrent Kernl 1
```

Figure 7.13: Configuration of CUDA platform

Chapter 8

Vector calculations

8.1 Step-wise CUDA programming: Program 1 (c8c01.cu)

We provide a first program that shows how data can be transferred from host to device and the other way around. As mentioned earlier this can be circumvented by newer methods that allow sharing of info between host and device. With reference to Program 1 of Figure 8.1 we observe the following.

In line 17 an array a is allocated on the host (heap). The C standard libary function malloc is invoked. It is initialized with data in lines 17-21.

In line 22 an array da is allocated on the device. The equivalent to malloc is invoked. This is cudaMalloc that has malloc's syntax. In line 23, with cudaMemcpy data is transferred to the destination (first argument) from the source (second argument), the size of data is indicated as the third argument, with a fourth argument to cudaMemcpy being the direction of the transfer (from Host to Device). Thus this argument indicates that the second argument is located on the host, and the first argument on the device.

The cudaDeviceSynchronize() are there to make sure that GPU activity has been completed, prior to the data transfer from the device to the host. Extra cautiouly a second cudaDeviceSynchronize() is inserted in line 28 prior to the printing.

By inspecting the kernel and look at the output of the execution we observe the following. The grid used is a 1D with 2 blocks. Each block is 1D with 8 threads. Thus the total number of threads in the grid is 16. So are the number of elements of a and da. The tid of line 7 is the threadID of 1D grid of 1D blocks derived earlier in Chapter 6. Thus tid has a range from 0 to 15. The printing of lines 9-11 confirm this. Finally the result is copied from device to host and output in lines 29-31. It is a good practice to deallocate memory in the reverse order of allocation.

```
#include <stdio.h>        /* c8c01.cu */
#include <cuda.h>
#include <cuda_runtime.h>
#define FTYPE float

__global__ void c8c01 (FTYPE *da) {
   int tid = blockIdx.x * blockDim.x +threadIdx.x;
   da[tid] = (FTYPE) tid;
   printf("blockIdx.x = %3d threadIdx.x = %3d tid= %3d da= %8f\n",
           blockIdx.x,      threadIdx.x,      tid,     da[tid] );
}
int main(void) {
 FTYPE *a, *da;
 int   i , grid=2,  blck=8;
 int   nnum, nsze;
 nnum=grid*blck; nsze=nnum*sizeof(FTYPE);
 a=(FTYPE*) malloc(nsze);
 for (i=0; i< nnum ; i++) {
   a[i]=nnum-(FTYPE)i;
   printf("a[%3d] = %8f\n",i,a[i]);
 }
 cudaMalloc((void**)&da,nsze);
 cudaMemcpy(da,a,nsze,cudaMemcpyHostToDevice);

  c8c01<<<grid,blck>>>(da);
 cudaDeviceSynchronize();
 cudaMemcpy(a,da,nsze,cudaMemcpyDeviceToHost);
 cudaDeviceSynchronize();
 for (i=0; i< nnum ; i++) {
   printf("a[%3d] = %8f\n",i,a[i]);
 }
 cudaFree(da);free((void*)a);
}
```

Figure 8.1: Program 1: Host - Device data echange

8.2 Program 2: c8c02.cu

Figure 8.2 shows the same functionality but by using cudaMallocManaged which has the arguments of malloc, and avoids the use of cudaMemcpy that cudaMalloc needs.

The use of cudaMalloc requires the following steps in CUDA programming. The use of cudaMallocManaged combines into one Steps 1 and 2, eliminates 3,4,5 or just maintains 4, and also eliminates step 7.

1. Setup input memory on host (CPU memory),
2. allocate input memory on device (GPU memory),
3. copy input memory from host to device,
4. allocate output on host,
5. allocate output on device,
6. compute on device after launching one or more GPU kernels from the host,
7. copy output from device to host, and
8. free all memory.

```
#include <stdio.h>        /* c8c02.cu */
#include <cuda.h>
#include <cuda_runtime.h>
#define FTYPE float

__global__ void c8c01 (FTYPE *da) {
   int tid = blockIdx.x * blockDim.x +threadIdx.x;
   da[tid] = (FTYPE) tid;
   printf("blockIdx.x = %3d threadIdx.x = %3d tid= %3d da= %8f\n",
           blockIdx.x,      threadIdx.x,      tid,     da[tid] );
}
int main(void) {
 FTYPE *a;
 int   i , grid=2,  blck=8;
 int   nnum, nsze;
 nnum=grid*blck; nsze=nnum*sizeof(FTYPE);
 cudaMallocManaged(&a,nsze);
 for (i=0; i< nnum ; i++) {
   a[i]=nnum-(FTYPE)i;
   printf("a[%3d] = %8f\n",i,a[i]);
 }
  c8c01<<<grid,blck>>>(a);
 cudaDeviceSynchronize();
 for (i=0; i< nnum ; i++) {
   printf("a[%3d] = %8f\n",i,a[i]);
 }
 cudaFree(a);
}
```

Figure 8.2: Program 2: Program 1 with cudaMallocManaged

8.3 Vector addition

In this section we would describe a variety of methods to perform a simple vector operation, that of vector addition. Two vectors a and b would be added and the result stored in a third vector c. The code is `c8vadd1.cu`, `c8vadd2.cu`, `c8vadd3.cu`, `c8vadd4.cu`, `c8vadd5.cu`, `c8vadd6.cu`. Note that not all code works correctly (e.g. `c8vadd2.cu`) all the time. We present it to show the limitations of some (obvious) grid choices. Moreover `c8vadd6.cu` is a cudaMallocManaged utilized version of `c8vadd4.cu`. The same framework can be used to convert the other programs similarly to the `c8vadd4.cu` into `c8vadd6.cu` conversion.

For `c8vadd1.cu` we would present its code completely into several parts that will span multiple pages. For the rest only the kernel function will be depicted, along with the kernel launch. The remaining lines are identical.

In addition our code also benchmarks the execution of the vector add operation. Timing results will be reported as well.

The kernel code is a function with name the name of the corresponding file minus the extension (e.g. c8vadd1).

8.3.1 c8vadd1.cu

In c8vadd1 the corresponding kernel is launched on a grid consisting of N blocks, each block containing one thread. Given all this prior discussion on warpSize, the choice of a block size of one is poor. The launch code is Figure 8.4. The launch code is extracted from lines 29, 30, and 32 of Figure 8.6. N is the length of the vector. With a total of N threads, each thread will add one element of the vector. The kernel code is Figure 8.3. Variable blockIdx.x provides a unique ID to each thread also mapping to an index of vectors a, b, or c.

After compilation using `nvcc c8vadd1.cu` and execution with `./a.out` we observe an execution time of 0.03789s. The default execution pick $N = 1024$ as a vector length. This can be changed through the command line, and `./a.out 32768` uses $N = 32768$. and reports execution time of 0.35488s. For `./a.out 65536` and $N = 32768$. it reports execution time of 0.77011s. It seems for large values $N \ggg 1024$, doubling of the vector length double execution time. The Mflop rate is a lowly 0.085Mflop/s. Neither a Gflop let alone a Tflop. Do not get tempted to increase blocksize. Limitations of the GPU will creep in eventually. Our code does a limited check to verify that the result is correct. Thus if you try `./a.out 1000000000` and wait some time you will realize that the launch did not launch! Moreover `./a.out 1000000` seems to work and outputs an answer after 11.33331s.

```
1 __global__ void c8vadd1 (FTYPE  *a, FTYPE *b, FTYPE *c, int n) {
2   int tid= blockIdx.x;
3       if (tid < n)
4         c[tid] = a[tid] + b[tid];
5 }
```

Figure 8.3: Vector addition: Version 1 (kernel)

```
1    grid  = N;
2    blck  = 1;
3   c8vadd1<<<grid , blck  >>> (da,db,dc,N);     /* c8vadd1 */
```

Figure 8.4: Vector addition: Version 1 (launch)

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <cuda_runtime.h>
#define NPREDEFINED 1024
#define FTYPE float
#define TPERB 512

void filld(FTYPE *z, int n,int type){
 int i;
 if (type==0)
   memset(z,0,n*sizeof(FTYPE));
 if (type==1)
   for(i=0;i<n; i++) z[i]=FTYPE(i+1);
 if (type==2)
   for(i=0;i<n; i++) z[i]=FTYPE(1+1);
 /* N - (FTYPE) i */
}
void printd(FTYPE *z, int n){
 int i;
 if (n>6) {
    printf("(N=%4d) %4d %4d %4d ... %4d %4d %4d\n",
         n,z[0],z[1],z[2],z[n-3],z[n-2],z[n-1]);
  }
 else {
    printf("(N=%4d)",n);
    for(i=0;i<n;i++) printf("%d ",z[i]);
    printf("\n");
  }
}
void printv(FTYPE *a,FTYPE *b, FTYPE *c, int n){
 int i;
 FTYPE sum1=(FTYPE)0, sum2=(FTYPE)0;
 for(i=0;i<n;i++) sum1+=a[i]+b[i];
 for(i=0;i<n;i++) sum2+=c[i]     ;
 for (i=0; i< 5 ; i++) {
   printf("%4d:(%4f , %4f , %4f)\n",i,a[i],b[i],c[i]);
 }
 printf(" ...\n");
 for (i=n-5; i< n ; i++) {
   printf("%4d:(%4f , %4f , %4f)\n",i,a[i],b[i],c[i]);
 }
 printf("Difference %f - %f must be zero %f\n",sum1,sum2,
  sum1-sum2);
}
```

Figure 8.5: Vector addition: Version 1 (part one)

```
__global__ void c8vadd1 (FTYPE  *a, FTYPE *b, FTYPE *c, int n) {
  int tid= blockIdx.x;
      if (tid < n)
        c[tid] = a[tid] + b[tid];
}

int  main(int argc, char **argv) {    /* c8vadd1 */
 FTYPE  *ha,  *hb,  *hc, *da, *db, *dc;
 size_t size;
 int    grid, blck,N;
 if (argc==1){
              N=NPREDEFINED;
              printf("Usage: %s N\n",argv[0]);
 }
 else         N=atoi(argv[1]);
 size=N*sizeof(FTYPE);
 cudaEvent_t start,stop;
 cudaEventCreate(&start);
 cudaEventCreate(&stop );
 ha=(FTYPE*) malloc(size); filld(ha,N,1);
 hb=(FTYPE*) malloc(size); filld(hb,N,2);
 hc=(FTYPE*) malloc(size); filld(hc,N,0); printv(ha,hb,hc,N);
 cudaMalloc((void**)&da,size);
 cudaMalloc((void**)&db,size);
 cudaMalloc((void**)&dc,size);
 cudaMemcpy(da,ha,size,cudaMemcpyHostToDevice);
 cudaMemcpy(db,hb,size,cudaMemcpyHostToDevice);
 cudaMemcpy(dc,hc,size,cudaMemcpyHostToDevice);
   grid  = N;
   blck  = 1;
 cudaEventRecord(start);
  c8vadd1<<<grid , blck  >>> (da,db,dc,N);
 cudaDeviceSynchronize();
 cudaEventRecord(stop);
 cudaMemcpy(hc,dc,size,cudaMemcpyDeviceToHost);
 cudaEventSynchronize(stop);
 FTYPE mill = 0.0f;
 cudaEventElapsedTime(&mill,start,stop);
 cudaDeviceSynchronize();
 printv(ha,hb,hc,N);
 cudaFree(dc); cudaFree(db); cudaFree(da);
 free(hc);     free(hb);     free(ha);
 printf("Elapsed Time %10.5f\n",mill);
 return(0);
}
```

Figure 8.6: Vector addition: Version 1 (part two)

8.3.2 c8vadd2.cu

In c8vadd2 the corresponding kernel is launched on a grid consisting of 1 block, each block containing N threads. Given all this prior discussion on warpSize, the choice of a grid of size 1 and of a block of size N is equally poor compared to our previous choices. The launch code is Figure 8.8. N is the length of the vector. With a total of N threads in the single block of the grid, each thread will add one element of the vector. The kernel code is Figure 8.7. Variable threadIdx.x provides a unique ID to each thread also mapping to an index of vectors a, b, or c. Compilation can be followed by a default execution `./a.out` that reports a 0.06963s, almost double of the time reported earlier. Furthermore `./a.out 32768` generates incorrect answers. We packed too many threads into one block.

```
__global__ void c8vadd2 (FTYPE  *a, FTYPE *b, FTYPE *c, int n) {
  int tid= threadIdx.x ;
      if (tid < n)
        c[tid] = a[tid] + b[tid];
}
```

Figure 8.7: Vector addition: Version 2 (kernel)

```
   grid  = 1;
   blck  = N;
   c8vadd2<<<grid , blck  >>> (da,db,dc,N); /* c8vadd2 */
```

Figure 8.8: Vector addition: Version 2 (launch)

8.3.3 c8vadd3.cu

In c8vadd3 the corresponding kernel is launched on a grid consisting of several blocks, whose number depends on N, the vector length, each block containing a fixed number of threads (constant TPERB gives the number of threads per block and is set to 512). The launch code is Figure 8.10. N is the length of the vector, TPERB is fixed to 512, and the grid calculation is a ceiling calculation to accommodate the case that N is not a multiple of TPERB. Note that in this setup we have a 1D grid of blocks, and each block is a 1D block of threads. The kernel code is Figure 8.9. Providing unique IDs to threads become more elaborate so we need to go back to section of linearized thread ID to pick the appropriate mapping. For the first time tid $< n$ might provide some robustness assistance. If $N = 1025$ for TPERB equal to 512, three blocks of 512 threads would be utilized. The first two blocks are at full capacity, the third block is almost empty. Line 3 of the kernel deals with the almost empty block of thread execution. Compilation can be followed by a default execution `./a.out` that reports a 0.02867s, better than the best run so far. Furthermore `./a.out 32768` runs in 0.02944s vs a 0.35488 of the Figure 8.3 kernel, i.e. ten times faster. Furthermore `./a.out 1000000` runs in 0.22717s vs a 11.33331 of the Figure 8.3 kernel, i.e. 50 times faster.

You may explore other configurations and capacities of blocks: vary TPERB accordingly.

```
1 __global__ void c8vadd3 (FTYPE  *a, FTYPE *b, FTYPE *c, int n) {
2   int tid= blockIdx.x * blockDim.x + threadIdx.x ;
3       if (tid < n)
4         c[tid] = a[tid] + b[tid];
5 }
```

Figure 8.9: Vector addition: Version 3 (kernel)

```
1    grid  = (N+TPERB-1)/TPERB;
2    blck  = TPERB;
3    c8vadd3<<<grid , blck  >>> (da,db,dc,N);  /* c8vadd3 */
```

Figure 8.10: Vector addition: Version 3 (launch)

8.3.4 c8vadd4.cu

The setup for `c8vadd4.cu` is similar to `c8vadd3.cu`. In all previous cases including `c8vadd3.cu` the length N of the vectors mapped one-to-one to the number of threads. Thus the number of threads was equal to N. In c8vadd4 similarly to the other kernels examined so far, the corresponding kernel is launched on a grid consisting of several blocks, whose number depends on N, the vector length, each block containing a fixed number of threads (constant TPERB gives the number of threads per block and is set to 512) for small values of N. But for large values of N the number of blocks is fixed. This means the threads will have to perform more work to complete the task assigned to them (and the grid). The launch code is Figure 8.12. Note that in this setup we have a 1D grid of blocks, and each block is a 1D block of threads. The kernel code is Figure 8.11. Providing unique IDs to threads becomes more elaborate so we need to adjust for the fact that thread perform more work. Experimental results are given in Figure 8.17.

You may explore other configurations and capacities of blocks: vary TPERB accordingly.

```
__global__ void c8vadd4 (FTYPE  *a, FTYPE *b, FTYPE *c, int n) {
  int tid= blockIdx.x * blockDim.x + threadIdx.x ;
  int step= gridDim.x * blockDim.x;
  int i;
  for(i=0;i<n;i+=step) {
    if ((i+tid) < n)
       c[i+tid] = a[i+tid] + b[i+tid];
  }
}
```

Figure 8.11: Vector addition: Version 4 (kernel)

```
   grid  = (MYMIN(N,32768)+TPERB-1)/TPERB;
   blck  = TPERB;
   c8vadd4<<<grid , blck  >>> (da,db,dc,N); /* c8vadd4.cu */
```

Figure 8.12: Vector addition: Version 4 (launch)

8.3.5 c8vadd5.cu

The setup for `c8vadd5.cu` is similar to that of `c8vadd4.cu`. The minor change is in the kernel; the for loop has become a while loop. The launch code is Figure 8.14. The kernel code is Figure 8.13. Experimental results are given in Figure 8.17. You may explore other configurations and capacities of blocks: vary TPERB accordingly.

```
__global__ void c8vadd5 (FTYPE  *a, FTYPE *b, FTYPE *c, int n) {
  int tid= blockIdx.x * blockDim.x + threadIdx.x ;
  while (tid<n){
       c[tid  ] = a[tid  ] + b[tid  ];
       tid += blockDim.x * gridDim.x;
  }
}
```

Figure 8.13: Vector addition: Version 5 (kernel)

```
   grid  = (MYMIN(N,32768 )+TPERB-1)/TPERB;
   blck  = TPERB;
   c8vadd5<<<grid , blck  >>> (da,db,dc,N); /* c8vadd5.cu */
```

Figure 8.14: Vector addition: Version 5 (launch)

8.3.6 c8vadd6.cu

The setup for `c8vadd6.cu` is similar to that of `c8vadd4.cu`; the former however uses cudaMallocManaged and managed memory that does not need host to device communication duplication. The launch code is Figure 8.16. The kernel code is Figure 8.15. Experimental results are given in Figure 8.17. You may explore other configurations and capacities of blocks: vary TPERB accordingly.

For large values of N it is the only one that works. However performance drops overall, and for larger values of N close to a billion performance is erratic. The launch kernal needs to adjust the grid size. It is very likely that spillage of registers occurs for such value that make performance 20 times worse than expected. Thus changing $(\mathrm{MYMIN}(\mathrm{N},1048576)+\mathrm{TPERB}-1)/\mathrm{TPERB}$ into $(\mathrm{MYMIN}(\mathrm{N},65536)+\mathrm{TPERB}-1)/\mathrm{TPERB}$ can accomplish this: a 5-factor improvement occurred. Decreasing the 65536 to 32768 can lead to a deterioration of performance.

```
__global__ void c8vadd6 (FTYPE  *a, FTYPE *b, FTYPE *c, int n) {
  int tid= blockIdx.x * blockDim.x + threadIdx.x ;
  int step= gridDim.x * blockDim.x;
  int i;
  for(i=0;i<n;i+=step) {
    if ((i+tid) < n)
       c[i+tid] = a[i+tid] + b[i+tid];
  }
}
```

Figure 8.15: Vector addition: Version 6 (kernel)

```
    grid  = (MYMIN(N,1048576)+TPERB-1)/TPERB;
    blck  = TPERB;
    c8vadd6<<<grid , blck  >>> (ha,hb,hc,N); /* c8vadd6.cu */
```

Figure 8.16: Vector addition: Version 6 (launch)

N	Version 3	Version 4	Version 5	Version 6
1024	0.02	0.02	0.02	0.3
32K	0.02	0.02	0.02	0.4
256K	0.07	0.07	0.06	1.1
1024K	0.23	0.24	0.24	3.5
4194304	0.92	0.92	0.92	13.82
16777216	3.62	3.76	3.76	64.91
67108864	14.54	15.02	15.00	213.30
268435456	Fail	Fail	Fail	801.52
536870912	Fail	Fail	Fail	8481.85-35913.23

Figure 8.17: Vector Addition: Experimental Results

8.4 Vector Multiply and Add

In this section we present code similar to that of the previous section dealing with vector addition. The code is for multiply and add. Thus for three vectors a[],b[],c[]. the element at offset i is computed as follows: c[i] = c[i] + $a[i] * b[i]$.

We provide the sixteen kernels below starting with Figure 8.19 and ending with Figure 8.33. All grid kernels use a 1D arrangement of blocks. Each block has an 1D arrangement of threads. The wrapper code is the same with two exceptions: (a) the launched kernel grid is described in the table of Figure 8.18, (b) Kernel 6's CUDA code utilized cudaMallocManaged similarly to the same numbered code for vector additions.

The table of Figure 8.18 is presented first followed by the kernels. The table presents some benchmarks for $N = 262144$. Kernel 2 failed to run because of thread capacity issues. Another kernel, Kernel 11, shows how one can bypass such problems. Some kernels such as Kernel 13, Kernel 14, and Kernel 15 are variant of Kernel 3, Kernel 4, and Kernel 5 with different launched grids. Also Kernel 7 and Kernel 8 are identical but with different launched grid. This shows the importance of grid choices.

For the table of Figure 8.18, column Grid indicates the grid dimension, the number of 1D blocks launched. Column Block indicates the number of threads in a 1D block. Performance indicates time in millisecond for all CUDA activity for $N = 262144$. Comment provides a rudimentary commentary for the corresponding kernel. Note that Kernels 13, 14, and 15 are labeled in Figure 8.31 through Figure 8.33 as Kernel03, Kernel04 and Kernel05 to highlight the similarity with kernels as Kernel 3, Kernel 4 and Kernel 5 respectively. A fraction such as $N/512$ for a kernel grid should be interpreted as ceiling of $N/512$ i.e. $(N+512-1)/512$. Similarly for $N/128$. For the starred versions, $N/512^*$, we use a grid of $(\min(N,32768)+512-1)/512$ or $(\min(N,32768)+128-1)/128$ blocks as indicated.

Kernel 8.19 is serial code with one grid launching one block containing one thread. Kernel 8.20 uses one grid launching N blocks each containing one thread: an underutilized block approach. The symmetric Kernel 8.21 loads too many threads in the single block grid. Naturally it fails for $N = 262144$.

Kernel	Grid	Block	Performance(ms)	Comment
0	1	1	53.36	Serial code; Work per thread N
1	N	1	3.08	Underutilized block;capacity issues
2	1	N	FAIL	One op per thread; capacity issues
3	N/512	512	0.093	Flat; no loops; capacity issues
4	N/512*	512	0.091	For-loop kernel
5	N/512*	512	0.085	While-loop kernel
6	N/512*	512	1.496	CudaManaged version For-loop
7	N/128*	128	0.258	Different inefficient For-loop
8	N/64	64	0.087	Kernel 7 grid variant
10	N	1	3.284	Restructured Kernel 1
11	1	128	1.056	Restructured Kernel 2
12	1	128	0.806	Restructured Kernel 11/Kernel 2
13(i.e. 03)	N/128	128	0.089	Kernel 3 different grid
14(i.e. 04)	N/128	128	0.089	Kernel 4 different grid
15(i.e. 05)	N/128	128	0.088	Kernel 5 different grid

Figure 8.18: Vector Multiply and Add kernels

```
__global__ void c8vmul0 (FTYPE  *a, FTYPE *b, FTYPE *c, int n) {
  int tid;
  for(tid=0; tid<n; ++tid)
   c[tid] += a[tid]*b[tid];
}
```

Figure 8.19: Vector multiply and add (Kernel 0)

```
__global__ void c8vmul1 (FTYPE  *a, FTYPE *b, FTYPE *c, int n) {
  int tid= blockIdx.x;
      if (tid < n)
        c[tid] += a[tid] * b[tid];
}
```

Figure 8.20: Vector multiply and add (Kernel 1)

```
__global__ void c8vmul2 (FTYPE  *a, FTYPE *b, FTYPE *c, int n) {
  int tid= threadIdx.x ;
      if (tid < n)
        c[tid] += a[tid] * b[tid];
}
```

Figure 8.21: Vector multiply and add (Kernel 2)

```
__global__ void c8vmul3 (FTYPE  *a, FTYPE *b, FTYPE *c, int n) {
  int tid= blockIdx.x * blockDim.x + threadIdx.x ;
      if (tid < n)
        c[tid] += a[tid] * b[tid];
}
```

Figure 8.22: Vector multiply and add (Kernel 3)

```
__global__ void c8vmul4 (FTYPE  *a, FTYPE *b, FTYPE *c, int n) {
  int tid= blockIdx.x * blockDim.x + threadIdx.x ;
  int step= gridDim.x * blockDim.x;
  int i;
  for(i=0;i<n;i+=step) {
    if ((i+tid) < n)
       c[i+tid] += a[i+tid] * b[i+tid];
  }
}
```

Figure 8.23: Vector multiply and add (Kernel 4)

```
__global__ void c8vmul5 (FTYPE  *a, FTYPE *b, FTYPE *c, int n) {
  int tid= blockIdx.x * blockDim.x + threadIdx.x ;
  while (tid<n){
       c[tid  ] += a[tid  ] * b[tid  ];
       tid += blockDim.x * gridDim.x;
  }
}
```

Figure 8.24: Vector multiply and add (Kernel 5)

```
__global__ void c8vmul6 (FTYPE  *a, FTYPE *b, FTYPE *c, int n) {
  int tid= blockIdx.x * blockDim.x + threadIdx.x ;
  int step= gridDim.x * blockDim.x;
  int i;
  for(i=0;i<n;i+=step) {
    if ((i+tid) < n)
       c[i+tid] += a[i+tid] * b[i+tid];
  }
}
```

Figure 8.25: Vector multiply and add (Kernel 6)

```
__global__ void c8vmul7 (FTYPE  *a, FTYPE *b, FTYPE *c, int n) {
  int tid= blockIdx.x * blockDim.x + threadIdx.x ;
  int i,m,bb;
  bb= blockDim.x *gridDim.x;
  m= n/bb;
  bb= tid* m;
  for(i=0;i<m;++i) {
    if ((i+bb ) < n)
       c[i+bb ] += a[i+bb ] * b[i+bb ];
  }
}
```

Figure 8.26: Vector multiply and add (Kernel 7)

```
__global__ void c8vmul8 (FTYPE  *a, FTYPE *b, FTYPE *c, int n) {
  int tid= blockIdx.x * blockDim.x + threadIdx.x ;
  int i,m,bb;
  bb= blockDim.x *gridDim.x;
  m= n/bb;
  bb= tid* m;
  for(i=0;i<m;++i) {
    if ((i+bb ) < n)
       c[i+bb ] += a[i+bb ] * b[i+bb ];
  }
}
```

Figure 8.27: Vector multiply and add (Kernel 8)

```
__global__ void c8vmul10(FTYPE  *a, FTYPE *b, FTYPE *c, int n) {
  int tid = blockIdx.x ;
  int nblk= gridDim.x;
  int nsub= n/nblk;
  int i;

  for(i=tid*nsub; i<(tid+1)*nsub;++i)
   c[i] += a[i]*b[i];
}
```

Figure 8.28: Vector multiply and add (Kernel 10)

```
__global__ void c8vmul11(FTYPE  *a, FTYPE *b, FTYPE *c, int n) {
  int tid = threadIdx.x ;
  int nthr= blockDim.x * gridDim.x ;
  int nsub= n/nthr;
  int i;

  for(i=tid*nsub; i<(tid+1)*nsub;++i)
   c[i] += a[i]*b[i];
}
```

Figure 8.29: Vector multiply and add (Kernel 11)

```
__global__ void c8vmul12(FTYPE  *a, FTYPE *b, FTYPE *c, int n) {
  int tid = threadIdx.x ;
  int nthr= blockDim.x * gridDim.x ;
  int i;

  for(i=0; i<n;i+=nthr)
   c[i+tid] += a[i+tid]*b[i+tid];
}
```

Figure 8.30: Vector multiply and add (Kernel 12)

```
__global__ void c8vmul03(FTYPE  *a, FTYPE *b, FTYPE *c, int n) {
  int tid = blockIdx.x * blockDim.x + threadIdx.x ;
  if (tid<n)
    c[tid] += a[tid]*b[tid];
}
```

Figure 8.31: Vector multiply and add (Kernel 13)

```
__global__ void c8vmul04(FTYPE  *a, FTYPE *b, FTYPE *c, int n) {
  int tid = blockIdx.x * blockDim.x + threadIdx.x ;
  int step= gridDim.x * blockDim.x;
  int i;
  for(i=0;i<n;i+=step) {
    if ((i+tid) < n)
       c[i+tid] += a[i+tid] * b[i+tid];
  }
}
```

Figure 8.32: Vector multiply and add (Kernel 14)

```
__global__ void c8vmul05(FTYPE  *a, FTYPE *b, FTYPE *c, int n) {
  int tid = blockIdx.x * blockDim.x + threadIdx.x ;
  int step= gridDim.x * blockDim.x;
  while (tid<n){
     c[tid] += a[tid] * b[tid];
     tid += step;
  }
}
```

Figure 8.33: Vector multiply and add (Kernel 15)

8.5 Inner Product

There are two implementations for matrix vector product One uses shared memory and a slower using mutexes. Note that the code is not robust. There is some N value, where N is the length of the two vectors, for which the implementations would give incorrect results: we assume that there are enough threads to be assigned to all elements of the two vectors a, b whose inner product would be computed.

8.5.1 Shared memory edition

The code for `c8inp1.cu` is displayed in three parts. Part1 is `c8inp1_p1.cu` that contains initialization and output functions plus declarations. Part1 is `c8inp1_p2.cu` that contains initialization and output functions plus declarations.

```
    grid  = (N+TPERB-1)/TPERB;
    blck  = TPERB;
  cudaEventRecord(start);
   c8inp1 <<<grid , blck  >>> (da,db,dc,N);     /* c8inp1  */
  cudaDeviceSynchronize();
  cudaEventRecord(stop);
```

Figure 8.34: Inner product (shmem) Launch

```
__global__ void c8inp1  (FTYPE  *a, FTYPE *b, FTYPE *c, int n) {
    __shared__ float mycache[TPERB];

   int   index = blockIdx.x * blockDim.x + threadIdx.x;
   int   cindx = threadIdx.x, step;
   FTYPE temp= (FTYPE)0;

   while (index<n) {
      temp += a[ index ] * b[ index ];
     index += blockDim.x * gridDim.x ;
   }
   mycache[ cindx ] = temp;

   __syncthreads();
   step = blockDim.x / 2 ;
   while (step!=0) {
     if (cindx < step) {
       mycache[cindx] += mycache[cindx+step];
     }
     __syncthreads();
     step >>=1;
   }
   if (threadIdx.x == 0 ) {
    c[blockIdx.x] = mycache[0];
   }
}
```

Figure 8.35: Inner product (shmem) Kernel

Let us consider a slightly more involved example that shows how threads can interact with each other. In Figure 8.35 we see the kernel for the inner vector product code. Let $P = \text{blockDim.x} \times \text{gridDim.x}$. Every

thread with threadID T = index, where index = blockIdx.x $\times$ blockDim.x + threadIdx.x (line 4), computes the partial products (and their sum contribution to the inner product) of the indexes T, $T+P$, $T+2P$, $T+3P$, and so on. The sum of those partial products is computed in variable temp. The threads of every block share a common shared memory. Thus the value of variable temp gets stored inside that shared memory named mycache at offset threadIdx.x. Note that the offset only depends on threadIdx.x rather than say blockIdx.x: different blocks have different shared memory assigned to each block. The size of shared memory is enough to accommodate the blockDim.x threads of a block. In our code blockDim.x is set to be a TPERB and by default equal to 512. This is also the size of the shared memory. As soon as in a block the threads have computed the partial vector products assigned to them and their sum stored in main memory, then the sum of the values stored in shared memory is performed. This requires $\lg TPERB$ rounds, where lg . is the logarithm base two of a number. In other words, a reduction reduces x sums into $x/2$ first, then $x/4$ and eventually into one sum stored at offset 0 of the shared memory. This value is retrieved in line 24 of the kernel into array c. The offset of the array is the blockIdx.x of the block involved in this specific calculation. Thus array c is of length equal to the number of blocks of the grid.

In Figure 8.37 we see the kernel, and in Figure 8.36 the launch for a inner product computation using mutexes.

```
1    grid  = (N+TPERB-1)/TPERB;
2    blck  = TPERB;
3  cudaEventRecord(start);
4   c8inp2 <<<grid , blck  >>> (da,db,dc,N,mutex);    /* c8inp2  */
5  cudaDeviceSynchronize();
6  cudaEventRecord(stop);
```

Figure 8.36: Inner product (mutex) Launch

```
__global__ void c8inp2  (FTYPE  *a, FTYPE *b, FTYPE *c, int n,int *mutex) {
    __shared__ float mycache[TPERB];

   int   index = blockIdx.x * blockDim.x + threadIdx.x;
   int   cindx = threadIdx.x, step;
   FTYPE temp= (FTYPE)0;
   int   isset=0;
   while (index<n) {
      temp += a[ index ] * b[ index ];
     index += blockDim.x * gridDim.x ;
   }
   mycache[ cindx ] = temp;

   __syncthreads();
   step = blockDim.x / 2 ;
   while (step!=0) {
     if (cindx < step) {
       mycache[cindx] += mycache[cindx+step];
     }
     __syncthreads();
     step >>=1;
   }
   if (threadIdx.x == 0 ) {
     do {
       if (isset=atomicCAS(mutex,0,1)==0) {
         *c += mycache[0];
       }
       if (isset) {
         *mutex=0;
       }
     }
     while (!isset);
   }
}
```

Figure 8.37: Inner product (mutex) Kernel

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <cuda_runtime.h>
#define NPREDEFINED 1024
#define FTYPE float
#define TPERB 512
#define NBLCK ((N+TPERB-1)/TPERB)

void filld(FTYPE *z, int n,int type){
 int i;
 if (type==0)
   memset(z,0,n*sizeof(FTYPE));
 if (type==1)
   for(i=0;i<n;++i) z[i]=FTYPE(i+1);
 if (type==2)
   for(i=0;i<n;++i) z[i]=FTYPE(1+1);
 if (type==3){
   for(i=0;i<n;++i) {
     if ((i%2) == 0)
        z[i]=(FTYPE)2;
     else
        z[i]=(FTYPE)1;
   }
 }
}
void printv(FTYPE *a,FTYPE *b, FTYPE val, int n){
 int i;
 double sum=(double)0;
 for(i=0;i<n;++i) {
   sum+=a[i]*b[i];
 }
 for (i=0; i< 4 ;++i) {
   printf("%4d:(%4f , %4f )\n",i,a[i],b[i]);
 }
 printf(" ...\n");
 for (i=n-4; i< n;++i) {
   printf("%4d:(%4f , %4f )\n",i,a[i],b[i]);
 }
 printf("Difference %f - %f must be zero %f\n",(double)sum,(double)val ,
  (double)(sum-val) );
}
```

Figure 8.38: Inner product (shmem) part 1 of 2

```
int  main(int argc, char **argv) {
 FTYPE  *ha,  *hb, *hc, *da, *db, *dc, hval=(FTYPE)0;
 size_t size,siz;
 int    grid, blck,N,i;
 if (argc==1){
      N=NPREDEFINED;
      printf("Usage: %s N\n",argv[0]);
 }
 else N=atoi(argv[1]);
 size=N*sizeof(FTYPE);
 siz=NBLCK*sizeof(FTYPE);
 cudaEvent_t start,stop;
 cudaEventCreate(&start);
 cudaEventCreate(&stop );
 ha=(FTYPE*) malloc(size); filld(ha,N,3);
 hb=(FTYPE*) malloc(size); filld(hb,N,3);
 hc=(FTYPE*) malloc(siz ); filld(hc,NBLCK,0);
 printv(ha,hb,hval,N);
 cudaMalloc((void**)&da,size);
 cudaMalloc((void**)&db,size);
 cudaMalloc((void**)&dc,siz );

 cudaMemcpy(da,ha,size,cudaMemcpyHostToDevice);
 cudaMemcpy(db,hb,size,cudaMemcpyHostToDevice);
 cudaMemset(dc,0,siz );

   grid  = (N+TPERB-1)/TPERB;
   blck  = TPERB;
 cudaEventRecord(start);
  c8inp1 <<<grid , blck  >>> (da,db,dc,N);    /* c8inp1  */
 cudaDeviceSynchronize();
 cudaEventRecord(stop);
 cudaMemcpy(hc,dc,siz ,cudaMemcpyDeviceToHost);
 cudaEventSynchronize(stop);
 FTYPE mill = 0.0f;
 hval=0;
 for(i=0;i< NBLCK ; ++i)
     hval += hc[i];
 cudaEventElapsedTime(&mill,start,stop);
 cudaDeviceSynchronize();
 printv(ha,hb,hval,N);
 cudaFree(dc); cudaFree(db); cudaFree(da);
 free(hb);     free(ha);
 printf("N=%d val=%f Elapsed Time %10.5f\n",hval,N,mill);
 return(0);
}
```

Figure 8.39: Inner product (shmem) part 2 of 2

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <cuda_runtime.h>
#define NPREDEFINED 1024
#define FTYPE float
#define TPERB 512
#define NBLCK ((N+TPERB-1)/TPERB)

void filld(FTYPE *z, int n,int type){
 int i;
 if (type==0)
   memset(z,0,n*sizeof(FTYPE));
 if (type==1)
   for(i=0;i<n;++i) z[i]=FTYPE(i+1);
 if (type==2)
   for(i=0;i<n;++i) z[i]=FTYPE(1+1);
 if (type==3){
   for(i=0;i<n;++i) {
     if ((i%2) == 0)
        z[i]=(FTYPE)2;
     else
        z[i]=(FTYPE)1;
   }
 }
}
void printv(FTYPE *a,FTYPE *b, FTYPE val, int n){
 int i;
 double sum=(double)0;
 for(i=0;i<n;++i) {
   sum+=a[i]*b[i];
 }
 for (i=0; i< 4 ;++i) {
   printf("%4d:(%4f , %4f )\n",i,a[i],b[i]);
 }
 printf(" ...\n");
 for (i=n-4; i< n ;++i) {
   printf("%4d:(%4f , %4f )\n",i,a[i],b[i]);
 }
 printf("Difference %f - %f must be zero %f\n",(double)sum,(double)val ,
  (double)(sum-val) );
}
```

Figure 8.40: Inner product (mutex) part 1 of 2

```
int  main(int argc, char **argv) {
 FTYPE  *ha,  *hb,  *da, *db, *dc, hval=(FTYPE)0;
 size_t size;
 int    grid, blck,N, *mutex,lckstate=0;
 if (argc==1){
      N=NPREDEFINED;
      printf("Usage: %s N\n",argv[0]);
 }
 else N=atoi(argv[1]);
 size=N*sizeof(FTYPE);

 cudaEvent_t start,stop;
 cudaEventCreate(&start);
 cudaEventCreate(&stop );
 ha=(FTYPE*) malloc(size); filld(ha,N,3);
 hb=(FTYPE*) malloc(size); filld(hb,N,3);

 printv(ha,hb,hval,N);
 cudaMalloc((void**)&da,size);
 cudaMalloc((void**)&db,size);
 cudaMalloc((void**)&dc,sizeof(FTYPE));
 cudaMalloc((void**)&mutex,sizeof(int));
 cudaMemcpy(da,ha,size,cudaMemcpyHostToDevice);
 cudaMemcpy(db,hb,size,cudaMemcpyHostToDevice);
 cudaMemcpy(dc,&hval,sizeof(FTYPE),cudaMemcpyHostToDevice);
 cudaMemcpy(mutex,&lckstate,sizeof(int),cudaMemcpyHostToDevice);
   grid  = (N+TPERB-1)/TPERB;
   blck  = TPERB;
 cudaEventRecord(start);
  c8inp2 <<<grid , blck  >>> (da,db,dc,N,mutex);    /* c8inp2  */
 cudaDeviceSynchronize();
 cudaEventRecord(stop);
 cudaMemcpy(&hval,dc,sizeof(FTYPE),cudaMemcpyDeviceToHost);
 cudaEventSynchronize(stop);
 FTYPE mill = 0.0f;



 cudaEventElapsedTime(&mill,start,stop);
 cudaDeviceSynchronize();
 printv(ha,hb,hval,N);
 cudaFree(dc); cudaFree(db); cudaFree(da);
 free(hb);     free(ha);
 printf("N=%d val=%f Elapsed Time %10.5f\n",hval,N,mill);
 return(0);
}
```

Figure 8.41: Inner product (mutex) part 2 of 2

Chapter 9

Matrix Operations

We start by showing code for matrix addition. It can be adopted for matrix subtraction and other operations that are element wise.

We then present NVIDIA matrix multiplication code starting with a simple solution and proceeding to solutions that use shared memory (shared by the threads of a thread block).

After that in a separate section we present several similar approaches that (a) store a matrix in a column-major order rather than the default column major, and (b) use row-major but offer variations of the scheme available through the NVIDIA examples.

9.1 Matrix Addition

The matrices are stored in the form of a 1D array in column major form. Thus $A[i][j]$ is stored in array a at offset $j*N+i$, for a square matrix of order N or a matrix A with N rows. Part 2 of the code for matrix addition is the kernel. Part 1 are testing functions and definition, and Part 3 is the main function that includes the kernel launch. The gird geometry is adapted to match the square structure of the input arrays. Thus if p is the number of threads per thread block (constant TPERB is p), the number of block of the grid would be N^2/p, and the grid would have a 2D geometry of $N/\sqrt{p} \times N/\sqrt{p}$ blocks arranged in rows and columns. A thread block of p threads would form a 2D structure of $\sqrt{p} \times \sqrt{p}$ threads also arranged in grid form. In the code SQRTP denotes the $\sqrt{p}$ of this discussion.

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <cuda_runtime.h>
#define NPREDEFINED 1024
#define FTYPE float
        /* SQRTP = sqrt(TPERB) */
#define TPERB  1024
#define SQRTP  32
        /* DOUBLECHECK PREV VALUES ARE CONSISTENT */
#define MYMIN(X,Y) ((X)<(Y)?(X):(Y))
#define R 4

void fillm(FTYPE *z, int n,int type){
 int i,j;
 if (type==0)
   memset(z,0,n*n*sizeof(FTYPE));
 if (type==1)
   for(i=0;i<n;++i)
    for(j=0;j<n;++j){z[j*n+i]=(FTYPE)(j+1);}
 if (type==2)
   for(i=0;i<n;++i)
    for(j=0;j<n;++j){z[j*n+i]=(FTYPE)(2*j+i+1);}
}
void printm(FTYPE *a,FTYPE *b, FTYPE *c, int n,int debug){
 int i,j;
 double sum1=(double)0, sum2=(double)0, sum3=(double)0;
 for(i=0;i<n*n;++i) sum1 += a[i];
 for(i=0;i<n*n;++i) sum2 += b[i];
 for(i=0;i<n*n;++i) sum3 += c[i];
 printf("asig=%f , bsig=%f , csig=%f \n",sum1,sum2,sum3);
if (debug){
 printf("Input A \n");
  for(i=0;(i<n);++i)
   for(j=0;(j<n);++j)
     if((   (i<R) || ((i>n-1-R) && (i<n))) &&
        (   (j<R) || ((j>n-1-R) && (j<n))) )
     printf("%6.1f%c",a[j*n+i],((j<(n-1))?' ':'\n'));
 printf("Input B \n");
  for(i=0;(i<n);++i)
   for(j=0;(j<n);++j)
     if((   (i<R) || ((i>n-1-R) && (i<n))) &&
        (   (j<R) || ((j>n-1-R) && (j<n))) )
     printf("%6.1f%c",b[j*n+i],((j<(n-1))?' ':'\n'));
 printf("Input C \n");
  for(i=0;(i<n);++i)
   for(j=0;(j<n);++j)
     if((   (i<R) || ((i>n-1-R) && (i<n))) &&
        (   (j<R) || ((j>n-1-R) && (j<n))) )
     printf("%6.1f%c",c[j*n+i],((j<(n-1))?' ':'\n'));
}
}
```

Figure 9.1: Matrix addition part 1

```
__global__ void c9ma01 (FTYPE  *a, FTYPE *b, FTYPE *c, int n) {
  int i= threadIdx.x + blockIdx.x * blockDim.x;
  int j= threadIdx.y + blockIdx.y * blockDim.y;
  if (i < n && j< n)
     c[j*n+i] = a[j*n+i]+b[j*n+i];
}
```

Figure 9.2: Matrix addition part 2

```
int  main(int argc, char **argv) {
 FTYPE  *ha,  *hb,  *hc, *da, *db, *dc;
 size_t size;
 int    N,debug=0;
 if (argc==1){
              N=NPREDEFINED; debug=0;
              printf("Usage: %s N debug(0 or 1)\n",argv[0]);
 }
 else {
              N=atoi(argv[1]); debug=atoi(argv[2]);
      }
 size=N*N*sizeof(FTYPE);
 cudaEvent_t start,stop;
 cudaEventCreate(&start);
 cudaEventCreate(&stop );
 ha=(FTYPE*) malloc(size); fillm(ha,N,1);
 hb=(FTYPE*) malloc(size); fillm(hb,N,2);
 hc=(FTYPE*) malloc(size); fillm(hc,N,0); printm(ha,hb,hc,N,debug);
 cudaDeviceSynchronize();
 cudaMalloc((void**)&da,size);
 cudaMalloc((void**)&db,size);
 cudaMalloc((void**)&dc,size);
 cudaMemcpy(da,ha,size,cudaMemcpyHostToDevice);
 cudaMemcpy(db,hb,size,cudaMemcpyHostToDevice);
 cudaMemcpy(dc,hc,size,cudaMemcpyHostToDevice);
 dim3  blck(SQRTP,SQRTP);
 dim3  grid((N+blck.x-1)/blck.x,(N+blck.y-1)/blck.y);
 cudaEventRecord(start);
  c9ma01 <<<grid , blck  >>> (da,db,dc,N);    /* c9ma01 */
 cudaDeviceSynchronize();
 cudaEventRecord(stop);
 cudaMemcpy(hc,dc,size,cudaMemcpyDeviceToHost);
 cudaEventSynchronize(stop);
 FTYPE mill = 0.0f;
 cudaEventElapsedTime(&mill,start,stop);
 cudaDeviceSynchronize();
 printm(ha,hb,hc,N,debug);
 cudaFree(dc); cudaFree(db); cudaFree(da);
 free(hc);     free(hb);     free(ha);
 printf("Elapsed Time %10.5f\n",mill);
 return(0);
}
```

Figure 9.3: Matrix addition part 3

9.2 Matrix transposition

We present some naive implementations for matrix transposition. The kernel of Figure 9.4 assumes a column-major storage of a matrix $A[][]$ as a 1D array $a[]$. The kernel of Figure 9.5 rearranges and use an i, j loop indexing versus the j, i loop order of the kernel in Figure 9.4.

The kernel of Figure 9.6 assumes a row-major storage of a matrix $A[][]$ as a 1D array $a[]$ and uses the j, i loop order of the kernel in Figure 9.4 .

All three kernels are launched with a grid of one block and one thread in it. A rather serial invocation!

The kernel of Figure 9.7 exhibits some paralellism. It is invoked with a invocation similar to the one given below assuming an 1D grid and 1D block of 1024 threads. It results in a hundred-fold increase in performance for $n = 1024$ and 300-fold increase for $n = 2048$ and square matrices as the ones indicates in all kernels.

$$c9tr01 \lll (n + 1024 - 1)/1024, 1024 \ggg (da, db, n);$$

```
__global__ void c9tr00c(FTYPE  *a, FTYPE *b, int n) {
   int i,j;
   for(j=0;j<n;j++)
      for(i=0;i<n;i++)
         b[i*n+j]=a[j*n+i];
}
```

Figure 9.4: Matrix transposition naive ji kernel (column-major)

```
__global__ void c9tr01c(FTYPE  *a, FTYPE *b, int n) {
   int i,j;
      for(i=0;i<n;i++)
        for(j=0;j<n;j++)
         b[i*n+j]=a[j*n+i];
}
```

Figure 9.5: Matrix transposition naive ij kernel (column-major)

```
__global__ void c9tr00r(FTYPE  *a, FTYPE *b, int n) {
   int i,j;
   for(j=0;j<n;j++)
      for(i=0;i<n;i++)
         b[j*n+i]=a[i*n+j];
}
```

Figure 9.6: Matrix transposition naive kernel (row-major)

9.2.1 Shared memory usage in transposition

Solutions presented earlier would never extract maximal GPU performance because of *uncoalesced global memory* (load) access. In CUDA, execution of threads on an SM occurs at warp level (32 threads). "Coalesced" access occurs when the 32 threads access adjacent memory locations. "Uncoalesced" access occurs

```
__global__ void c9tr01 (FTYPE  *a, FTYPE *b, int n) {
   int i = blockIdx.x * blockDim.x + threadIdx.x ;
   int j;
      for(j=0;j<n;j++)
        b[j*n+i] = a[i*n+j];
}
```

Figure 9.7: Matrix transposition naive kernel (row-major)

when the memory locations have gaps (i.e. the stride is greater than 1). In matrix transposition no matter how we store the matrix as an one-dimensional entity, either the writing or the reading would occur in an uncoalesced way. Using ANSI C language's `restrict` statement we allow for optimization of assignment operations by NVIDIA's CUDA. This is shown in the kernel of Figure 9.8. It results into a 30% improvement in execution time over the kernal of Figure 9.7.

```
__global__ void c9tr01 (const FTYPE  * __restrict__ a, FTYPE *b, int n) {
   int i = blockIdx.x * blockDim.x + threadIdx.x ;
   int j;
      for(j=0;j<n;j++)
        b[j*n+i] = a[i*n+j];
   // column-major write of b ; row-major read of a
}
```

Figure 9.8: Matrix transposition naive kernel (row-major)

9.2.2 Further optimization

Some further optimization can be achieved through the use of shared memory! We also use information available in the reference "GP-GPU Programming with CUDA" by Larry Brown. We tranpose the matrix by transposing small sub-blocks of it (say 32×32) that are stored in shared memory. It is very likely that performance of shared memory would be worse. This is because shared memory is organized in 32 banks, each warp has 32 threads and either row or column access generates shared memory conflicts that are serialized among threads. In the example of Figure 9.9 row accesses are to independent banks (i.e. no conflicts) but column accesses are to the same bank. Thus in the former case the 32 elements of a row can be read in parallel in one step, but in the other case, the columns would require serialization i.e. 32 steps for a read. Adding one extra column that is left unused breaks this pattern. See below in the kernel of Figure 9.10. For example the padded column of the first row is assigned to Bank 0 and thus the second row starts with Bank 1, not Bank 0 that the first row started with. This results in a 50% performance improvement for transposing a square matrix of order $n = 1024$.

Note that the kernel grid has blocks with 1024 threads arranged in a 2D pattern of 32×32 inside a block. All optimizations described use such a layout.

Structures of arrays or arrays of structures? In general, an array of C `structs` is not optimal in a GPU. Convert first into a `struct` of arrays by "transposing" the data! Use shared memory to handle block transposes!

```
__global__ void c9tr05 (FTYPE  *a, FTYPE *b, int n) {
   __shared__ float small[SQRTP][SQRTP];
   int i = blockIdx.x * blockDim.x + threadIdx.x ;
   int j = blockIdx.y * blockDim.y + threadIdx.y ;
   small[threadIdx.y][threadIdx.x] = a[j*n+i];
   __syncthreads();
       i = blockIdx.y * blockDim.y + threadIdx.x ;
       j = blockIdx.x * blockDim.x + threadIdx.y ;
   b[j*n+i] = small[threadIdx.x][threadIdx.y];
}
```

Figure 9.9: Matrix transposition with shared memory

```
__global__ void c9tr05p(FTYPE  *a, FTYPE *b, int n) {
   __shared__ float small[SQRTP][SQRTP+1];
   int i = blockIdx.x * blockDim.x + threadIdx.x ;
   int j = blockIdx.y * blockDim.y + threadIdx.y ;
   small[threadIdx.y][threadIdx.x] = a[j*n+i];
   __syncthreads();
       i = blockIdx.y * blockDim.y + threadIdx.x ;
       j = blockIdx.x * blockDim.x + threadIdx.y ;
   b[j*n+i] = small[threadIdx.x][threadIdx.y];
}
```

Figure 9.10: Matrix transposition with (padded) shared memory

```
0_0   1_1  2_2  3_3   ...     30_30  31_31   i is column  _j is Bank  ; all of i column in Bank i.
         ...
0_0   1_1  2_2  3_3   ...     30_30  31_31

Add a 'dummy column
0_0   1_1  2_2  3_3   ...     30_30  31_31 32_0
0_1   1_2  2_3  3_4   ...     30_31  31_0  32_1
         ...                     ...
0_31  1_0  2_1  3_2           30_29  31_30 32_31
```

Figure 9.11: How to break conflicts!

9.3 NVIDIA examples for matrix multiplication

We multiply two matrics a and B and store the result in matrix C i.e. $C = A \times B$. Let the dimensions of A and B be $n \times p$ and $p \times m$ so that the dimension of C is $n \times m$.

For the sample code presented in this section we shall assume that $N = n = p = m$, i.e. the matrices are all square of the same dimension (geometry) or order N.

Each thread block would be responsible for the computation of a square submatrix of c.

The discussion that follows is from "Cuda C Programming, Design Guide", NVIDIA, PG-02829-0001_v7.5, Sep 2015, pages 39-45.

Matrices A, B, C are to be structured as one-dimensional arrays and elements stored in row-major form with indexes starting from 0 (with row 0 left to right first, then row 1 left to right and so on). The block size determines the geometry of a block to 64 or 1024 threads in a 2-dimensional grid i.e. blockDim.x = 32, blockDim.y = 32, blockDim.z = 1, for the case that a block maps to 1024 threads. In experiments we are going to examine also the case of 64 threads arranged as a 8×8 2D thread block.

Matrices A, B, C are first allocated and initialized, and then copied from host to device memory. The ensuing matrix multiplication requires that every row of a is read m times (columns of B and C), and every column of B is read n times (rows of A and C).

9.3.1 NVIDIA version 1: simple implementation

Figure 9.13 contains the kernel. Figure 9.12 contains initialization and debugging code plus definitions. Constant TPERB is the number of threads per block arranges as a 2D structured of SQRTP $\times$ SQRTP threads where SQRTP $= \sqrt{\text{TPERB}}$. The default values are 1024 and 32 respectively. We only tested the code with this set and also 64 and 8 respectively. Figure 9.14 contains the main function that included communication between host and device and the launch of the kernel grid. The kernel is self explanatory.

9.3.2 NVIDIA version 2: a better implementation

Figure 9.15 contains the kernel of the second approach that does block matrix multiplication and utilizes shared memory shared by all the threads of a thread block. Bear in mind that SQRTP $= \sqrt{\text{TPERB}}$, and TPERB is the number of threads per thread block.

The thread block indexed (br, bc) will compute a subblock of matrix C of geometry SQRTP $\times$ SQRTP, indexed also (br, bc). The thread block (br, bc) will need to read all sublocks of a owned by thread blocks indexed $(\text{br}, *)$, where the star $*$ indicates any value from 0 to $n/\text{SQRTP} - 1$. Likewise, the thread block (br, bc) will need to read all sublocks of b owned by thread blocks indexed $(*, \text{bc})$, where the star $*$ indicates any value from 0 to $n/\text{SQRTP} - 1$. Those blocks are "read" by fetching them into the shared memory. Kernel 2, in lines 28-29 will have one block of a identified with (br, j), and one block of b identified with (j, bc) being brought into the shared memory (AA and BB) respectively; the thread (r, c) of thread block (br, bc) will bring one element of block (br, j) of a and one element of block (j, bc) and that element within the corresponding block will have relative row and column index r and c respectively. A barrier synchronization of the threads of the thread block is realized in line 30, before the thread of thread block (br, bc) proceed to the next and compute the block matrix product $a(\text{br}, j) * b((j, \text{bc})$. Overall iterations of lines 27-34 all the matrix products for all values of j from 0 to $n/\text{SQRTP} - 1$ thus computing a sub block of C of geometry SQRTP $\times$ SQRTP.

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <cuda_runtime.h>
#define NPREDEFINED 1024
#define FTYPE float
        /* SQRTP = sqrt(TPERB) */
#define TPERB  1024
#define SQRTP  32
        /* DOUBLECHECK PREV VALUES ARE CONSISTENT */
#define MYMIN(X,Y) ((X)<(Y)?(X):(Y))
#define R 4

void rfillm(FTYPE *z, int n,int type){
 int i,j;
 if (type==0)
   memset(z,0,n*n*sizeof(FTYPE));
 if (type==1)
   for(j=0;j<n;++j)
      for(i=0;i<n;++i){z[i*n+j]=(FTYPE)(j+1);}
 if (type==2)
   for(j=0;j<n;++j)
      for(i=0;i<n;++i){z[i*n+j]=(FTYPE)(2*j+i+1);}
}
void printm(FTYPE *a,FTYPE *b,FTYPE *c,int n,int debug){
 int i,j;
 double sum1=(double)0, sum2=(double)0, sum3=(double)0;
 for(i=0;i<n*n;++i) sum1 += a[i];
 for(i=0;i<n*n;++i) sum2 += b[i];
 for(i=0;i<n*n;++i) sum3 += c[i];
 printf("asig=%f , bsig=%f , csig=%f \n",sum1,sum2,sum3);
if (debug){
 printf("Input A \n");
  for(i=0;(i<n);++i)
   for(j=0;(j<n);++j)
     if((   (i<R) || ((i>n-1-R) && (i<n))) &&
        (   (j<R) || ((j>n-1-R) && (j<n))) )
     printf("%6.1f%c",a[i*n+j],((j<(n-1))?' ':'\n'));
 printf("Input B \n");
  for(i=0;(i<n);++i)
   for(j=0;(j<n);++j)
     if((   (i<R) || ((i>n-1-R) && (i<n))) &&
        (   (j<R) || ((j>n-1-R) && (j<n))) )
     printf("%6.1f%c",b[i*n+j],((j<(n-1))?' ':'\n'));
 printf("Input C \n");
  for(i=0;(i<n);++i)
   for(j=0;(j<n);++j)
     if((   (i<R) || ((i>n-1-R) && (i<n))) &&
        (   (j<R) || ((j>n-1-R) && (j<n))) )
     printf("%6.1f%c",c[i*n+j],((j<(n-1))?' ':'\n'));
}
}
```

Figure 9.12: NVIDIA matrix multiplication version 1 (part1)

```
/* Source:  PG-02829-0001_v7.5 Sep 2015 */
/*  Cuda C Programming, Design Guide, NVIDIA,(pages 39-45) */
/* row-major row-i, col-j is X[i][j] = X.elts + i*X.cols +j */
__global__ void c9mm10 (FTYPE   *A, FTYPE  *B, FTYPE  *C, int n) {
    FTYPE Cv = (FTYPE) 0;
    int r,c,i;
    r = blockIdx.y * blockDim.y + threadIdx.y;
    c = blockIdx.x * blockDim.x + threadIdx.x;
    for(i = 0; i < n ; ++i)
        Cv += A[r * n + i] * B[i * n + c];
    C[r * n + c] = Cv;
}
```

Figure 9.13: NVIDIA matrix multiplication version 1 (part 2: kernel)

```
int  main(int argc, char **argv) {
 FTYPE  *ha,  *hb,  *hc, *da, *db, *dc;
 size_t size;
 int    N,debug=0;
 if (argc==1){N=NPREDEFINED; debug=0;
              printf("Usage: %s N debug(0 or 1)\n",argv[0]);
 }
 else { N=atoi(argv[1]); debug=atoi(argv[2]); }
 size=N*N*sizeof(FTYPE);
 cudaEvent_t start,stop;
 cudaEventCreate(&start);
 cudaEventCreate(&stop );
 ha=(FTYPE*) malloc(size); rfillm(ha,N,1);
 hb=(FTYPE*) malloc(size); rfillm(hb,N,2);
 hc=(FTYPE*) malloc(size); rfillm(hc,N,0); printm(ha,hb,hc,N,debug);
 cudaDeviceSynchronize();
 cudaMalloc(&da,size);
 cudaMalloc(&db,size);
 cudaMalloc(&dc,size);
 cudaMemcpy(da,ha,size,cudaMemcpyHostToDevice);
 cudaMemcpy(db,hb,size,cudaMemcpyHostToDevice);
 cudaMemcpy(dc,hc,size,cudaMemcpyHostToDevice);
 dim3  blck(SQRTP,SQRTP);
 dim3  grid((N+blck.x-1)/blck.x,(N+blck.y-1)/blck.y);
 cudaEventRecord(start);
  c9mm10 <<<grid , blck  >>> (da,db,dc,N);    /* c9mm10 */
 cudaDeviceSynchronize();
 cudaEventRecord(stop);
 cudaMemcpy(hc,dc,size,cudaMemcpyDeviceToHost);
 cudaEventSynchronize(stop);
 FTYPE mill = 0.0f;
 cudaEventElapsedTime(&mill,start,stop);
 cudaDeviceSynchronize();
 printm(ha,hb,hc,N,debug);
 cudaFree(dc); cudaFree(db); cudaFree(da);
 free(hc);     free(hb);     free(ha);
 printf("Elapsed Time %10.5f\n",mill);
 return(0);
}
```

Figure 9.14: NVIDIA matrix multiplication version 1 (part3)

```
/* Source:  PG-02829-0001_v7.5 Sep 2015 */
/*  Cuda C Programming, Design Guide, NVIDIA,(pages 39-45) */
/* row-major row-i, col-j is X[i][j] = X.elts + i*X.cols +j */

  __device__ float gelt(FTYPE *A, int r, int c, int n) {
       return A[r*n + c];
  }
  __device__ void selt(FTYPE *A, int r, int c, float value, int n) {
       A[r*n + c] = value;
  }
  __device__ FTYPE* getsmatrix(FTYPE *A, int r, int c, int n) {
       FTYPE *As;
       As = &A[n * SQRTP * r + SQRTP * c];
       return(As);
   }

__global__ void c9mm11 (FTYPE   *A, FTYPE  *B, FTYPE  *C, int n) {
  __shared__ FTYPE AA[SQRTP][SQRTP], BB[SQRTP][SQRTP];
  FTYPE *aa, *bb, *cc;
  FTYPE Cv = (FTYPE)0;
  int r,c,br,bc,i,j;
  br = blockIdx.y  ; bc = blockIdx.x ;
   r = threadIdx.y ;  c = threadIdx.x;
  cc = getsmatrix(C,br,bc,n);    Cv = 0;
  for (j = 0; j < (n/SQRTP); ++j) {
      aa      =getsmatrix(A,br,j,n ); bb      = getsmatrix(B,j ,bc,n);
      AA[r][c]=gelt(aa,r,c,n)       ; BB[r][c]= gelt(bb,r,c,n);
      __syncthreads();
      for(i = 0; i < SQRTP ; ++i)
        Cv += AA[r][i] * BB[i][c];
      __syncthreads();
    }
    selt(cc,r,c,Cv,n);
}
```

Figure 9.15: NVIDIA matrix multiplication version 2 kernel

9.3.3 NVIDIA version 3: a refined implementation

Figure 9.16 contains the kernel of the third approach that tries to optimize the fetching of the blocks of A and B.

```
/* Source:  PG-02829-0001_v7.5 Sep 2015 */
/*  Cuda C Programming, Design Guide, NVIDIA,(pages 39-45) */
/* row-major row-i, col-j is X[i][j] = X.elts + i*X.cols +j */

  __device__ float gelt(FTYPE *A, int r, int c, int n) {
       return A[r*n + c];
  }
  __device__ void selt(FTYPE *A, int r, int c, float value, int n) {
       A[r*n + c] = value;
  }

  __device__ FTYPE* getsmatrix(FTYPE *A, int r, int c, int n) {
       FTYPE *As;
       As = &A[n * SQRTP * r + SQRTP * c];
       return(As);
   }

__global__ void c9mm12 (FTYPE   *A, FTYPE  *B, FTYPE  *C, int n) {
  __shared__ FTYPE AA[SQRTP][SQRTP], BB[SQRTP][SQRTP];
    FTYPE Cv = (FTYPE) 0;
    int r,c,br,bc,i,j,k;
    int astr,aend,astp,bstr,bstp;
    br = blockIdx.y  ; bc = blockIdx.x ;
     r = threadIdx.y ;  c = threadIdx.x;

    astr = br * n  * SQRTP; aend = astr + n - 1; astp= SQRTP;
    bstr = bc * SQRTP     ;                      ; bstp= n * SQRTP;

    for(i = astr , j = bstr ; i <= aend  ; i += astp , j += bstp ) {
      AA[r][c] = A[i+ n * r + c ]; BB[r][c] = B[j+ n * r + c ];
      __syncthreads();
      for( k=0; k< SQRTP ; ++k)
        Cv  += AA[r][k] * BB[k][c];
      __syncthreads();
    }
    k = n * SQRTP * br + SQRTP * bc ;
    C[k+ n * r + c] = Cv;
}
```

Figure 9.16: NVIDIA matrix multiplication version 3 kernel

9.4 More Matrix multiplication

We first present simple code for column-major matrix multiplication.

We then present the kernels only for five more versions. We do not comment on the code as it resembles the NVIDIA code and draws from it. Furthemore, we present experimental results of all nine kernels: (a) the three NVIDIA ones identified as Kernel 11, Kernel 12, Kernel 13 matching to version 1, version 2, and version 3 presented earlier, and (b) the six new kernels starting with Kernel 1 (column-major), and Kernel 2 through Kernel 6 all row-major.

The experimental results (time in milliseconds) are for three problem sizes of square matrices of order $N = 1024$, $N = 2048$ and $N = 4096$ and two thread block configurations 32×32 and 8×8 mapping to TPERB $= 1024$ and TPERB $= 64$ respectively.

Kernel	$N = 1024$	$N = 2048$	$N = 4096$
1	39.80	295.78	2464.61
2	39.35	287.25	2476.20
3	16.28	129.74	853.71
4	84.62	563.07	4399.87
5	14.72	106.79	853.11
6	76.43	550.03	4398.79
10 (NVIDIA 1)	34.38	398.51	2484.09
11 (NVIDIA 2)	13.35	106.51	854.70
12 (NVIDIA 3)	16.12	128.49	854.87

Figure 9.17: Expiremental results for Matrix Multiply; 32×32 blocks

Kernel	$N = 1024$	$N = 2048$	$N = 4096$
1	60.18	470.57	4126.77
2	60.30	462.82	4109.33
3	28.07	240.36	2070.73
4	34.34	254.00	2134.46
5	28.19	236.85	2036.47
6	34.35	250.72	2124.88
10 (NVIDIA 1)	60.22	458.90	4105.09
11 (NVIDIA 2)	28.22	238.88	2065.06
12 (NVIDIA 3)	28.77	243.37	1928.55

Figure 9.18: Expiremental results for Matrix Multiply; 8×8 blocks

```
__global__ void c9mm01 (FTYPE  *a, FTYPE *b, FTYPE *c, int n) {
  int j= threadIdx.y + blockIdx.y * blockDim.y;
  int i= threadIdx.x + blockIdx.x * blockDim.x;
  FTYPE csum = (FTYPE) 0;
  int k;
  if (i < n && j < n ) {
  for(k=0; k < n ; ++k) {
    csum +=  a[k*n+i] * b[j*n+k];
  }
  c[j*n+i]=csum;
  }
}
```

Figure 9.19: Matrix Multiplication Kernel 1 (column-major)

```
__global__ void c9mm02 (FTYPE  *a, FTYPE *b, FTYPE *c, int n) {
  int i= threadIdx.y + blockIdx.y * blockDim.y;
  int j= threadIdx.x + blockIdx.x * blockDim.x;
  FTYPE csum = (FTYPE) 0;
  int k;

 if (i < n && j < n){
  for(k=0; k < n ; ++k) {
   csum +=  a[i*n+k] * b[k*n+j];
  }
  c[i*n+j]=csum;
 }
}
```

Figure 9.20: Matrix Multiplication Kernel 2

```
__global__ void c9mm03 (FTYPE  *a, FTYPE *b, FTYPE *c, int n) {
  int i= threadIdx.y + blockIdx.y * blockDim.y;
  int j= threadIdx.x + blockIdx.x * blockDim.x;
  __shared__ float ta[SQRTP][SQRTP];/* in general: blockDim.y, blockDim.x */
  __shared__ float tb[SQRTP][SQRTP];/* in general: blockDim.x, blockDim.y */
  FTYPE csum = (FTYPE) 0;
  int k, m,base;
  for (m=0; m < gridDim.x ; ++m){
    base = i*n + m * SQRTP  + threadIdx.x;
    if (base >= (n*n)) {
       ta[threadIdx.y][threadIdx.x] = (FTYPE) 0;
    }
    else {
       ta[threadIdx.y][threadIdx.x] = a[base];
    }
    base = (m * SQRTP  + threadIdx.y)*n+j;
    if (base >= (n*n)) {
       tb[threadIdx.y][threadIdx.x] = (FTYPE) 0;
    }
    else {
       tb[threadIdx.y][threadIdx.x] = b[base];
    }
    __syncthreads();
    for(k=0; k < SQRTP  ; ++k) {
       csum +=  ta[threadIdx.y][k] * tb[k][threadIdx.x];
    }
     __syncthreads();
  }
  if (i< n && j < n ) {
     c[i*n+j]=csum;
  }
}
```

Figure 9.21: Matrix Multiplication Kernel 3

```
__global__ void c9mm04 (FTYPE  *a, FTYPE *b, FTYPE *c, int n) {
  int i= threadIdx.y + blockIdx.y * blockDim.y;
  int j= threadIdx.x + blockIdx.x * blockDim.x;
  __shared__ float ta[SQRTP][SQRTP];
  __shared__ float tb[SQRTP][SQRTP];
  FTYPE csum = (FTYPE) 0;
  int k, m,base;

  for (m=0; m < gridDim.x ; ++m){
    base = i*n + m * SQRTP  + threadIdx.x;
    if (base >= (n*n)) {
       ta[threadIdx.y][threadIdx.x] = (FTYPE) 0;
    }
    else {
       ta[threadIdx.y][threadIdx.x] = a[base];
    }
    base = (m * SQRTP  + threadIdx.y)*n+j;
    if (base >= (n*n)) {
       tb[threadIdx.x][threadIdx.y] = (FTYPE) 0;  /* transpose */
    }
    else {
       tb[threadIdx.x][threadIdx.y] = b[base];  /* transpose */
    }
    __syncthreads();

    for(k=0; k < SQRTP  ; ++k) {
       csum +=  ta[threadIdx.y][k] * tb[threadIdx.x][k];  /* transpose */
    }
     __syncthreads();
  }
  if (i< n && j < n ) {
     c[i*n+j]=csum;
  }
}
```

Figure 9.22: Matrix Multiplication Kernel 4

```
__global__ void c9mm05 (FTYPE  *a, FTYPE *b, FTYPE *c, int n) {
  int i= threadIdx.y + blockIdx.y * blockDim.y;
  int j= threadIdx.x + blockIdx.x * blockDim.x;
  __shared__ float ta[SQRTP][SQRTP];/* in general: blockDim.y, blockDim.x */
  __shared__ float tb[SQRTP][SQRTP];/* in general: blockDim.x, blockDim.y */
  FTYPE csum = (FTYPE) 0;
  int k, m,base;

  for (m=0; m < gridDim.x ; ++m){
    base = i*n + m * SQRTP  + threadIdx.x;
    ((base>=(n*n))?  ta[threadIdx.y][threadIdx.x] = (FTYPE) 0:
                     ta[threadIdx.y][threadIdx.x] = a[base]);
    base = (m * SQRTP  + threadIdx.y)*n+j;
    ((base>=(n*n))?  tb[threadIdx.y][threadIdx.x] = (FTYPE) 0:
                     tb[threadIdx.y][threadIdx.x] = b[base]);
    __syncthreads();
    for(k=0; k < SQRTP  ; ++k) {
       csum +=  ta[threadIdx.y][k] * tb[k][threadIdx.x];
    }
     __syncthreads();
  }
  if (i< n && j < n ) {
     c[i*n+j]=csum;
  }
}
```

Figure 9.23: Matrix Multiplication Kernel 5

```
__global__ void c9mm06 (FTYPE  *a, FTYPE *b, FTYPE *c, int n) {
  int i= threadIdx.y + blockIdx.y * blockDim.y;
  int j= threadIdx.x + blockIdx.x * blockDim.x;
  __shared__ float ta[SQRTP][SQRTP];
  __shared__ float tb[SQRTP][SQRTP];
  FTYPE csum = (FTYPE) 0;
  int k, m,base;

  for (m=0; m < gridDim.x ; ++m){
    base = i*n + m * SQRTP  + threadIdx.x;
    ((base>=(n*n))?  ta[threadIdx.y][threadIdx.x] = (FTYPE) 0:
                     ta[threadIdx.y][threadIdx.x] = a[base]);
    base = (m * SQRTP  + threadIdx.y)*n+j;
    ((base>=(n*n))?  tb[threadIdx.x][threadIdx.y] = (FTYPE) 0:
                     tb[threadIdx.x][threadIdx.y] = b[base]);
    __syncthreads();

    for(k=0; k < SQRTP  ; ++k) {
       csum +=  ta[threadIdx.y][k] * tb[threadIdx.x][k];  /* transpose */
    }
     __syncthreads();
  }
  if (i< n && j < n ) {
     c[i*n+j]=csum;
  }
}
```

Figure 9.24: Matrix Multiplication Kernel 6

Chapter 10

Scan Operations

10.1 Parallel sum

We show how one can implement a parallel sum routine. The sum function (sequential executed on the host) is given in Figure 10.1 and is quite simple.

```
1     // A[0..n-1]
2     sum=0;
3     for(i=0 ; i < n ; ++i)
4        sum = sum + A[i];
5     A[0]=sum;
```

Figure 10.1: Serial code for sum

We first do parallel sum on a block (as in one block containing the currently maximum number of 1024 threads), and then show how to generalize this to multiple threadblocks. However both the number of blocks must be a power of two and so should the number of threads. We did casual checking to confirm that the vectorsize does not need to be a power of two, in general.

10.1.1 One block only parallel sum

The sample code here is for a single thread block and its threades. Thus it works for the maximum of 1024 threads that are defined (or are possible) for a block. The simplicity or not of the code depends on its functionality.

Figure 10.2 offers a while-loop structured Kernel 1.

Figure 10.3, Figure 10.4, Figure 10.5, offer a for-loop structured Kernel 2, Kernel 3, and Kernel 5 (there is no Kernel 4 in this writeup). For multiplication by two the code can use addition, for other multiplications and divisions involving a power of two a shift is preferable to the indicated operation. We tried not to overload the code with optimizations.

The placement of a __syncthreads() could be optimized or corrected.

```
/* Limited to one threadblock */
__global__ void c10ps01(FTYPE  *A, int n) {
  int tid  = threadIdx.x ;
  int step=1, bound = n ;
  float a, b ;

  while (step <= n ) {
   if (tid <= bound/2) {
     (((2*tid)  <=bound)? a= A[ 2*tid]  : a=0.0);
     (((2*tid+1)<=bound)? b= A[ 2*tid+1]: b=0.0);
       __syncthreads();  /* to swap or not swap with previous line? */
        A[tid]= a+b;
   }
   step = step<<1;
   bound= bound>>1;
   }   /* Psum in A[0] */
}
```

Figure 10.2: Kernel 1 : while loop

```
/* Limited to one threadblock */
__global__ void c10ps02(FTYPE  *A, int n) {
  int tid  = threadIdx.x ; /* Limited use: Threads of a */
  int step = 1 ;           /* 1D block; at most 1024    */

  for(step = 1 ; step < blockDim.x ; step *= 2 ) {
     if ((tid % (2*step)) == 0) {
          A[tid] += A[tid+step];
     }
     __syncthreads();
  }  /* Psum in A[0] */
}
}
```

Figure 10.3: Kernel 2 : for loop

```
/* Limited to one threadblock */
__global__ void c10ps03(FTYPE  *A, int n) { /* Limited use: Threads of a */
  int tid  = threadIdx.x ;                  /* /* 1D block; at most 1024 */
  int step = 1 , src ;

  for(step = 1 ; step < blockDim.x ; step *= 2 ) {
     src = tid * 2 * step ;
     if ( src < blockDim.x ) {
          A[src] += A[src+step];
     }
     __syncthreads();
  }  /* Psum in A[0] */
}
```

Figure 10.4: Kernel 3 : for loop alternaive

```
1 /* Limited to one threadblock */
2 __global__ void c10ps05(FTYPE  *A, int n) {
3   int pid  = threadIdx.x ;
4   int step = 1 ;
5
6   for(step = blockDim.x /2 ; step  > 0  ; step >>= 1 ) {
7      if ( pid < step ) {
8           A[pid] += A[pid+step];
9      }
10     __syncthreads();
11  } /* Psum in A[0] */
12 }
```

Figure 10.5: Kernel 5 : for loop alternaive

10.1.2 More generic parallel sum

We show how to generalize the previous kernels to do parallel sum on multiple threadblocks. However both the number of blocks must be a power of two and so should the number of threads.

Kernel 20 of Figure 10.6 is Kernel 2. Whereas in Kernel 2 the sum of $A[0..n-1]$ was stored in $A[0]$ this is not the case here. Every block and in particular the block with blockIdx.x replaces $A[\text{blockIdx.x}]$ with the parallel sum of the values assigned to the threads of the block. All this values are transferred from global memory (A is stored there) into the shared cache memory. Line 17 might cause problems including the two synchronizations in lines 14 and 19. A further ironing out of sychronization issues is known later in Kernel 23 (for line 17) and Kernel 24 (for lines 14 and 19).

```
1  /* Modified kernel ps02; assume #TPERB is power of 2, *
2   * and number of blocks also a power of two           *
3   * parallel sum is NOT readily available it is the    *
4   * sum_i i=0 to blockDim.x-1 of A[i]                  *
5   * line shmem[tid} += might cause problems; fix is    *
6   * available in kernel ps23  ...                      */
7  __global__ void c10ps20(FTYPE  *A, int n) {
8    int tid  = threadIdx.x ;
9    int step = 1 ;
10   int pid  = blockIdx.x * blockDim.x + threadIdx.x ;
11   __shared__ FTYPE shmem[TPERB];
12
13   shmem[tid] = A[pid];
14   __syncthreads();
15   for(step = 1 ; step < blockDim.x ; step += step ) {
16      if ((tid % (2*step)) == 0) {
17           shmem[tid] += shmem[tid+step];
18      }
19      __syncthreads();
20   }  /* Parallel sum is sum_i < blockDim.x A[i] */
21   if (tid==0) A[blockIdx.x] = shmem[0];
22 }
```

Figure 10.6: Kernel 20 : generalizing kernel 2

In Kernel 21 of Figure 10.7 we avoid the modulo operation.

In Kernel 22 of Figure 10.8 we provide an alternative for-loop configuration.

In Kernel 23 of Figure 10.9 we first fix the issue of parallel sum as described in the case of a PRAM algorithm. The parallel sum of n numbers using n processors utilizes only half of those processors. In the case of a CUDA-based approach, the number of thread blocks utilized is half of the grid blocks. This is taken care in Kernel 23. In Kernel 23 of Figure 10.9 we also slightly modify line 15 of Kernel 22. Without this modification the code would have been buggy because of timing issues. This change can be propagated to the other Kernel as well.

In Kernel 24 of Figure 10.10 we have rearranged and used only one __syncthreads() function call that is embedded in line 13 rather than in line 10 and line 18 (or rather after the right brace of line 17). This change can propagate cautiously to the other codes as well.

We provide a comparative study of the five kernels Kernel 20, through Kernel 24. This is depicted in Figure 10.11. Note that parallel sum beyond around $n = 2^{23}$ leads to instability of the computed value of the sum. (Unless we compute a sum of ones or twos.) All timings are in milliseconds.

These experimental results are for the platform of Figure 7.13 also shown here (Quadro P600).

```
/* Assumes powers of two (see ps20 kernel info *
 * Derives from kernel ps03 that dealt with    *
 * only one block; see there for note on shmem*/
__global__ void c10ps21(FTYPE  *A, int n) {
  int tid  = threadIdx.x ;
  int step = 1 , src ;
  int pid  = blockIdx.x * blockDim.x + threadIdx.x ;
  __shared__ FTYPE shmem[TPERB];

  shmem[tid] = A[pid];
  __syncthreads();

  for(step = 1 ; step < blockDim.x ; step +=step ) {
     src = tid * 2 * step ;
     if ( src < blockDim.x ) {
          shmem[src] += shmem[src+step];
     }
     __syncthreads();
  }  /* Parallel sum is sum_i < blockDim.x A[i] */
  if (tid==0) A[blockIdx.x] = shmem[0];
}
```

Figure 10.7: Kernel 21 : generalizing kernel 3

For an alternative platform see Figure 10.13 for its configuration and Figure 10.14 for additional experimental results.

Other optimizations are possible: NVIDIA CUDA manual describe them. They can result in a three-fold or better increase in performance.

```
/*  Read info re kernel ps20; all applicable  *
 *  but based on simplified kernel ps05       */
/*  Read also ps24 re realignment of __sync   */
__global__ void c10ps22(FTYPE  *A, int n) {
  int tid  = threadIdx.x ;
  int step = 1 ;
  int pid  = blockIdx.x * blockDim.x + threadIdx.x ;
  __shared__ FTYPE shmem[TPERB] ;

  shmem[tid] = A[pid];
  __syncthreads();

  for(step = blockDim.x /2 ; step  > 0  ; step >>= 1 ) {
     if ( tid < step ) {
          shmem[tid] += shmem[tid+step] ;
     }
     __syncthreads() ;
  }  /* Parallel sum is sum_i < blockDim.x A[i] */
  if (tid==0) A[blockIdx.x] = shmem[0] ;
}
```

Figure 10.8: Kernel 22 : generalizing kernel 5

```
__global__ void c10ps23(FTYPE  *A, int n) {
  int tid  = threadIdx.x ;
  int step = 1 ;
  int pid  = blockIdx.x * (2*blockDim.x) + threadIdx.x ;
  __shared__ FTYPE shmem[TPERB] ;
  FTYPE temp;

  temp= ((pid < n)? A[pid]: 0);
  if ((pid+blockDim.x) <n)
     temp += A[pid+blockDim.x];
  shmem[tid] = temp;
  __syncthreads();

  for(step = blockDim.x /2 ; step  > 0  ; step >>= 1 ) {
     if ( tid < step ) {
          shmem[tid] =temp= temp+ shmem[tid+step] ;
     }
     __syncthreads() ;
  }
  if (tid==0) A[blockIdx.x] = shmem[0];
}
```

Figure 10.9: Kernel 23 : for loop alternaive

```
/* This is kernel ps21 with realignment        *
 * of __syncthreads(); under testing          */
__global__ void c10ps24(FTYPE  *A, int n) {
  int tid  = threadIdx.x ;
  int step = 1 , src ;
  int pid  = blockIdx.x * blockDim.x + threadIdx.x ;
  __shared__ FTYPE shmem[TPERB];

  shmem[tid] = A[pid];
  /* __syncthreads(); */

  for(step = 1 ; step < blockDim.x ; step +=step ) {
     __syncthreads();
     src = tid * 2 * step ;
     if ( src < blockDim.x ) {
          shmem[src] += shmem[src+step];
     }
  }  /* Parallel sum is sum_i < blockDim.x A[i] */
  if (tid==0) A[blockIdx.x] = shmem[0];
}
```

Figure 10.10: Kernel 24 : for loop alternaive

Version	$n = 2^{20}$	$n = 2^{21}$	$n = 2^{22}$
Kernel 20	1.35	2.67	5.33
Kernel 21	0.81	1.62	3.22
Kernel 22	0.64	1.25	2.48
Kernel 23	0.61	1.21	2.40
Kernel 24	0.76	1.51	3.01

Figure 10.11: Parallel Sum Experimental Results

```
Number of Devices is 1
Device Number is 0
Device Name   is Quadro P600
Total Global M : 2095841280
Shared Mem/Block 49152
Registers /Block 65536
WarpSize         32
memPitch         2147483647
maxThreads/Block 1024
maxThreads Dim   2147483647 65535 65535
TotConst  Memory 65536
Major Revision   6
Minor Revision   1
Clock Rate       1556500
deviceOverlap    1
SM in device     3
Compute  Mode    0
Concurrent Kernl 1
```

Figure 10.12: Configuration of CUDA platform

```
Number of Devices is 1
Device Number is 0
Device Name   is Tesla P100-PCIE-16GB
Shared Mem/Block 49152
Registers /Block 65536
WarpSize         32
memPitch         2147483647
maxThreads/Block 1024
maxThreads Dim   2147483647 65535 65535
TotConst  Memory 65536
Major Revision   6
Minor Revision   0
Clock Rate       1328500
deviceOverlap    1
SM in device     56
Compute  Mode    0
Concurrent Kernl 1
```

Figure 10.13: Configuration of alternative CUDA platform

Version	$n = 2^{20}$	$n = 2^{21}$	$n = 2^{22}$
Kernel 20	0.75	1.45	2.83
Kernel 21	0.36	0.66	1.30
Kernel 22	0.30	0.56	1.10
Kernel 23	0.32	0.60	1.15
Kernel 24	0.35	0.66	1.30

Figure 10.14: Parallel Sum Experimental Results (alternative platform)

Index