Parallel Computing - Notes - v0.12.0-dev

260236

December 2024

Preface

Every theory section in these notes has been taken from the sources:

- Validity of the single processor approach to achieving large scale computing capabilities. [1]
- Introduction to parallel algorithms, Carnegie Mellon University. [2]
- Course slides. [3]
- Reevaluating Amdahl's law. [4]
- OpenMP by Example, Johnston Hans. [5]
- Introduction to parallel computing, volume 110. [6]
- NVIDIA Tesla: A unified graphics and computing architecture. [7]
- Structured Parallel Programming: Patterns for Efficient Computation. [8]
- The critical directive, OpenMP, Microsoft. [9]
- Multithreading Architecture. [10]
- Lecture 5, Synchronization I - University of Michigan, EECS Department, Prof. Ronald Dreslinski. [11]
- Cache, Wikipedia. [12]

About:

GitHub repository

These notes are an unofficial resource and shouldn't replace the course material or any other book on parallel computing. It is not made for commercial purposes. I've made the following notes to help me improve my knowledge and maybe it can be helpful for everyone.

As I have highlighted, a student should choose the teacher's material or a book on the topic. These notes can only be a helpful material.

Contents

1 PRAM

1.1 Prerequisites

Before we introduce the PRAM model, we need to cover some useful topics.

- A **Machine Model** describes a "machine". It gives a value to the operations on the machine. It is necessary because: it makes it easy to deal with algorithms; it achieves complexity bounds; it analyses maximum parallelism.
- A **Random Access Machine (RAM)** is a model of computation that describes an abstract machine in the general class of register machines. Some features are:
  - **Unbounded** number of local memory cells;
  - Each memory cell can hold an integer of **unbounded** size;
  - Instruction set includes simple operations, data operations, comparator, branches;
  - All operations take **unit time**;
  - The definition of **time complexity** is the number of instructions executed;
  - The definition of **space complexity** is the number of memory cells used.

1.2 Definition

Definition 1: PRAM

A **parallel random-access machine (parallel RAM or PRAM)** is a **shared-memory abstract machine**. As its name indicates, the PRAM is intended as the parallel-computing analogy to the random-access machine (RAM) (not to be confused with random-access memory). In the same way that the RAM is used by sequential-algorithm designers to model algorithmic performance (such as time complexity), the **PRAM is used by parallel-algorithm designers to model parallel algorithmic performance** (such as time complexity, where the number of processors assumed is typically also stated).

The PRAM model has many interesting features:

- **Unbounded collection of RAM processors** (P_0, P_1, and so on);
- Processors don't have tape;
- Each processor has **unbounded registers**;
- **Unbounded** collection of **share memory cells**;
- All **processors** can **access all memory cells in unit time**;
- All **communication via shared memory**.

1.3 How it works

1.3.1 Computation

A single **processor** of the PRAM, at each computation, is **composed of 5 phases** (carried out in parallel by all the processors):

1. **Reads a value from one of the cells** $X(1), \ldots, X(N)$
2. Reads one of the shared memory cells $A(1), A(2), \ldots$
3. Performs some internal computation
4. **May write into one of the output cells** $Y(1), Y(2), \ldots$
5. May write into one of the shared memory cells $A(1), A(2), \ldots$

1.3.2 PRAM Classificiation

During execution, a subset of processors may remain idle. Also, some processors can read from the same cell at the same time (not really a problem), but they could also try to write to the same cell at the same time (**write conflict**). For these reasons, PRAMs are classified according to their read/write capabilities (realistic and useful):

- **Exclusive Read (ER)**. All processors can simultaneously read from distinct memory locations.
- **Exclusive Write (EW)**. All processors can simultaneously write to distinct memory locations.
- **Concurrent Read (CR)**. All processors can simultaneously read from any memory location.
- **Concurrent Write (CW)**. All processors can write to any memory location.

  But what value is ultimately written?

  It depends on the mode we choose:

  - **Priority Concurrent Write**. Processors have priority based on which value is decided, the **highest priority is allowed to complete write**.
  - **Common Concurrent Write**. All processors are allowed to complete write **if and only if all the value to be written are equal**. Any **algorithm** for this model has to **make sure that this condition is satisfied**. **Otherwise**, the **algorithm is illegal** and the **machine state will be undefined**.
  - **Arbitrary/Random Concurrent Write**. One **randomly** chosen **processor is allowed to complete write**.

1.3.3 Strengths of PRAM

PRAM is attractive and important model for designers of parallel algorithms because:

- It is **natural**. The number of operations executed per one cycle on P processors is at most P (equal to P is the ideal case).
- It is **strong**. Any processor can read/write any shared memory cell in unit time.
- It is **simple**. It abstracts from any communication or synchronization overhead, which makes the complexity and correctness of PRAM algorithm easier.
- It can be used as a **benchmark**. If a problem has no feasible/efficient solution on PRAM, it has no feasible/efficient solution for any parallel machine.

1.3.4 How to compare PRAM models

Consider two generic PRAMs, models A and B. Model A is **computationally stronger** than model B ($A \geq B$) **if and only if any algorithm** written for model B will **run unchanged** on model A in the **same parallel time** and with the **same basic properties**.

However, there are some useful metrics that can be used to compare models:

- **Time to solve problem** of **input size** n on **one processor**, using **best sequential algorithm**:
$$T^* (n) \tag{1}$$
- **Time** to solve problem of input size n on **p processors**:
$$T_p (n) \tag{2}$$
- **Speedup on p processors**:
$$\mathrm{SU}_p (n) = \frac{T^* (n)}{T_p (n)} \tag{3}$$
- **Efficiency**, which is the work done by a processor to solve a problem of input size n divided by the work done by p processors:
$$E_p (n) = \frac{T_1 (n)}{pT_p (n)} \tag{4}$$
- **Shortest run time** on any process p:
$$T_\infty (n) \tag{5}$$

- **Cost**, equal to processors and time:

$$C(n) = P(n) \cdot T(n) \tag{6}$$

- **Work**, equal to the total **number of operations**:

$$W(n) \tag{7}$$

Some properties on the metrics:

- The time to solve a problem of input n on a single processor using the best sequential algorithm *is not equal to* the time to solve a problem of input n in parallel using one of the p processors available. In other words, **the problem should not be solvable on a single processor on a parallel machine** (otherwise, what would be the point of using a parallel model?)

$$T^* \neq T_1$$

- $\mathrm{SU}_P \leq P$
- $\mathrm{SU}_P \leq \dfrac{T_1}{T_\infty}$
- $E_p \leq 1$
- $T_1 \geq T^* \geq T_p \geq T_\infty$
- $T^* \approx T_1 \Rightarrow E_p \approx \dfrac{T^*}{pT_p} = \dfrac{\mathrm{SU}_p}{p}$
- $E_p = \dfrac{T_1}{pT_p} \leq \dfrac{T_1}{pT_\infty}$
- $T_1 \in O(C), T_p \in O\left(\dfrac{C}{p}\right)$
- $W \leq C$
- $p \approx \text{AREA} \quad W \approx \text{ENERGY} \quad \dfrac{W}{T_p} \approx \text{POWER}$

1.4 MVM algorithm

The **Matrix-Vector Multiply (MVM) algorithm** consists of four steps:

1. **Concurrent read of vector** $X\,(1:n)$ (transfer N elements);
2. **Simultaneous reads of different sections of the general matrix** A (transfer $\frac{n^2}{p}$ elements to each processor);
3. **Compute** $\frac{n^2}{p}$ operations per processor;
4. **Simultaneous writes** (transfer $\frac{n}{p}$ elements from each processor).

Let i be the processor index, so the MVM algorithm is simply written as:

```
1 GLOBAL READ (Z ← X)
2 GLOBAL READ (B ← A_i)
3 COMPUTE (W := BZ)
4 GLOBAL WRITE (W → y_i)
```

Algorithm 1: Matrix-Vector Multiply (MVM)

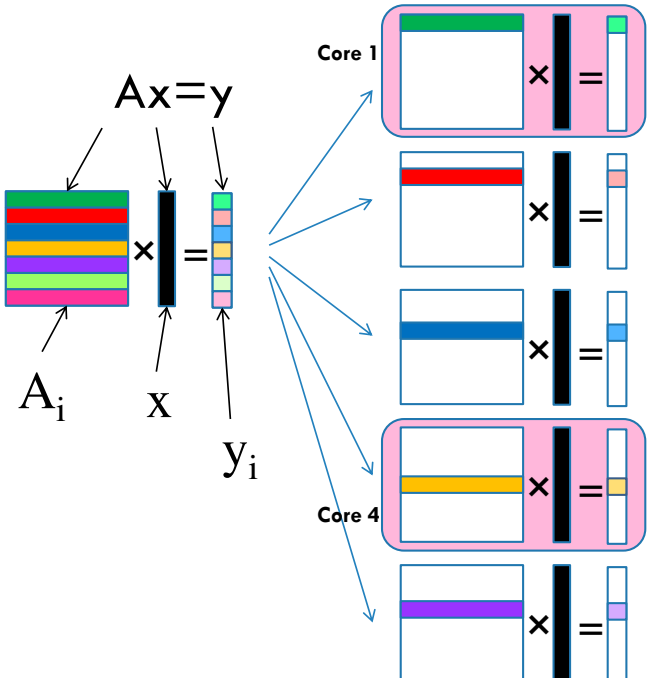


Figure 1: Example of MVM algorithm.

The performance of the MVM algorithm is as follows:

- The **time to solve** a problem of size n^2 is equal to the big O of the squared size of the problem as input divided by the number of processors available:
$$T_p\left(n^2\right) = O\left(\frac{n^2}{p}\right)$$
- The **cost** is equal to the number of processors and the time it takes to solve the problem. So it is quite trivial:
$$C = O\left(\cancel{P} \cdot \frac{n^2}{\cancel{p}}\right) = O\left(n^2\right)$$
- The **work** is equal to the cost, and the **linear power** P is equal to the ratio of work and time to solve the problem on p processors:
$$W = C \qquad \frac{W}{T_p} = P$$
- The **perfect efficiency** is equal to:
$$E_p = \frac{T_1}{pT_p} = \frac{n^2}{p\frac{n^2}{p}} = 1$$

1.5 SPMD sum

The **Single Program Multiple Data (SPMD)** is a term that has been used to **describe computational models** for exploiting parallelism, where **multiple processors work together to execute a program to get results faster**.

In this section, we will see an SPMD approach on a Parallel Random Access Machine (PRAM). We will introduce one of the most common and simple mathematical operations: the sum.

The following pseudocode takes as **input an array** of size $n = 2^k$. In this case, n is a power of 2 because it ensures that the array can be evenly divided at each step of the computation. The value k is the number of iterations or levels of the summation process.

```
1  BEGIN
2      GLOBAL READ (A ← A(I))
3      GLOBAL WRITE (A → B(I))
4      FOR H = 1 : K
5          IF i ≤ n ÷ 2^h THEN BEGIN
6              GLOBAL READ (X ← B(2i - 1))
7              GLOBAL READ (Y ← B(2i))
8              Z := X + Y
9              GLOBAL WRITE (Z → B(i))
10         END
11     IF I = 1 THEN
12         GLOBAL WRITE (Z → S)
13 END
```

Algorithm 2: Single Program Multiple Data (SPMD) sum

- First, read the entire input array `A` and copy the read data to another array `B`.
- Loop over `h` (`1 to k`). In each iteration, for each index i less than or equal to $n \div 2^h$, read values from array `B` at positions $2i - 1$ and $2i$; sum these values (and store the result in `Z`) and store the result (`Z`) back into `B(`i`)`.
- Once all iterations are complete, the final sum is stored in a variable `S`.

For example, if $n = 8$, then k would be 3, meaning that the algorithm will run for 3 iterations to sum all the elements in parallel.

h	i	adding
1	1	1,2
	2	3,4
	3	5,6
	4	7,8
2	1	1,2
	2	3,4
3	1	1,2

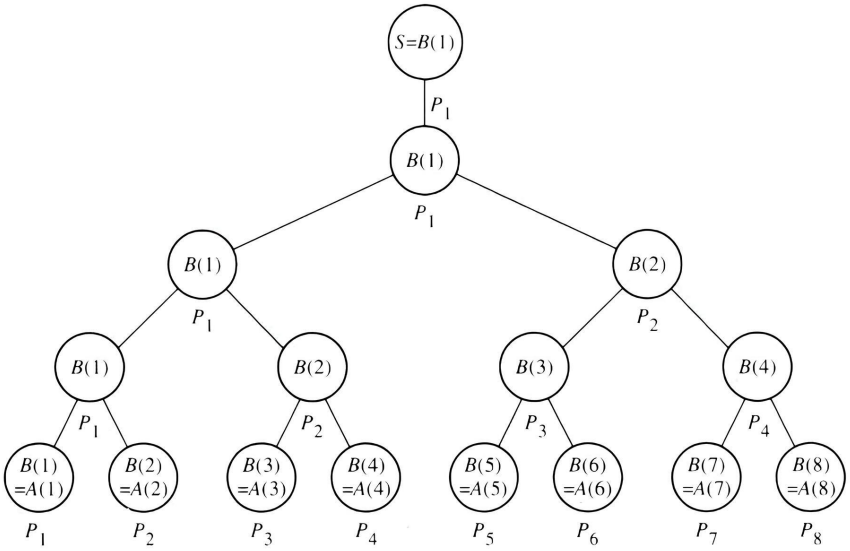
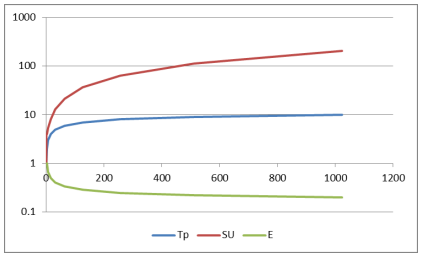


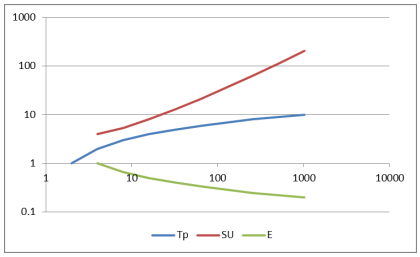
Figure 2: Computation of the sum of eight elements on a PRAM with eight processors. Each internal node represents a sum operation. The specific processor executing the operation is indicated below each node.

Performance of sum

When the size of the array is equal to the number of processors ($N = P$), the **speedup and efficiency decrease**:

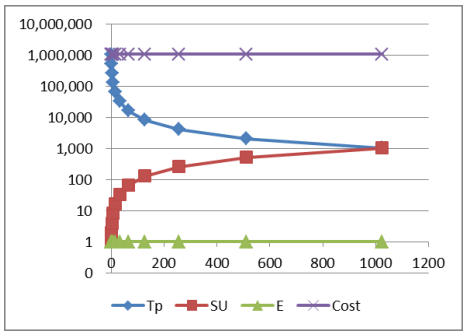
- $T^*(N) = T_1(N) = N$
- $T_{N=P}(N) = 2 + \log N$
- $\mathrm{SU}_P = \dfrac{N}{2 + \log N}$
- $T^*(N) = P \cdot (2 + \log N) \approx N \log N$
- $E_p = \dfrac{T_1}{pT_p} = \dfrac{N}{N \log N} = \dfrac{1}{\log N}$





If the size of the array is much larger than the number of processors ($N \gg P$), the **speedup and power are linear**, the **cost is fixed** and the **efficiency is maximum (equal to 1)**:

- $T^*(N) = T_1(N) = N$
- $T_p(N) = \frac{N}{p} + \log p$
- $\mathrm{SU}_P = \frac{N}{\frac{N}{p} + \log p} \approx P$
- $\mathrm{COST} = p\left(\frac{N}{p} + \log p\right) \approx N$
- $\mathrm{WORK} = N + P \approx N$
- $E_p = \frac{T_1}{pT_p} = \frac{N}{p\left(\frac{N}{p} + \log p\right)} \approx 1$



$n = 1'000'000$

Example 1

Refer to Figure 2 (page 12), the performance metrics are:

- $T_8 = 5$
- $C = 8 \cdot 5 = 40$ (could do 40 steps)
- $W = 2n = 16$ (16 on 40, wasted 24)
- $E_p = \frac{2}{\log n} = \frac{2}{3} = 0.67$
- $\frac{W}{C} = \frac{16}{40} = 0.4$

There is also the **Prefix Sum**, which takes **advantage of idle processors in the sum**. It computes all prefix sums:

$$S_i = \sum_{1}^{i} a_j \qquad a_1, \quad a_1 + a_2, \quad a_1 + a_2 + a_3$$

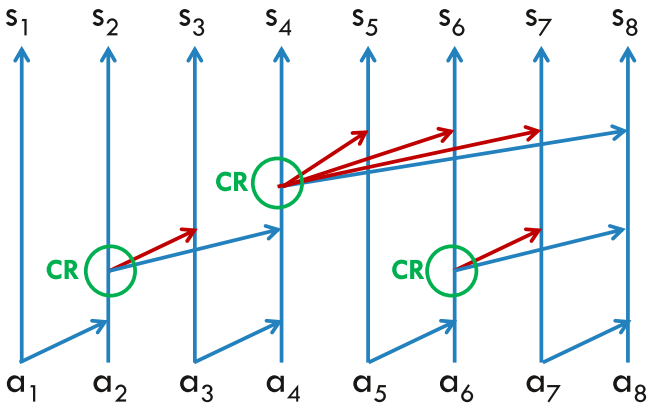


Figure 4: Prefix sum.

1.6 MM algorithm

The **Matrix Multiply (MM) algorithm** consists of three steps:

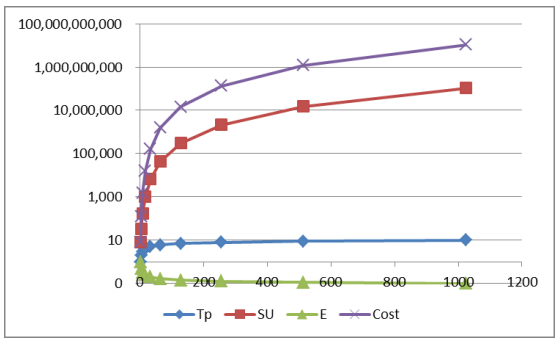
1. **Compute the two matrices** $A_{i,l}$ and $B_{l,j}$, so use the concurrent read.
2. Make the **sum**.
3. **Store** the result using exclusive write.

```
1 BEGIN
2     T_{i,j,l} = A_{i,l} B_{l,j}
3     FOR = H = 1 : K
4         IF l ≤ n ÷ 2^h THEN
5             T_{i,j,l} = T_{i,j,2l-1} + T_{i,j,2l}
6     IF l = 1 THEN
7         C_{i,j} = T_{i,j,1}
8 END
```

Algorithm 3: Matrix Multiply (MM)

Performance of MM

- $T_1 = n^3$
- $T_{p=n^3} = \log n$
- $\text{SU} = \dfrac{n^3}{\log n}$
- $\text{Cost} = n^3 \log n$
- $E_p = \dfrac{T_1}{pT_p} = \dfrac{1}{\log n}$



1.7 PRAM variants and Lemmas

The PRAM model presented here is one of the most commonly used. However, there are other important variants:

- PRAM model with a **limited number of shared memory cells** (small memory PRAM). If the input data set exceeds the capacity of the shared memory, the I/O values can be evenly distributed among the processors.
- PRAM model with **limited number of processors** (small PRAM). If the number of execution threads is higher, processors can interleave multiple threads.
- PRAM model with **limited size of one machine word**.
- PRAM model with **access conflicts handling**. These are restrictions on simultaneous access to shared memory cells.

Lemma 1. *Assume $P' < P$ and same size of shared memory. Any problem that can be solved for a P processor PRAM in T steps can be solved in a P' processor PRAM in:*

$$T' = O\left(\frac{TP}{P'}\right) \tag{8}$$

Proof. Partition P is simulated processors into P' groups of size $\frac{P}{P'}$ each. Associate each of the P' simulating processors with one of these groups. Each of the simulating processors simulates one step of its group of processors by:

- Executing all their read and local computation substeps first;
- Executing their write substeps then.

QED

Lemma 2. *Assume $M' < M$. Any problem that can be solved for a P processor and M-cell PRAM in T steps can be solved on a $\max(P, M')$-processors M'-cell PRAM in $O\left(\frac{TM}{M'}\right)$ steps.*

Proof. Partition M simulated shared memory cells into M' continuous segments S, of size $\frac{M}{M'}$ each. Each simulating processor P'_I $(1 \leq I \leq P)$, will simulate processor P_I of the original PRAM. Each simulating processor P'_I $(1 \leq I \leq M')$, stores the initial contents of S_I into its local memory and will use $M'[I]$ as an auxiliary memory cell for simulation of accesses to cell of S_I.
Simulation of one original read operation:

```
1 EACH P'_I (I = 1, ..., max(P, M')) REPEATS FOR K = 1, ..., M/M'
2     WRITE THE VALUE OF THE K-TH CELL OF S_I INTO M'[I] (I = 1, ..., M')
3     READ THE VALUE WHICH THE SIMULATED PROCESSOR P_I (I = 1, ..., P)
      WOULD READ IN THIS SIMULATED SUBSTEP, IF IT APPEARED IN THE
      SHARED MEMORY
```

The local computation substep of P_I $(I = 1, \ldots, P)$ is simulated in one step by P'_I. SImulation of one original write operation is analogous to that of read.
QED

1.8 PRAM implementation

The PRAM is an ideal model for creating parallel algorithms. Now we look at "*is it really implementable?*" The short answer is yes.

The longest answer is the following. There are already some examples of PRAM being converted to real machine models, such as Explicit Multi-Threading (XMT), Rigel, Tilera, etc. If conversion is not easy or possible, the implementation can be "*direct*":

- The concurrent read is implemented as a detect-and-multicast technique.
- The concurrent write is implemented depending on the end result we want to achieve. Fetch-and-operate and prefix-sum are examples of serialized writing; otherwise, the CRCW technique is used:
  - Common CRCW: detect and merge
  - Priority CRCW: detect-and-priorities
  - Arbitrary CRCW: arbitrary

Example 2: Boolean DNF (sum of products) common CRCW

A logical formula is considered to be in DNF if it is a disjunction of one or more conjunctions of one or more literals.
Consider X as the sum of products of AND/OR operations:

$$X = a_1 b_1 + a_2 b_2 + \dots$$

The PRAM code, with X initialized to 0 and task index equal to \$, is:

```
if (a$b$) X = 1;
```

The common result is that not all processors write X and those that do write 1. The time complexity is $O(1)$. It works on common, priority and arbitrary CRCW.

Despite the previous example, exists also the PRAM SoP for the concurrent write. Let boolean X as:

$$X = a_1 b_1 + a_2 b_2 + \dots$$

The PRAM algorithm is:

```
if (aibi) X = 1;
```

Where all cores which write into X, **write the same value**.

✔ PRAM advantages

- Large body of algorithms.
- Easy to think about it.
- Sync version of shared memory. It eliminates sync and common issues, allows focus on algorithms, but allows adding these issues and allows conversion to async versions.
- Exists architectures for both synch (PRAM) and async (SM) model.
- PRAM algorithms can be mapped to other models.

1.9 Amdahl's and Gustafson's Laws

The **Amdahl's Law** is a formula which gives the **theoretical speedup in latency of the execution of a task at fixed workload that can be expected of a system whose resources are improved**. The law can be stated as:

Definition 2: Amdahl's Law

The overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used.

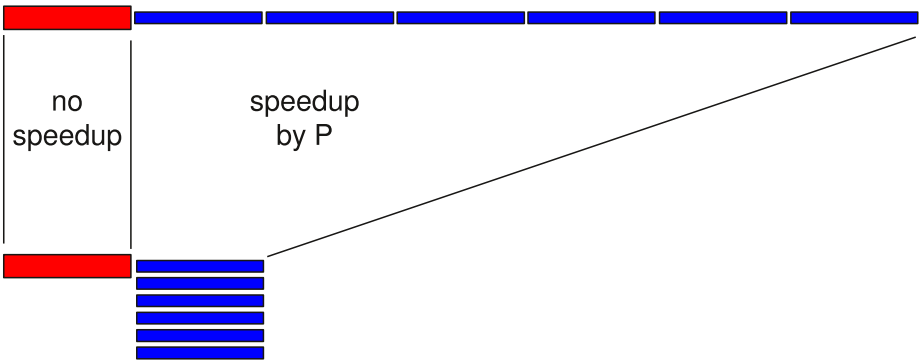
In practice, Amdahl's law says that the computation consists of interleaved segments of two types:

1. **Serial segments** (which cannot be parallelized);
2. **Parallelizable segments**.

Therefore, the metrics we can obtain are the time on P processors metric, that it is greater than the fraction of time on a processor divided by the processors P, and the speedup metric, that it is less than the number of processors P:

$$T_P > \frac{T_1}{P} \qquad SU < P$$

Graphically, we can see a fixed part of the line, which is the **serial segment** (no speedup), and a set of instructions that can be **parallelized** (the sum of these segments is equal to the unit time 1).



Furthermore, if we identify the parallelizable segment as f and the serial segment as $1 - f$, we obtain the following expressions:

$$\begin{aligned} SU\,(P, f) &= \frac{T_1}{T_P} = \frac{T_1}{T_1 \cdot (1-f) + \frac{T_1 \cdot f}{P}} = \frac{1}{(1-f) + \frac{f}{P}} \\ \lim_{P \to \infty} SU\,(P, f) &= \frac{1}{1-f} \end{aligned} \tag{9}$$

In the following figure we can see the speedup with parameter f. Note the pessimism: for a problem with inherent $f = 90\%$, there is no point in using more than 10 processors.

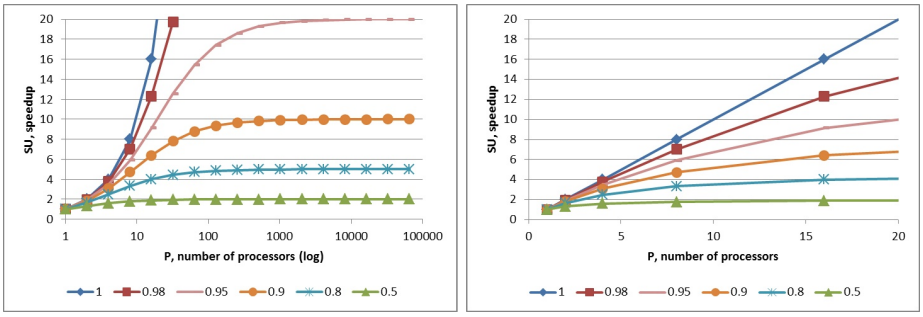


Figure 5: Amdahl's law, $SU(P)$, parameter f.

The original paper presenting Amdahl's Law [1] can be viewed by clicking on the link below or by scanning the QR code.

Amdahl's Law

Amdahl's law applies only to the cases where the problem size is fixed. In practice, as more computing resources become available, they tend to get used on larger problems (larger datasets), and the time spent in the parallelizable part often grows much faster than the inherently serial work. In this case, **Gustafson's law gives a less pessimistic and more realistic assessment of the parallel performance**. [8]

Gustafson's Law gives the speedup in the execution time of a task that theoretically gains from parallel computing, using a hypothetical run of the task on a single-core machine as the baseline. To put it another way, it is the **theoretical "slowdown" of an already parallelized task if running on a serial machine**.

Against Amdahl's law, Gustafson suggests the following ideas:

- Portion f is not fixed;
- The absolute serial time is fixed;
- Parallel problem size is increased to exploit more processors;
- Fixed serial time (s of total) and fixed parallel time ($1-s$ of total) are invariants;
- **Fixed time model** and not fixed size model (as Amdahl's law):

$$SU(P) = \frac{T_1}{T_P} = \frac{s + P \cdot (1-s)}{s + (1-s)} = s + P \cdot (1-s) \tag{10}$$

Gustafson's law suggests a **linear speedup** and is **empirically applicable to highly parallel algorithms**.

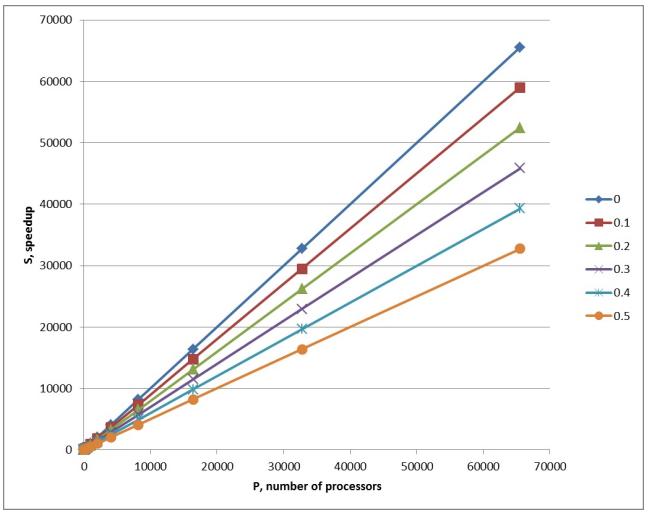


Figure 6: Gustafson's law.

Amdahl's Law states that as computing power increases, computational requirements remain the same. In other words, the **analysis of the same data will take less time with more computing power**.

Gustafson, on the other hand, argues that **more computing power leads to more careful and complete analysis of the data**. Where it would not have been possible or practical to simulate the impact of nuclear denotation on every building, car, and their contents (including furniture, structural strength, etc.) because such a calculation would have taken more time than was available to provide an answer, the increase in computing power will prompt researchers to add more data to more fully simulate more variables, giving a more accurate result.

The original paper presenting Gustafson's Law [4] can be viewed by clicking on the link below or by scanning the QR code.

Gustafson's Law

2 Fundamentals of architecture

The main purpose of this chapter is to introduce some basics of parallel computing theory. It will introduce the simplest and trivial processor and the more complex and efficient variants. The topics introduced are explained in a simple way and without any deepening, because it is only an introduction. For those who have studied computer science, this chapter might be a little boring and you might notice that some topics are explained in a simplistic way.

2.1 Introduction

2.1.1 Simplest processor

Inside a computer, a processor executes instructions.

- **Fetch/Decode**: Determine which instruction to run next;
- **ALU** (execution unit): Performs the operation described by an instruction, which may change values in the processor's registers or the computer's memory;
- **Registers**: maintain program state, store values of variables used as inputs and outputs to operations.

The simplest and most basic processor executes **one instruction per clock cycle**.

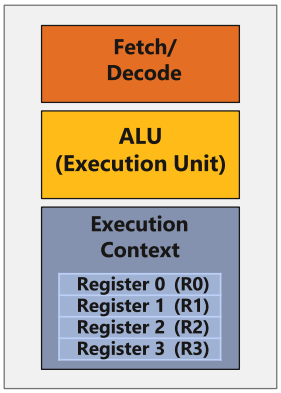


Figure 7: The simplest and most basic processor.

2.1.2 Superscalar processor

A more "complex" and realistic model is the **superscalar processor**. This **processor can decode and execute up to two instructions per clock**. The execution is slightly different from the simplest processor. The **processor automatically finds independent instructions in an instruction sequence and can execute them in parallel on multiple execution units**.

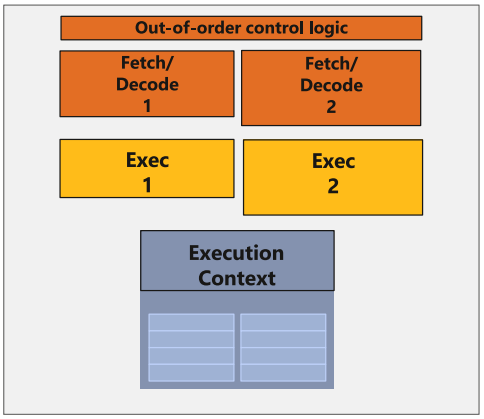


Figure 8: The superscalar processor.

The superscalar processor takes advantage of **Instruction-Level Parallelism (ILP)**[1] within an instruction stream.

- Processing **different instructions** from the same instruction stream **in parallel** (**within a core**).
- **Parallelism is automatically detected by the hardware during execution**.

[1] Instruction-level parallelism (ILP) is the parallel or simultaneous execution of a sequence of instructions in a computer program. More specifically ILP refers to the average number of instructions run per step of this parallel execution.

2.1.3 Single Instruction, Multiple Data (SIMD) processor

Adding execution units (ALUs) to the simplest processor can increase compute capability. Amortize the cost/complexity of managing an instruction stream across many ALUs using **Single Instruction, Multiple Data (SIMD)** processing. Therefore, the **same instruction is sent to all ALUs**. This **operation is performed in parallel on all ALUs**.

✔ **Advantages**

- **Efficient for data-parallel workloads**: amortize control costs over many ALUs.
- Vectorization done by:
  - Compiler (**explicit SIMD**): parallelism is explicitly requested by the programmer through intrinsics, conveyed through parallel language semantics, and inferred through loop dependency analysis by the "auto-vectorizing" compiler. In other words, the **SIMD parallelization is done at compile time, and when we inspect the program binary, we can see the SIMD instructions**.
  - At runtime by hardware (**implicit SIMD**): the **compiler generates a binary with scalar instructions**, but n instances of the program are always executed together on the processor. The **hardware** (not the compiler) is **responsible for the simultaneous execution of the same instruction by multiple program instances on different data on SIMD ALUs**.

2.1.4 Multi-Core Processor

A **Multi-Core Processor (MCP)** is a **microprocessor** on a single integrated circuit (IC) with **two or more separate central processing units (CPUs)**, called *cores* to emphasize their multiplicity (e.g., *dual-core* or *quad-core*). Each core reads and executes program instructions, specifically ordinary CPU instructions (such as `add`, `move data`, and `branch`). However, the MCP can **execute instructions on separate cores simultaneously**, **increasing overall speed for programs that support multithreading or other parallel computing techniques**.

✔ **Advantages**

- Provides **thread-level parallelism**: execute a completely different instruction stream on each core simultaneously.
- **Software creates threads to expose parallelism to hardware** (e.g., via threading API)

2.2 Accessing Memory

2.2.1 What is a memory?

A computer's memory is organized as an array of bytes. Each byte is identified by its address in memory (its position in that array).

Address	Value
0x0	16
0x1	255
0x2	14
0x3	0
0x4	0
0x5	0
0x6	6
0x7	0
0x8	32
0x9	48
0xA	255
0xB	255
0xC	255
0xD	0
0xE	0
0xF	0
0x10	128
⋮	⋮
0x1F	0

Table 1: Example illustration of the program's memory address space of 32 bytes, range from `0x0` to `0x1F`.

From the processor's point of view, loading an instruction to access the contents present in memory is done with the `ld` assembly instruction. For example, `ld R0 ← mem[R2]` means "take the value from register `R2` and put that value into register `R0`".

Before we introduce new concepts, let us take a moment to explain some important **terminology**:

- **Memory Access Latency**, is the **time** it takes for the **memory system to deliver data to the processor**.
- **Processor Stall**. A **processor** stalls when it **cannot execute the next instruction in an instruction stream because of a dependency on a previous instruction that has not been completed**. Accessing memory is a major source of stalling, which is one of the main reasons why memory accesses should be limited.

For **example**, in the following three assembler instructions, the add has to wait for the loading of `R2` and `R3` values, making parallelization more complicated:

```
ld r0 mem[r2]
ld r1 mem[r3]
add r0, r0, r1
```

- **Memory Bandwidth**, is the **rate at which the memory system can provide data to a processor**.

  Bandwidth is the critical resource in modern computing.

  High-performance parallel programs will:

  1. **Organize computation to fetch data from memory less frequently**. For example, reuse data previously loaded by the same thread (temporal locality optimizations) or share data across threads (inter-thread cooperation);
  2. Prefer to **perform additional arithmetic to store/reload values**;
  3. **Programs need to access memory infrequently** to take advantage of modern processors.

2.2.2 How to reduce processor stalls

2.2.2.1 Cache

One of the most common solutions is caching.

A **cache** is a hardware or software **component that stores data so that future requests for that data can be served faster**; the data stored in a cache might be the result of an earlier computation or a copy of data stored elsewhere.

- A **Cache Hit** occurs when the **requested data is found** in a cache;
- A **Cache Miss** occurs when **it cannot**.

Cache hits are served by reading data from the cache, which is faster than recomputing a result or reading from a slower data store, so the **more requests that can be served from the cache, the faster the system performs**. [12]

Many modern CPUs have logic that predicts what data will be accessed in the future and "pre-fetches" that data into caches. **Prefetching** reduces stalls because the data is resident in the cache when it is accessed. But beware, the other side of the coin is that if the **guess is wrong**, the **performance is worse than the system without prefetching**!

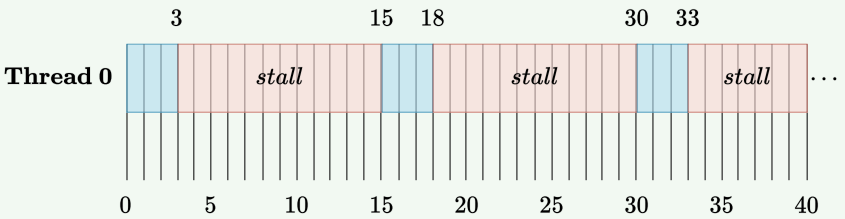
2.2.2.2 Multi-threading

A **Multithreaded Processor** is one that has the **ability to follow multiple instruction streams without software intervention**. In practice, then, this **includes any machine that stores multiple program counters (PCs) in hardware within the processor** (i.e., on chip, for microprocessor-era machines). [10]

The main idea in this architecture is to **interleave processing of multiple threads on the same core to hide stalls**. In other words, if we can't make progress on the current thread, we work on another one.

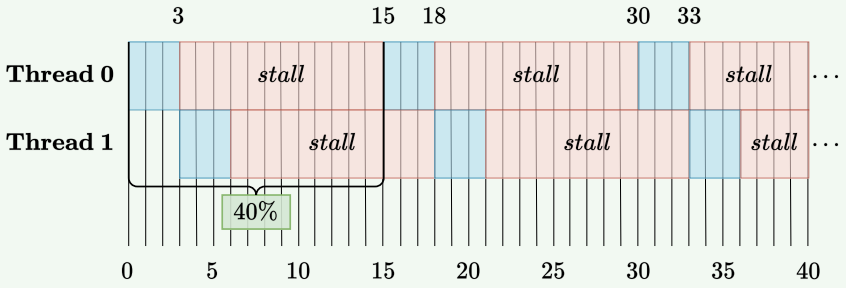
Example 1: core utilization

To better understand our explanation, suppose we are running a program where threads perform **three arithmetic instructions followed by a memory load** (with 12 cycle latency).

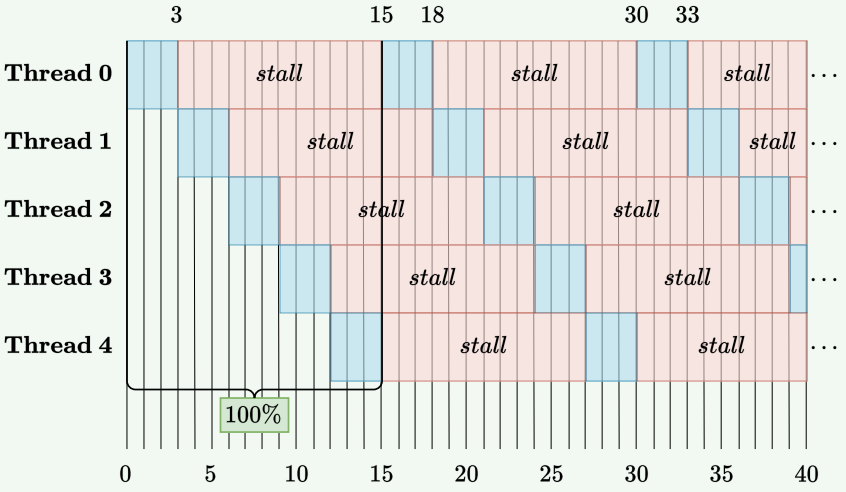


From the figure, it is clear that if we consider an arithmetic instruction and a memory stall, the core is not fully optimized to work at 100%. In

practice, we see that a single arithmetic instruction takes 3 clock cycles and the memory stall takes 12 clock cycles. This means that from this situation we are using the CPU at only 20% (3 work clock cycles on 15)! Without suggesting the final solution, try to see what happens when we add another thread.



We have gained three clock cycles, and now we are taking advantage of the 40% of the core.
Now, how many threads do we need to achieve 100% utilization? The answer is simple: the number of clock cycles of the operations to be done before the stall plus the clock cycles of the stall divided by the working operations (operations that are not stalls). In our case: $15 \div 3 = 5$.



Note that if we add more threads, there will be no benefit because the CPU is already at 100%.

✔ Multithreaded Processor benefits

- A processor with multiple hardware threads has the **ability to avoid stalls** by executing instructions from other threads when one thread must wait for a long latency operation to complete. The latency of the memory operation is not changed by multithreading, it just no longer causes reduced processor utilization.
- A **multithreaded processor hides memory latency by performing**

arithmetic from other threads. Program that feature more arithmetic per memory access need fewer threads to hide memory stalls.

Type of hardware-supported multithreading

- **Core manages execution contexts for multiple threads**. This type still has the same number of ALU resources: multi-threading only helps to use them more efficiently in the face of high latency operations such as memory access. The **processor decides which thread to run each clock cycle**.
- **Coarse-Grain Multithreading**, also called **Block Multithreading** or **Switch-On-Event Multithreading**, has multiple hardware contexts associated with each processor core. A hardware context is the program counter, register file, and other data required to enabled a software thread to execute on a core. However, only one hardware context has access to the pipeline at a time. [10]
- **Fine-Grain Multithreading (FGMT)**, also known as **Interleaved Multithreading** or **Temporal Multithreading**, is the type just described on the previous pages, as in the example on page 27.
- **Simultaneous Multithreading (SMT)** has multiple hardware contexts associated with each core. In a simultaneous multithreaded processor, instructions from multiple threads are available to be issued on any cycle. Therefore, all hardware contexts, and in particular all register files, must be accessible to the pipeline and its execution resources. [10]

  In other words, **each clock**, the **core selects instructions from multiple threads to execute on ALUs**.

3 Programming models

3.1 Implicit SPMD Program Compiler (ISPC)

Before introducing the ISPC compiler, we give the definition of SPMD.

Definition 1: Single Program, Multiple Data (SPMD)

Single Program, Multiple Data (SPMD) is a term that has been used to refer to computational models for exploiting parallelism, where **multiple processors work together to execute a program to achieve faster results**.
The **difference** between *SPMD* and *SIMD* (page 24) is that in SPMD parallel execution, **multiple autonomous processors simultaneously execute the same program at independent points**, rather than in SIMD it is vectorization at the instruction level so that **each CPU instruction processes multiple data elements**.
In other words:

- SPMD: is the **programming abstraction**, because the programmer has to think; the program is written in terms of this abstraction.
- SIMD: in general, the compilers (ISPC) issue special vector instructions that execute the logic performed by each parallel instance created (ISPC gang spawned). In addition, the compilers handle the mapping of conditional control flow to vector instructions.

The difference and the terminology used by ISPC will become clearer in the following pages. We suggest that finish this section and come back here in a moment.

Definition 2: Implicit SPMD Program Compiler (ISPC)

Implicit SPMD Program Compiler (ISPC) is a **compiler for a variant of the C programming language**, with extensions for *Single Program, Multiple Data (SPMD)* programming. Under the SPMD model, the programmer writes a program that generally appears to be a regular serial program, though the execution model is actually that a number of program instances execute in parallel on the hardware. In other words, the **ISPC gives the programmer some API to do parallelization on the code; it also generates high quality SIMD code to increase performance**.

The definition, implementation, and other details are explained in the official Intel GitHub repository.

How it works?

Let us take a general main program; when we call an `ispc` function, it causes a **spawn of gang of ISPC program instances upon return, all instances have completed**. These **instances execute the same ISPC code simultaneously**, and **each instance has its own copy of local variables**. Take the following ISPC code as an example:

```
1  export void ispc_sinx(
2      uniform int N,
3      uniform int terms,
4      uniform float* x,
5      uniform float* result
6  ){
7      // assume N % programCount = 0
8      for (uniform int i=0; i<N; i+=programCount) {
9          int idx = i + programIndex;
10         float value = x[idx];
11         float numer = x[idx] * x[idx] * x[idx];
12         uniform int denom = 6; // 3!
13         uniform int sign = -1;
14         for (uniform int j=1; j<=terms; j++) {
15             value += sign * numer / denom
16             numer *= x[idx] * x[idx];
17             denom *= (2*j+2) * (2*j+3);
18             sign *= -1;
19         }
20         result[idx] = value;
21     }
22 }
```

In the example, the `programCount` (row 8) and `programIndex` (row 9) variables, `uniform` (row 2, and so on) data type tell us:

- `programIndex` gives the index of the SIMD-lane being used for running each program instance (in other words, it's a varying **integer value that has value zero for the first program instance, and so forth**).
- `programCount` gives the **total number of instances in the *gang***.
- A variable that is declared with the `uniform` qualifier represents a **single value that is shared across the entire *gang***.

Together, these can be used to uniquely map executing program instances to input data (programIndex and programCount, uniform data type).

With the ISPC analogy, the **SPMD programming model** should be clear:

1. **Single thread of control** (typically a main program);
2. **Invoke** the **SPMD function** (in the previous example, the `ispc_sinx` function);
3. **SPMD execution**, then **multiple instances of the function run in parallel** (multiple logical threads of control);
4. **Returns** and **resumes a single thread** of control.

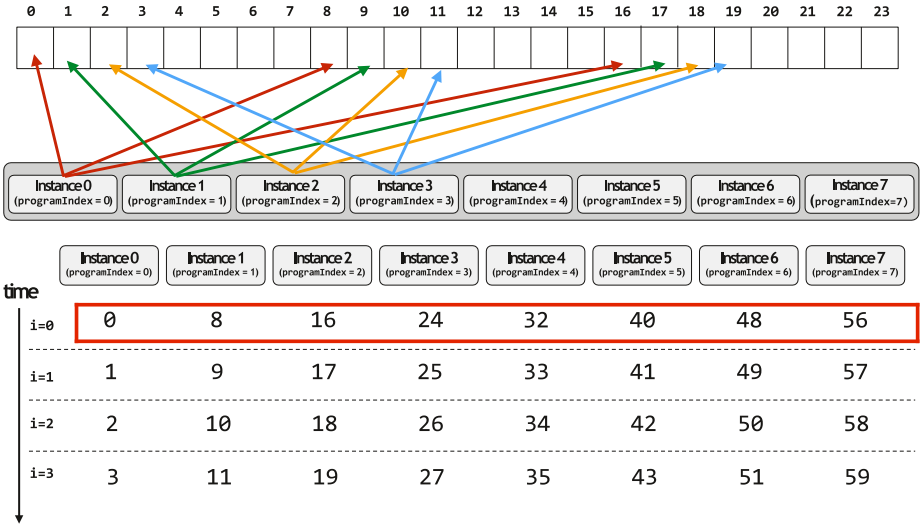


Figure 9: Example of execution with 8 instances (`programCount` equal to 8). For all program instances, there are eight non-contiguous values in memory. A special instruction called `gather` is needed to implement this, but unfortunately it is a more complex and expensive SIMD instruction rather than a contiguous implementation.

Figure 9 shows a possible execution of the ISPC function using 8 instances. The result is obtained and all is well. But there is one interesting observation. **Each ISPC instance writes each value in a non-contiguous way**. This can be done better:

```
export void ispc_sinx_v2(
    uniform int N,
    uniform int terms,
    uniform float* x,
    uniform float* result
){
    // assume N % programCount = 0
    uniform int count = N / programCount;
    int start = programIndex * count;
    for (uniform int i=0; i<count; i++) {
        int idx = start + i;
        float value = x[idx];
        float numer = x[idx] * x[idx] * x[idx];
        uniform int denom = 6; // 3!
        uniform int sign = -1;
        for (uniform int j=1; j<=terms; j++) {
            value += sign * numer / denom
            numer *= x[idx] * x[idx];
            denom *= (j+3) * (j+4);
            sign *= -1;
        }
        result[idx] = value;
    }
}
```

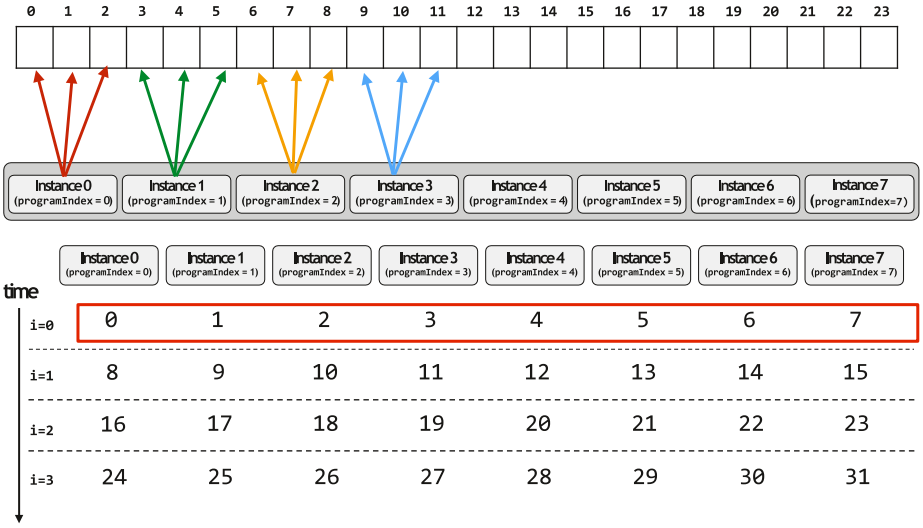


Figure 10: Example of execution with 8 instances (`programCount` equal to 8). A single "packed vector load" instruction efficiently implements this. For all program instances, since the eight values are contiguous in memory.

3.2 Shared Address Space Model

We give a general introduction to memory in the chapter 2.2. In parallel computing theory, **each thread communicates with other threads using read-/write operations**. These instructions operate on a special area called the **Shared Address Space** (also called **Shared Variables**).

Definition 3: Shared Address Space

The **Shared Address Space** view of a parallel platform supports a common data space that is accessible to all processors. Processors interact by modifying data objects stored in this shared-address-space. [6]

Now the first and trivial question should be: a powerful tool is the possibility to allow communication between threads, but *how can we guarantee that two or more threads accessing the same resource do not create well known problems, such as* race condition*?*

This property, commonly called **mutual exclusion** or **atomic operation**, can be guaranteed with some techniques:

- **Lock/Unlock mutex around a critical section**:

```
Lock lock_variable;

// some operations, such as spawn of threads

lock_variable.lock();
// critical section
lock_variable.unlock();
```

- Some languages have first-class **support for atomicity of code blocks**:

```
atomic {
    // critical section
}
```

- Intrinsics for **hardware-supported atomic read-modify-write operations**:

```
atomicAdd(x, 10);
```

The shared address space **requires hardware support to be efficiently implemented**. The main idea is that **each processor can directly reference the contents of any memory location**. Some interesting examples that can be explored in depth are: SUN Niagara 2, Knights landing (KNL): 2nd Generation Intel Xeon Phi processor.

3.3 Message Passing model of communication

In parallel computing, the **message passing model** **allows** threads (or processes) to **communicate by sending and receiving messages**. This model is used to **facilitate data exchange between threads running in their own private address spaces**.

Each thread operates within its private address space, meaning they **do not share memory directly**. When they need to communicate with a specific thread without using a *shared address space model*, they use the *message passing model* to send (or receive) the data.

- Usually, the **sender** specifies
  - the recipient;
  - the buffer to be transmitted;
  - an optional message identifier (a sort of tag).
- Meanwhile, the **receiver** specifies:
  - who the sender is;
  - the buffer where to store data;
  - an optional message identifier (again, a sort of tag).

The message passing model is the **only way to exchange data between threads** because it guarantees three main advantages:

- **Data Isolation**: Using message passing, **each thread maintains its address space**, reducing the risk of race conditions.
- **Scalability**: Message passing **scales well with distributed systems and multi-core architectures**, making it suitable for large-scale parallel computing.
- **Flexibility**: It **allows for explicit control over data exchange and synchronization, providing Flexibility in parallel program design**. Explicit control refers to the fact that in message passing, each communication action is explicitly defined by send and receive operations. This means we can precisely dictate which data is sent, when, and to whom it is sent. Furthermore, the synchronization is managed very well because message passing naturally incorporates synchronization. When a process sends a message, it can block until it is received, ensuring that the sender and receiver are synchronized.

Why message passing is preferable to the shared address space model

Unlike shared memory systems that require complex hardware mechanisms to implement system-wide load and store operations, message passing systems do not need this capability. They **only need to communicate messages between nodes**. It can be considered as a great advantage for message passing models, which for this reason is very much used in the supercomputers and clusters. Finally, this model has the ability to **connect commodity hardware systems together to form a large parallel machine**. This means we can use off-the-shelf components to build powerful computing clusters.

3.4 Data-Parallel model

In parallel computing, the **Data-Parallel model** is characterized by **applying the same operation simultaneously across multiple data points**. This model is particularly effective for tasks involving large datasets where the same computation needs to be performed on each data element.

This model organizes **computation as operations on sequences of elements** (e.g., perform the same function on all elements of a sequence). In programming languages, the basic data type used for this purpose is called `Sequence` (e.g., C++). Despite the name of the datatype used in the languages, the definition is that it **is an ordered collection of elements, where each element can be accessed and manipulated using various sequence operators**. The most common operators are:

- **Map**. The `map` function is a **higher-order function**[2] that **operates on sequences**. It applies a **side-effect-free unary function**[3] $f : a \to b$ **to all elements of an input sequence**, producing an **output sequence of the same length**.

  Example 1: Python Analogy

  For a better understanding, we provide a very simple Python code to see how a `map` function works. In Python there is a `map` function that does exactly what we say.

```python
# Define a trivial function that squares a number
def square(x):
    return x * x

# Create a sample list
numbers = [1, 2, 3, 4, 5]

# Use the map function to apply the 'square' function
# to each element in the 'numbers' list
squared_numbers = map(square, numbers)

# Convert the result to a list and print it
print(list(squared_numbers))
```

  And the result is:

```
[1, 4, 9, 16, 25]
```

  Now the main idea is: since the `map` is a function without side effects, we can **apply it to all elements of the sequence in any order without changing the output of the program**. This allows **reordering or parallel processing of sequence elements to optimize performance**.

[2] A **higher-order function** is a function that can do one or both of the following:
- Take other functions as arguments (parameters).
- Returns a function or value as its result.

[3] A function that takes only one argument and doesn't suffer from side effects.

- **Reduction**. The reduction born from the need to make parallel operations of iterations. For example, in a `for` loop, if we need a progressive sum of an element, we should implement some kind of mechanism to manage the synchronization. Using the reduction strategies, the compiler will do this job. We suggest reading the Reduction section (5.3.1.1, page 64) in the OpenMP chapter to understand what we mean.
- Others like **scan** and **shift**.

4 Parallel Programming Models and pthreads

4.1 How to create parallel algorithms and programs

Although *parallel algorithms* and *parallel programs* are in the same father set, the parallel computing topic, these two arguments are a little different.

Definition 1: Parallel Algorithms

A **parallel algorithm**, as opposed to a traditional serial algorithm, is an **algorithm which can do multiple operations in a given time**.

Definition 2: Parallel Programs

A **parallel program** is a **program that uses multiple CPU cores**, with **each core performing a task independently**.

However, designing *parallel algorithms* is not an easy task because there is no heuristic for designing *parallel algorithms*. There are some rules that help in the design. The same reasoning applies to *parallel programs*, because they depend on the chosen language and architecture.

Furthermore, there is no single correct solution, but several possible parallel solutions. A **good first approach is to start with machine-independent issues** (concurrency) and **delay target-specific issues as much as possible**.

Design a parallel algorithm	Design a parallel program
Understand the problem to be solved	Analyze the target architecture(s)
Analyze data dependencies	Choose the best parallel programming model and language
Partition the solution	Analyze the communications (cost, latency, bandwidth, visibility, synchronization, etc.)

Table 2: Design parallel algorithms and parallel programs.

The **PCAM (Partitioning, Communication, Agglomeration, Mapping)** methodology described by Argonne National Laboratory is intended to promote an exploratory **approach to design in which machine independent issues**, such as *concurrency*, are **considered early** and **machine specific aspects of design are deferred until late in the design process**. In other words, we immediately consider the machine-independent issues (e.g., concurrency) at the beginning of the design approach, and all machine-specific aspects are postponed to an advanced stage of the design process.

This methodology structures the design process into **four distinct stages**:

1. **Partitioning**. The **computation** that is to be performed and the **data** operated on by this computation are **decomposed into small tasks**. Practical issues such as the number of processors in the target computer are ignored, and attention is focused on recognizing opportunities for parallel execution.

2. **Communication**. The communication required to **coordinate task execution** is determined, and appropriate **communication structures and algorithms are defined**.

3. **Agglomeration**. The task and **communication structures** defined in the first two stages of a design are **evaluated with respect to performance requirements and implementation costs**. If necessary, tasks are combined into larger tasks to improve performance or to reduce development costs.

4. **Mapping**. **Each task is assigned to a processor** in a manner that attempts to satisfy the competing goals of **maximizing processor utilization** and **minimizing communication costs**. Mapping can be specified statically or determined at runtime by load-balancing algorithms.

In the **first two stages**, we focus on **concurrency** and **scalability** and seek to **discover algorithms with these qualities**. In the **third and fourth stages**, attention shifts to **locality** and **other performance-related issues**.

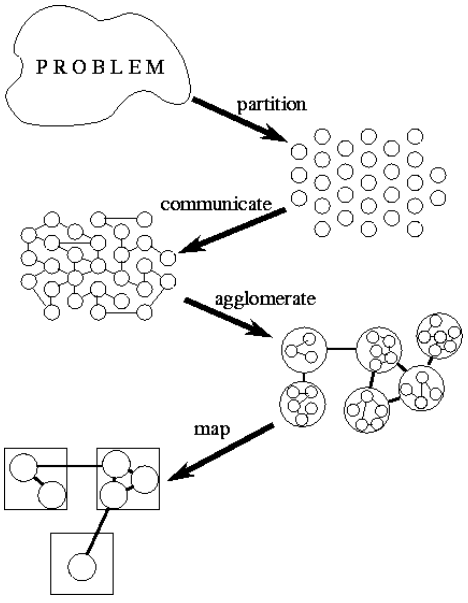


Figure 11: PCAM design methodology for parallel programs. Starting with a problem specification, develop a partition, determine communication requirements, agglomerate tasks, and finally map tasks to processors.

4.2 Analyze parallel algorithms

Whether we want to analyze our parallel algorithm created with the PCAM model or evaluate a general parallel algorithm, we need some metrics.

The classical **metrics** needed to evaluate a parallel algorithm are:

- **Time complexity**: quantifies the amount of **time required to produce a solution**.
- **Resource complexity**: quantifies how many **resources are needed to produce the solution in that time**.

In general, to analyze a **parallel algorithm**, we can consider its **structure as a directed acyclic graph (DAG)**[4], where the nodes are the task and the edges are the data dependencies.

Parallel Algorithm Terminology and Metrics

- **Concurrent tasks**, each task is executed ***independently***.
- **Parallel tasks**, each task is executed at the ***same time*** (because multiple computing resources are available).
- **Work** W is the ***number of operations executed***. It may be higher than the sequential version of the algorithm due to communication overhead, etc.
- **Span** S is the ***longest chain of dependencies*** (i.e., the critical path) that determines the ***minimum time required to execute the algorithm***. This is a *lower bound* on the running time, regardless of the number of processors. The range indicates the ability of an algorithm to get better performance on more processors.

  #### How do we calculate the Span metric?

  1. As we just said, we **represent a parallel algorithm as a DAG** graph, where nodes represent tasks and edges represent dependencies between tasks;
  2. We **assign weights to each node** that represent the **time required to perform the corresponding task**;
  3. We try to **find the Critical Path**. In other words, we determine the path from the start node to the end node that has the *maximum cumulative weight*;
  4. Finally, the **sum** of the **weights of the nodes on the critical path** gives us the span value!

[4] A **Directed Acyclic Graph (DAG)** is a directed graph, i.e. with oriented edges, without cycles.

- **Parallelism** P is the ***measure of efficiency in the use of resources***. Trivially, it is the number of operations performed divided by the longest chain of dependencies:

$$P = \frac{W}{S} \tag{11}$$

It indicates **how many processors can be effectively used by the computation**. If the work is equal to the span, the parallelism is 1 and the computation is sequential. Ideally, but not necessarily, we win with polylogarithmic span, because if the work is $O(n \log n)$ and the span is $O(\log^2 n)$, then the parallelism is $O\left(\frac{n}{\log n}\right)$, which is actually quite high (and unlikely to be a bottleneck on most machines in the next 25 years). [2]

This measure is *one of the most important*. It indicates the **number of processors that are not idle**. It is obvious that a **good parallel algorithm is designed to have the lowest possible work** (less operation, then less resource usage, then less cost, and so on) **and the highest possible parallelism** (achievable by reducing the span, and this should be trivial, since the metric P is given by work divided by the span, so reducing the denominator, you can get a higher value).

As in all things, there is a **trade-off** between the lowest possible "work" and the highest possible "parallelism". Reducing the work too much could eliminate the possibility of parallelizing our algorithm, and on the other hand, reducing the span too aggressively could cause communication/synchronization overhead.

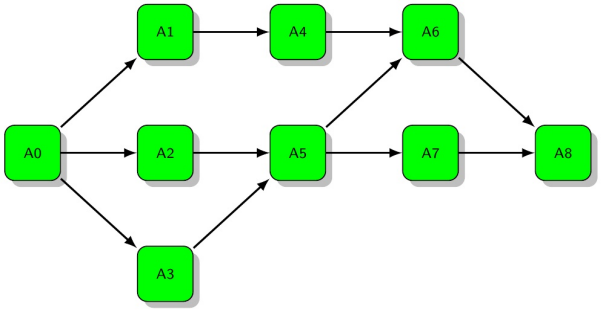


Figure 12: Example of DAG implementation with work equal to 9, span equal to 5, and parallelism equal to 1.8 ($9 \div 5$). The span calculus is not well known, has been calculated *a priori*.

Finally, we use a mathematical annotation and not only a graphical (DAG) annotation. The **Composition Rules** help determine how to combine smaller parallel tasks into a larger algorithm, while analyzing the Work and Span of the combined algorithm:

- **Single operation**. An operation takes 1 unit of work and 1 unit of span time.

$$W(op) = 1 \qquad S(op) = 1$$

- **Sequential Composition**.
  - The total work of executing e_1 and e_2 **sequentially** is the sum of their individual work.

$$W(e_1, e_2) = W(e_1) + W(e_2)$$

  - The total span of executing e_1 and e_2 **sequentially** is the sum of their individual spans.

$$S(e_1, e_2) = S(e_1) + S(e_2)$$

- **Parallel Composition**.
  - The total work of executing e_1 and e_2 in **parallel** is still the sum of their individual works.

$$W(e_1||e_2) = W(e_1) + W(e_2)$$

  - The total span of executing e_1 and e_2 in **parallel** is the ***maximum of their individual spans***, since they can be executed simultaneously.

$$S(e_1||e_2) = \max(W(e_1), W(e_2))$$

4.3 Technologies

Some famous architecture to work with parallel programming:

- **Verilog/VHDL** are *hardware description languages*. The target architectures are *ASIC and FPGA*. The **parallelism** and the **communication** are **explicit**.

  ✔ Pros

  - Complete control on computation and memory
  - No overhead introduced in the computation
  - Provides access to potentially large computational power

  ⚠ Cons

  - Requires specific hardware (e.g., ASIC or FPGA) to implement functionality
  - Difficult to learn: completely different programming language and programming paradigm
  - Depends on the chosen target architecture

- **MPI** is a *library*. The target architectures are *Multi CPUs*. The **parallelism** is **implicit** and the **communication** is **explicit**.

  ✔ Pros

  - Can be adopted on different types of architecture
  - Scalable solutions
  - Synchronization and data communication are explicitly managed

  ⚠ Cons

  - Communication can introduce significant overhead
  - Programming paradigm more difficult than shared memory-based ones
  - Standard does not reflect immediately advances in architecture characteristics

- **PThread** is a *library*. The target architectures are *Multi-core CPUs*. The **parallelism** is **explicit** and the **communication** is **implicit**.

  ✔ Pros

  - Can be adopted on different types of architecture
  - Explicit parallelism and full control over application

  ⚠ Cons

  - Task management overhead can be significant
  - Not easily scalable solutions
  - Low level API

- **OpenMP** is a *C/Fortran extensions*. The target architectures are *Multi-core CPUs*. The **parallelism** is **explicit** and the **communication** is **implicit**.

  ✔ Pros

  - Easy to learn
  - Scalable solution
  - Parallel applications can also be executed sequentially

  ⚠ Cons

  - Mainly focused on shared memory homogeneous systems
  - Requires small interaction between tasks

- **CUDA** is a *C extensions*. The target architectures are *CPU plus GPU(s)*. The **parallelism** is **implicit/explicit** and the **communication** is **implicit/explicit**.

  ✔ Pros

  - Provides access to the computational power of GPUs
  - Writing a CUDA kernel is quite easy
  - Already optimized libraries

  ⚠ Cons

  - Targets only NVIDIA GPUs
  - Difficult to extract massive parallelism from application
  - Difficult to optimize CUDA kernel

- **OpenCL** is a *C/C++ extensions and API*. The target architectures are *heterogeneous architecture*. The **parallelism** is **implicit/explicit** and the **communication** is **implicit/explicit**.

  ✔ Pros

  - Target-independent standard
  - Hides architecture details
  - Same programming infrastructure for every heterogeneous architecture: CPU + GPU (and FPGA)

  ⚠ Cons

  - Difficult programming paradigm for its heterogeneity
  - Hiding of architecture details makes difficult to obtain best performances
  - Gradually abandoned

- **Apache Spark** is an *API*. The target architectures are *multi CPUs*. The **parallelism** is **implicit** and the **communication** is **implicit**.

  ✔ Pros

  - API for different languages
  - Explicit parallelization and communication are not required
  - Preinstalled on cloud provide VMs

  ⚠ Cons

  - Suitable only for big data applications
  - Does not (yet) fully support GPUs

Regardless of these technologies, it is quite common to mix some of them:

- **OpenMP + CUDA**: allows to exploit multi-core CPU and GPU. CUDA is used to parallelize GPU code and OpenMP is used to parallelize CPU code.
- **MPI + OpenMP**: the most common scenario are:
  1. MPI used to express coarser parallelism (multi CPU) and OpenMP used to express finer parallelism (multi core).
  2. MPI used to implement communication and OpenMP used to parallelize computation.
- **OpenCL + Verilog or VHDL**: in principle, hardware kernels (implemented for example on FPGA) can be used as accelerators; OpenMP used to describe parallelism among different processing elements; Verilog/VHDL used to describe hardware kernel. An example of target: Intel Xeon Scalable.

4.4 Threads

4.4.1 Flynn's taxonomy

Flynn's taxonomy is a **classification of computer architectures**, proposed by Michael J. Flynn. The classification system has been used as a tool in the design of modern processors and their functionalities. Since the rise of multiprocessing central processing units (CPUs), a multiprogramming context has evolved as an extension of the classification system.

The four initial classifications defined by Flynn are based upon the number of concurrent instruction (or control) streams and data streams available in the architecture:

- **Single Instruction stream, Single Data stream (SISD)**
- **Single Instruction stream, Multiple Data streams (SIMD)**
- **Multiple Instruction stream, Single Data stream (MISD)**
- **Multiple Instruction stream, Multiple Data stream (MISD)**

It is important to quote it because it is the basis for the development of many advanced technologies.

4.4.2 Definition

A UNIX process can be created by the operating system and contains information about program resources and program execution status.

Definition 3: Thread

A **thread** is an **independent stream of instructions within a process**. Threads can be scheduled by the operating system, and each thread can run concurrently with other threads. A thread also has local resources and can access the shared process resources.

In other words, a thread can be thought of as any **procedure that runs independently of its main program**. We can create each thread dynamically during execution. A good point is that a multi-threaded program is lighter than a multi-process program.

When a thread exists within a process, it shares most of the process resources, for example:

- Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads.
- Two pointers having the same value point to the same data.
- **Implicit communication** by reading and writing shared variables.
- **Reading and writing to the same memory locations requires explicit synchronization by the programmer**. If this rule is not followed, the code may suffer from a data race or race condition [5] problem.

The most common models for threaded programs are the manager / worker model[6] and pipeline.

This chapter introduces the POSIX threads model.

Definition 4: pthreads

POSIX Threads, commonly known as **pthreads**, is an **execution model** that exists independently from a programming language, as well as a parallel execution model. It **allows a program to control multiple different flows of work that overlap in time**. Each flow of work is referred to as a thread, and creation and control over these flows is achieved by making calls to the POSIX Threads API.

POSIX threads and OpenMP are two **implementations of a shared memory parallel programming model using threads**. The **programmer is responsible for handling parallelism and synchronization**, usually through a library of subroutines or a set of compiler directives. Typically, hardware vendors have implemented their own proprietary versions of threads, but in this course we will look at POSIX threads (pthreads) and OpenMP.

[5] In parallel computing, a **Data Race** or **Race Condition** is a software problem that occurs when two threads (or processes) access the same variables, and at least one does a write. They can finish in a different order than expected.

[6] The manager/worker pattern is described as follows. The idea is that the work that needs to be done can be divided by a "manager" into separate pieces and the pieces can be assigned to individual "worker" processes. Thus the manager executes a different algorithm from that of the workers, but all of the workers execute the same algorithm. Most implementations of MPI allow MPI processes to be running different programs (executable files), but it is often convenient (and in some cases required) to combine the manager and worker code into a single program.

4.4.3 pthreads API

In 1995, the IEEE POSIX 1003.1c standard specified the API for explicitly managing threads. An **API is a set of C language programming types and procedure calls**.

- **Header file** to include in the main file: `pthread.h`.
- To **compile** and use it, it is necessary to add the **flag** `-pthread` to the gcc (or g++) options.

The API are divided by what we want to do. In general, there are two sets: thread management and thread synchronization.

- Thread Management
  - Creation (page 48)
  - Termination (page 49)
  - Joining (page 50)
  - Detaching (page 51)
  - Joining through Barriers (page 52)
- Thread Synchronization
  - Mutexes (page 53)
  - Condition variables (page 53)

4.4.3.1 Creation

Once threads are created, they are peers, and **may create other threads**. There is **no implied hierarchy or dependency between threads**. The **maximum number of threads depends on the implementation**. Doc.

pthread API: `pthread_create`

```
int pthread_create(
    pthread_t * thread,
    const pthread_attr_t * attr,
    void * (* start_routine) (void *),
    void * arg
)
```

- **Return value**: on success, `pthread_create()` returns 0; on error, it returns an error number, and the contents of `*thread` are undefined.
- **Arguments**:
  - `thread`: identifier for the new thread returned by the subroutine.
  - `attr`: used to set thread attributes, such as joinable, detached, scheduling and stack size.

- `start_routine`: the C routine that the thread will execute once it is created.
- `arg`: argument passed to `start_routine`. It must be passed by address as a pointer cast of type void.

4.4.3.2 Termination

The thread returns from its startup routine when its "life" ends. The thread makes a call to the `pthread_exit` subroutine. Doc.

pthread API: `pthread_exit`

```
1 void pthread_exit(void *retval)
```

- **Return value**: this function does not return to the caller.
- **Arguments**:
  - `retval`: function terminates the calling thread and returns a value via `retval`.

The thread is canceled by another thread via the `pthread_cancel` routine. Doc.

pthread API: `pthread_cancel`

```
1 int pthread_cancel(pthread_t thread)
```

- **Return value**: on success, `pthread_cancel()` returns 0; on error, it returns a nonzero error number.
- **Arguments**:
  - `thread`: the `pthread_cancel()` function sends a cancellation request to the thread `thread`.

4.4.3.3 Joining

The join function **blocks the calling thread until the specified thread exits**. Doc.

pthread API: `pthread_join`

```
1 int pthread_join(pthread_t thread, void **retval)
```

- **Return value**: on success, `pthread_join()` returns 0; on error, it returns an error number.
- **Arguments**:
  - `thread`: the `pthread_join()` function waits for the thread specified by `thread` to terminate. If that thread has already terminated, then `pthread_join()` returns immediately. The thread specified by `thread` must be joinable.

    If multiple threads simultaneously try to join with the same thread, the results are undefined. If the thread calling `pthread_join()` is canceled, then the target thread will remain joinable (i.e., it will not be detached).
  - `retval`: if `retval` is not `NULL`, then `pthread_join()` copies the exit status of the target thread (i.e., the value that the target thread supplied to `pthread_exit()`) into the location pointed to by `retval`. If the target thread was canceled, then `PTHREAD_CANCELED` is placed in the location pointed to by `retval`.

4.4.3.4 Detaching

The detach function **marks a thread as detached**. When a thread is detached, its **resources are automatically released back to the system when the thread terminates**, without the need for another thread to join with it.

Why would I need to detach a thread and not join it?

Good question. The answer depends on what we have to do.

- **Fire and forget tasks**. When we start a thread to perform a task that doesn't require further interaction or result processing, releasing it ensures that the resources are automatically cleaned up when the task is complete.
- **Resource management**. Detaching avoids the need for another thread to call `pthread_join()`, which can save system resources and reduce the complexity of our code. It's especially useful in a highly concurrent application with many short-lived threads.
- **Avoid deadlocks**. When we have potential circular dependencies or complex synchronization between threads, detaching threads can help avoid deadlocks by eliminating the need for one thread to wait on another.
- **Long-running background tasks**. For tasks that should run independently in the background and not block the main program flow, detaching makes sense. We make sure they clean up after themselves without having to explicitly manage their lifecycle.

Doc.

pthread API: `pthread_detach`

```
1 int pthread_detach(pthread_t thread)
```

- **Return value**: on success, `pthread_detach()` returns 0; on error, it returns an error number.
- **Arguments**:
  - `thread`: the `pthread_detach()` function marks the thread identified by `thread` as detached. When a detached thread terminates, its resources are automatically released back to the system without the need for another thread to join with the terminated thread.

    Attempting to detach an already detached thread results in unspecified behavior.

4.4.3.5 Joining through Barriers

The *barrier init* function **initializes a barrier object**, and the *barrier wait* function **blocks a thread until the specified number of threads have called it**. A **barrier object** is, in parallel computing, a synchronization tool that ensures that multiple threads reach a certain point of execution before any of them continue. It's like a **checkpoint that everyone must reach before continuing**, ensuring coordinated progress in a parallel algorithm.[7] Doc.

pthread API: `pthread_barrier_init`

```
int pthread_barrier_init(
    pthread_barrier_t * barrier,
    pthread_barrierattr_t * attr,
    unsigned int count
)
```

pthread API: `pthread_barrier_wait`

```
int pthread_barrier_wait(pthread_barrier_t * barrier)
```

- **Return value**: on success, function return 0; on error, they return an error number.
- **Arguments**: the main and most important argument is `count`, which specifies the **number of threads to wait** for.

[7] For example, imagine multiple threads working on different parts of a matrix. A barrier can ensure that all threads finish their part of the computation before moving on to the next phase, such as combining results or performing subsequent operations.

4.4.3.6 Mutexes

Mutex variables are the basic **method of protecting shared data when multiple writes occur**. Only **one thread can lock a mutex variable at a time**. If multiple threads attempt to lock a mutex, only one thread will succeed. **Threads that fail to acquire the mutex are blocked**. There is also the `trylock` function, which returns immediately if the mutex is currently locked (by any thread, including the current thread). Note that a **lock function has the potential to create a deadlock situation**.

There are also **three types of mutex** that can be set using the `settype` function:

- **Normal Mutex** (`PTHREAD_MUTEX_NORMAL`). A **normal mutex does not check for errors such as deadlock**. If a thread tries to lock a mutex it already owns, the thread will deadlock.
- **Error Check Mutex** (`PTHREAD_MUTEX_ERRORCHECK`). Provides **error checking**. If a thread tries to lock a mutex it already owns, `lock` function will return an error instead of deadlocking.
- **Recursive Mutex** (`PTHREAD_MUTEX_RECURSIVE`). Allows the same thread to lock the mutex multiple times without deadlocking. Each lock must have a corresponding unlock.

Doc.

pthread API: `pthread_mutex_lock`

```
1 int pthread_mutex_lock(pthread_mutex_t *mutex)
```

pthread API: `pthread_mutex_trylock`

```
1 int pthread_mutex_trylock(pthread_mutex_t *mutex)
```

pthread API: `pthread_mutex_unlock`

```
1 int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

4.4.3.7 Condition variables

Mutexes implement synchronization by serializing data accesses. Condition variables allow threads to synchronize explicitly by signaling the meeting of a condition. Without condition variables, the programmer would need to poll to check if the condition is met.

5 OpenMP v5.2

5.1 Introduction

OpenMP is a scalable model that gives parallel programmers a simple and flexible interface for developing portable parallel applications in C/C++ and Fortran.

- **Header file** to include in the main file: `omp.h`.
- To `compile` and use it, it is necessary to add the `flag -fopenmp` to the gcc (or c++) options.

The following link contains about 200 slides on OpenMP. It is an introduction, but it covers every argument required in this PoliMI course. The slides are made by a Senior Principal Engineer, Mattson Tim. He's a Senior Principal Engineer at Intel, where he's been since 1993. The slide link.

✔ Benefits

- Standard across a variety of shared memory architectures and platforms.
- Supports three famous languages: Fortran, C and C++.
- Scalable from embedded systems to the supercomputer.
- The directives are intuitive, and with a limited set of directives we can implement parallel algorithms.
- Incremental parallelization of serial program.
- Coarse-grained and fine-grained parallelism. See here here an interesting difference between coarse-grained and fine-grained architecture:

? How it works?

OpenMP is based on the **fork-join paradigm**. A "master" thread forks a specified number of "slave" threads. Tasks are divided among the "slaves", and each "slave" runs concurrently as the runtime allocates threads to different processors.

1. `Thread #0` **born**. OpenMP programs start with a single thread;
2. **Fork**. At the start of a parallel region, the *master* creates a team of parallel worker threads (*slaves*).
3. **OpenMP code block**. Statements in the parallel block are executed in parallel by each thread.
4. **Join**. At the end of the parallel region, all threads synchronize (implicit barrier, see definition on page 52) and join the master thread. [5]

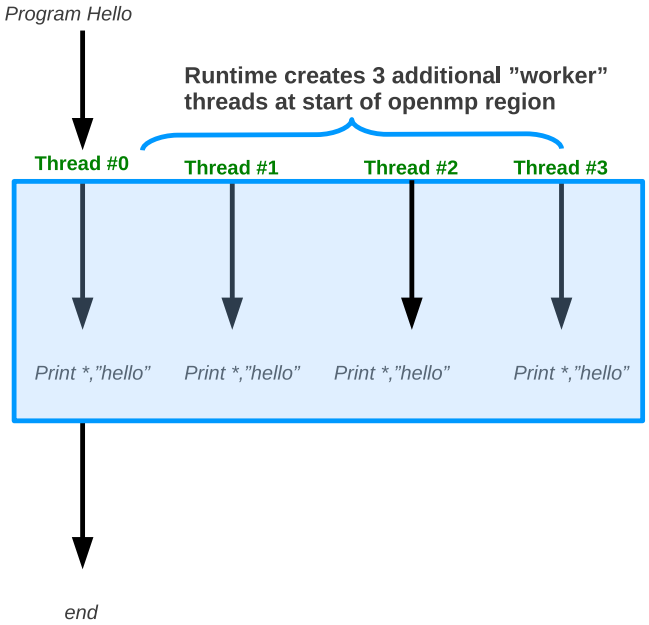


Figure 13: Example of an OpenMP program called `Hello`. At runtime, the *master* thread forks 3 additional *slave* threads to print `hello`. The example is very trivial, but here is a graphical representation of the workflow. An interesting thing to note is that **OpenMP creates** a sort of **region where each thread executes all the instructions in the OpenMP block**. This is important to understand. [5]

5.2 Basic syntax

We can manage OpenMP work flow using the directive syntax. We remember that the reference guide of OpenMP is available on their website:

Reference Guide 

A **directive** is a combination of the base-language mechanism and a *directive-specification* (the *directive-name* followed by *optional clauses*). A construct consists of a directive and, often, additional base language code. In C++ directives are formed from either pragmas or attributes.

OpenMP: `pragma omp`

```
#pragma omp directive-specification
```

The **number of OpenMP threads can be set** using:

- At compilation time: using the environment variable `OMP_NUM_THREADS`
- At runtime: using the function Doc.

  OpenMP: `omp_set_num_threads`

  ```
  void omp_set_num_threads(int num_threads)
  ```

Other useful function to get information about threads:

- The **number of threads in the current team**: Doc.

  OpenMP: `omp_get_num_threads`

  ```
  int omp_get_num_threads()
  ```

  The binding region for an `omp_get_num_threads` region is the innermost enclosing `parallel` region. If called from the sequential part of a program, this routine returns 1.

- The **upper bound on the number of threads** that could be used to form a new team if a `parallel` construct without a `num_threads` clause were encountered after execution returns from this routine. Doc.

OpenMP: `omp_get_max_threads`

```
int omp_get_max_threads()
```

- The **thread number of the calling thread**, within the current team. Doc.

OpenMP: `omp_get_thread_num`

```
int omp_get_thread_num()
```

- The **elapsed wall clock time in seconds**. Doc.

OpenMP: `omp_get_wtime`

```
double omp_get_wtime()
```

- The **precision of the timer (seconds between ticks)** used by `omp_get_wtick`. Doc.

OpenMP: `omp_get_wtick`

```
double omp_get_wtick()
```

OpenMP programs execute serially until they reach a `parallel` directive. As we have explained at page 55, the thread that was executing the code spawns a group of "slave" threads and becomes the "master" (thread ID 0). The code in the structured block is replicated, each thread executes a copy. At the end of the block there is an implied barrier, only the "master" thread continues.

OpenMP: `pragma omp parallel`

```
#pragma omp parallel optional-clauses
```

The parallel directive has **optional** clauses, the most commonly used are:

- Specify the **number of threads to spawn**:

```
#pragma omp parallel num_threads(int)
```

- Conditional parallelization with:

```
#pragma omp parallel if (condition)
```

Example 1: parallel if condition

```
#include <stdio.h>
#include <omp.h>

void test(int val)
{
    #pragma omp parallel if (val != 0)
    if (omp_in_parallel()) {
        #pragma omp single
        printf_s(
            "val = %d, parallelized with %d threads\n",
            val, omp_get_num_threads()
        );
    } else {
        printf_s("val = %d, serialized\n", val);
    }
}

int main( )
{
    omp_set_num_threads(2);
    test(0);
    test(2);
}
```

The output will be:

```
val = 0, serialized
val = 2, parallelized with 2 threads
```

- Data scope clauses (explained in the following pages).

The **number of threads in a parallel region is determined by the following factors**, in order of priority (high to low):

1. Evaluation of the `if` clause;
2. Value of the `num_threads` clause;
3. Use of the `omp_set_num_threads()` library function;
4. Setting of the `OMP_NUM_THREADS` environment variable;
5. Implementation default, e.g., the number of CPUs on a node.

5.3 Work sharing

Work-sharing constructs divide the execution of a region of code among the team members who encounter it. A work-sharing construct must be enclosed in a parallel region for the directive to be executed in parallel. Note that the **constructs do not start new threads**. Also, there is **no implicit barrier at the *entry*** of a work-sharing construct, but **there is an implicit barrier at the *exit*** of a work-sharing construct.

5.3.1 For

The for directive **shares iterations of a loop across the team** (data parallelism).

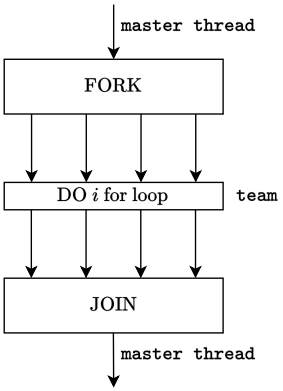


Figure 14: OpenMP for loop.

OpenMP: `pragma omp for`

```
#pragma omp parallel
{
    #pragma omp for
    /* for loop */
}
```

The for directive parallelize execution of iterations. The number of iteration cannot be internally modified. Some common clauses are:

- *`schedule`* that describes **how iterations of the loop are distributed among the threads in the team**. The schedule type can be either `dynamic`, `guided`, `runtime`, or `static`.
  - *`static`*. Loop **iterations are divided into blocks of size** `chunk` and then statically allocated to threads. If `chunk` is **not specified**, the **iterations are divided evenly** (if possible) **among the threads**.

OpenMP: `static` schedule

```
#pragma omp for schedule(static, chunk-size)
```

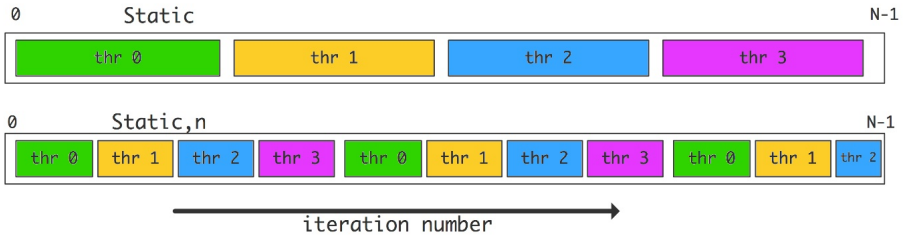


Figure 15: `static` schedule.

– *`dynamic`*. Loop **iterations are divided into blocks of size** `chunk` and **distributed among the threads *at runtime***; when a thread completes one chunk, it is **dynamically allocated another**. The default chunk size is 1. In fact, we can see in the image that the order is not always the same.

OpenMP: `dynamic` schedule

```
#pragma omp for schedule(dynamic, chunk-size)
```

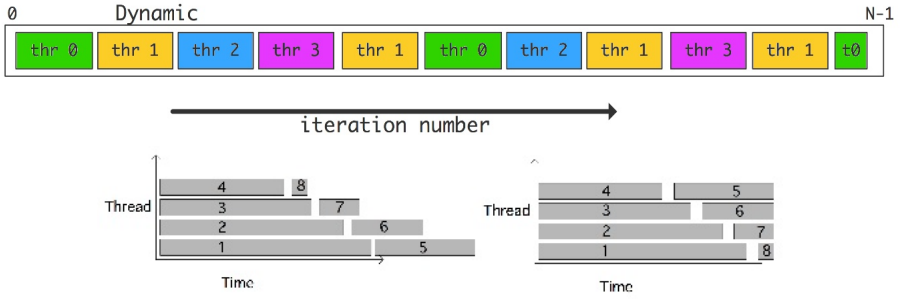


Figure 16: `dynamic` schedule.

– *`runtime`*. Depends on environment variable `OMP_SCHEDULE`.

– *`guided`*. Static, gradually decreases the chunk size (`chunk` specifies the smallest one).

OpenMP: `guided` schedule

```
#pragma omp for schedule(guided)
```

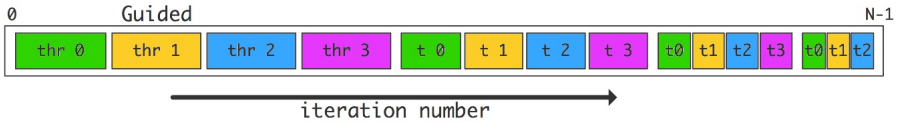


Figure 17: `guided` schedule.

Example 2: `schedule` types

```
#include <stdio.h>
#include <omp.h>

#define NUM_THREADS 4
#define STATIC_CHUNK 5
#define DYNAMIC_CHUNK 5
#define NUM_LOOPS 20
#define SLEEP_EVERY_N 3

int main( )
{
    int nStatic1[NUM_LOOPS],
        nStaticN[NUM_LOOPS];
    int nDynamic1[NUM_LOOPS],
        nDynamicN[NUM_LOOPS];
    int nGuided[NUM_LOOPS];

    omp_set_num_threads(NUM_THREADS);

    #pragma omp parallel
    {
        #pragma omp for schedule(static, 1)
        for (int i = 0 ; i < NUM_LOOPS ; ++i)
        {
            if ((i % SLEEP_EVERY_N) == 0)
                Sleep(0);
            nStatic1[i] = omp_get_thread_num( );
        }

        #pragma omp for schedule(static, STATIC_CHUNK)
        for (int i = 0 ; i < NUM_LOOPS ; ++i)
        {
            if ((i % SLEEP_EVERY_N) == 0)
                Sleep(0);
            nStaticN[i] = omp_get_thread_num( );
        }

        #pragma omp for schedule(dynamic, 1)
        for (int i = 0 ; i < NUM_LOOPS ; ++i)
        {
            if ((i % SLEEP_EVERY_N) == 0)
                Sleep(0);
            nDynamic1[i] = omp_get_thread_num( );
        }

        #pragma omp for schedule(dynamic, DYNAMIC_CHUNK)
        for (int i = 0 ; i < NUM_LOOPS ; ++i)
        {
            if ((i % SLEEP_EVERY_N) == 0)
```

```
50                  Sleep(0);
51              nDynamicN[i] = omp_get_thread_num( );
52          }
53
54          #pragma omp for schedule(guided)
55          for (int i = 0 ; i < NUM_LOOPS ; ++i)
56          {
57              if ((i % SLEEP_EVERY_N) == 0)
58                  Sleep(0);
59              nGuided[i] = omp_get_thread_num( );
60          }
61      }
62
63      printf_s("
        -------------------------------------------------\n")
        ;
64      printf_s("| static | static | dynamic | dynamic |
        guided |\n");
65      printf_s("|    1   |    %d   |    1    |    %d    |
              |\n",
66                  STATIC_CHUNK, DYNAMIC_CHUNK);
67      printf_s("
        -------------------------------------------------\n")
        ;
68
69      for (int i=0; i<NUM_LOOPS; ++i)
70      {
71          printf_s("|    %d   |    %d   |    %d    |    %d
            |"
72                      "    %d   |\n",
73                      nStatic1[i], nStaticN[i],
74                      nDynamic1[i], nDynamicN[i], nGuided[
        i]);
75      }
76
77      printf_s("
        -------------------------------------------------\n")
        ;
78  }
```

The result will be:

```
1  -------------------------------------------------
2  | static | static | dynamic | dynamic | guided |
3  |    1   |    5   |    1    |    5    |        |
4  -------------------------------------------------
5  |    0   |    0   |    0    |    2    |    1   |
6  |    1   |    0   |    3    |    2    |    1   |
7  |    2   |    0   |    3    |    2    |    1   |
8  |    3   |    0   |    3    |    2    |    1   |
9  |    0   |    0   |    2    |    2    |    1   |
10 |    1   |    1   |    2    |    3    |    3   |
11 |    2   |    1   |    2    |    3    |    3   |
12 |    3   |    1   |    0    |    3    |    3   |
13 |    0   |    1   |    0    |    3    |    3   |
14 |    1   |    1   |    0    |    3    |    2   |
15 |    2   |    2   |    1    |    0    |    2   |
16 |    3   |    2   |    1    |    0    |    2   |
17 |    0   |    2   |    1    |    0    |    3   |
18 |    1   |    2   |    2    |    0    |    3   |
19 |    2   |    2   |    2    |    0    |    0   |
```

```
|     3     |     3     |     2     |     1     |     0     |
|     0     |     3     |     3     |     1     |     1     |
|     1     |     3     |     3     |     1     |     1     |
|     2     |     3     |     3     |     1     |     1     |
|     3     |     3     |     0     |     1     |     3     |
-------------------------------------------------
```

- *nowait* to **avoid synchronization at the end of the parallel loop**. It overrides the barrier implicit in a directive.

```
#pragma omp for nowait
```

Example 3: nowait clause

```
#include <stdio.h>

#define SIZE 5

void test(int *a, int *b, int *c, int size)
{
    int i;
    #pragma omp parallel
    {
        #pragma omp for nowait
        for (i = 0; i < size; i++)
            b[i] = a[i] * a[i];

        #pragma omp for nowait
        for (i = 0; i < size; i++)
            c[i] = a[i]/2;
    }
}

int main( )
{
    int a[SIZE], b[SIZE], c[SIZE];
    int i;

    for (i=0; i<SIZE; i++)
        a[i] = i;

    test(a,b,c, SIZE);

    for (i=0; i<SIZE; i++)
        printf_s("%d, %d, %d\n", a[i], b[i], c[i]);
}
```

The output will be:

```
0, 0, 0
1, 1, 0
2, 4, 1
3, 9, 1
4, 16, 2
```

5.3.1.1 Reduction

In parallel programming, sometimes there are some exceptional cases when we use the `for` statement where the variables inside the code block are not so easy to manage (memory viewpoint). For example, consider the following case:

```c
#include "stdio.h"
#include "omp.h"
#define MAX 10


int main(int argc, char const *argv[])
{
    double ave = 0.0;
    double A[MAX] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int i;
    #pragma omp parallel for
    for(i = 0; i < MAX; i++) {
        ave += A[i];
    }
    ave = ave / MAX;
    printf("Value %f\n", ave);
    return 0;
}
```

⚠ Too many threads modifying the same variable

In this case, we are combining values into a single accumulation variable (called `ave`). There is a true dependence between loop iterations that can't be trivially removed. A **continuous execution produces different results!**

```
$ ./example.out
Value 1.300000
$ ./example.out
Value 2.400000
$ ./example.out
Value 2.500000
$ ./example.out
Value 2.100000
$ ./example.out
Value 1.900000
```

This is a very common situation and a solution, which can also be **used as a synchronization technique**, is called a **reduction**.

A reduction variable in a loop **aggregates** (i.e., accumulates) a **value that depends on each iteration of the loop** and **doesn't depend on the iteration order**.

OpenMP: `reduction`

```c
#pragma omp parallel for reduction(operator: list)
```

A reduction clause:

- It **makes a local copy** of each *list* variable and initialized depending on the *operator*;

- It **updates occur on the local copy**;
- **Local copies are reduced into a single value and combined with the original global value**.

Therefore, the variables in *list* must be shared in the enclosing parallel region.

Many different associative operands (*operator* value) can be used with reduction:

- `+` with initial value 0
- `*` with initial value 1
- `-` with initial value 0
- `min` with initial value as largest positive number
- `max` with initial value as most negative number
- `&` with initial value ~ 0
- `|` with initial value 0
- `^` with initial value 0
- `&&` with initial value 1
- `||` with initial value 0

Using the `reduction` clause, the code written at the beginning of the paragraph can be fixed as follows:

```c
#include "stdio.h"
#include "omp.h"
#define MAX 10


int main(int argc, char const *argv[])
{
    double ave = 0.0;
    double A[MAX] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int i;
    // use reduction
    #pragma omp parallel for reduction(+: ave)
    for(i = 0; i < MAX; i++) {
        ave += A[i];
    }
    ave = ave / MAX;
    // now it prints the correct result 5.5
    printf("Value %f\n", ave);
    return 0;
}
```

✔ **Avoid race condition**

5.3.2 Sections

Section identifies code **sections to be divided among all threads**.

Sections allow to specify that the enclosed section(s) of code are to be executed in parallel. **Each section is executed once by a thread in the team.** Doc.

OpenMP: sections

```
#pragma omp [parallel] sections [clauses]
{
    #pragma omp section
    {
        code_block
    }
}
```

The sections directive identifies a noniterative work-sharing construct that specifies a set of constructs that are to be divided among threads in a team. Each section is executed once by a thread in the team.

Each section is preceded by a `section` directive, although the `section` directive is optional for the first section. The `section` directives must appear within the lexical extent of the `sections` directive. There's an **implicit barrier** at the end of a `sections` construct, unless a `nowait` is specified.

Restrictions to the sections directive are as follows:

- A `section` directive must not appear outside the lexical extent of the `sections` directive.
- Only a single `nowait` clause can appear on a `sections` directive.

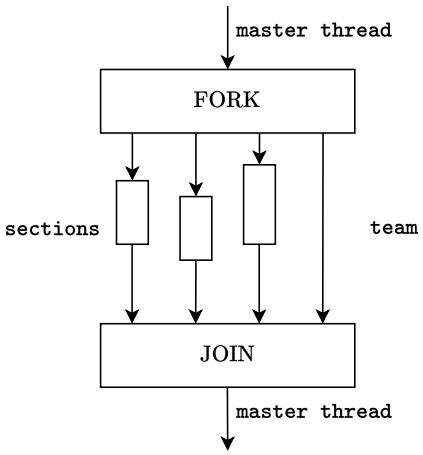


Figure 18: Breaks work into separate, discrete sections, each executed by a thread (functional parallelism).

5.3.3 Single/Master

A **section** (not the directive) **of code should be executed on a single thread, not necessarily the main (master) thread**. The `single` directive identifies a construct that specifies that the associated structured block is executed by only one thread in the team (not necessarily the master thread).

OpenMP: `single` and `master`

```
#pragma omp parallel
{
    #pragma omp single
    {
        /* code section */
    }
    # pragma omp master
    {
        /* code section */
    }
}
```

- `single` specifies that a section of a code is **executed only by a single thread**. Doc.
- `master` specifies that a section of a code is **executed only by the master**. Doc.

There's an **implicit barrier** after the `single` construct unless a `nowait` clause is specified.

5.3.4 Tasks

The following section has been enhanced with slides from Senior Principal Engineer Mattson Tim. He's a senior principal engineer at Intel, where he's been since 1993. His profile can be seen here and the slides are available online here. He has also made an interesting YouTube series on the introduction to OpenMP.

Tasks are **independent units of work**. They consist of: *code to execute*, *data environment*, and *internal control variables* (ICV). **Threads perform the work of each task**. The runtime **system decides when to execute tasks**; each task can be deferred or executed immediately.

In other words, an OpenMP **task is a block of code contained in a parallel region that can be executed simultaneously with other tasks in the same region**.

Some useful terminology:

- ***Task construct***. It identifies the **task directive plus the structured block**.
- ***Task***. It is the **package of code and instructions** for **allocating data** created when a **thread encounters a task construct**.
- ***Task region***. It is the dynamic sequence of **instructions generated by the execution of a task by a thread**.

Tasks are guaranteed to complete at thread barriers (using the `barrier` directive) or at task barriers (using the `taskwait` directive):

```
#pragma omp parallel // omp directive to parallel the code
{
    #pragma omp task // multiple foo tasks created here,
                     // one for each thread
    foo();
    #pragma omp barrier // all foo tasks guaranteed
                        // to be completed here
    #pragma omp single  // only one thread can access to
                        // this piece of code
    {
        #pragma omp task // one bar task created here
        bar();
    }
    // foo task guaranteed to be completed here
}
```

Example 4: Fibonacci with tasks

Let us see a Fibonacci example of data scoping using the tasks. In the following code, we create the Fibonacci function and we create two tasks, but each task has a private variable and these variables are also used in the return statement:

```
int fib(int n) {
    int x, y;
    if(n < 2)
```

```
        return n;
    #pragma omp task
    x = fib(n-1);
    #pragma omp task
    y = fib(n-2);
    #pragma omp taskwait
    return x + y;
}
```

A good solution is to "share" the `x` and `y` variables because we need both values to calculate the sum.

```
int fib(int n) {
    int x, y;
    if(n < 2)
        return n;
    #pragma omp task shared(x)
    x = fib(n-1);
    #pragma omp task shared(y)
    y = fib(n-2);
    #pragma omp taskwait
    return x + y;
}
```

✔ Main advantage

Note the following code:

```
// create a team of threads
#pragma omp parallel
{
    // one thread executes the single construct
    // and other threads wait at the implied
    // barrier at the end of the single construct
    #pragma omp single
    { // block 1
        node *p = head;
        while(p) { // block 2
            // the single thread creates a task
            // with its own value for the pointer p
            #pragma omp task firstprivate(p)
                process(p);
            p = p -> next; // block 3
        }
        // execution moves beyond the barrier
        // once all the tasks are complete
    }
}
```

The tasks have the **potential to parallelize irregular patterns and recursive function calls**. See Figure 19 (page 70) to understand how the runtime system can optimize execution using the tasks.

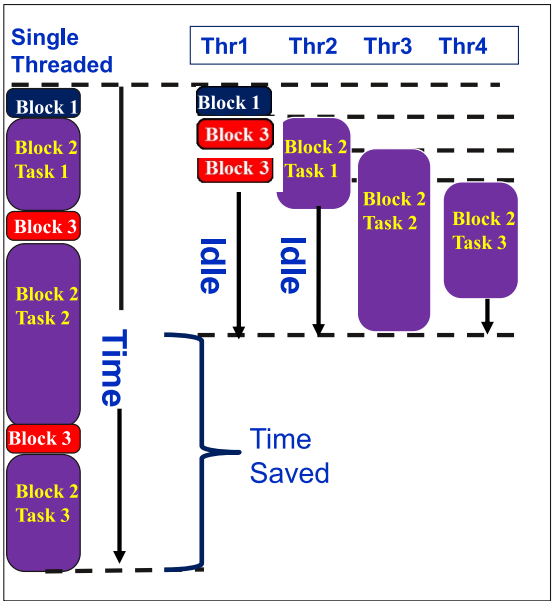


Figure 19: The main advantage of the tasks is to parallelize irregular patterns and recursive function calls.

Synchronization

When a thread encounters a `task` construct, the task is created but not immediately executed. The tasks are guaranteed to be completed:

- At a **barrier** (implicit or explicit)
- At **task synchronization points**:
  - `taskwait` construct specifies a **wait on the completion of child tasks of the current task**. Doc.

OpenMP: `pragma omp taskwait`

```
1 #pragma omp taskwait
```

  - `taskgroup` construct specifies a **wait on the completion of child tasks of the current task** (such as `taskwait`) and **their descendent tasks** (main difference). Doc.

OpenMP: `pragma omp taskgroup`

```
1 #pragma omp taskgroup
```

5.3.4.1 Task dependences

By creating tasks instead of sections, each thread can execute any task as long as its input is ready. Since the internal scheduler decides how to manipulate the execution of tasks, there is a useful **clause to indicate a dependency between tasks**. Doc.

OpenMP: `depend`

```
#pragma omp task depend(dependence-type: variable)
```

The `depend` clause enforces additional constraints on the scheduling of tasks or loop iterations. These constraints establish dependencies only between sibling tasks or between loop iterations.
The *dependence-type* can be `in` or `out`.

In addition to the depend clause, we can also use the priority clause to hint that more important tasks should be executed more frequently.

The `priority` clause is a **hint for the priority of the generated task**. The priority-value is a **non-negative integer expression** that provides a hint for task execution order. Among all tasks ready to be executed, **higher priority tasks** (those with a higher numerical value in the priority clause expression) **are recommended to execute before lower priority ones**. The **default** priority-value when no priority clause is specified **is zero** (the *lowest priority*).

If a value is specified in the priority clause that is higher than the *max-task-priority-var* ICV (internal control variables) then the implementation will **use the value of that ICV**. A program that relies on task execution order being determined by this priority-value may have unspecified behavior.

- The `omp_get_max_task_priority` routine returns the **maximum value that can be specified in the `priority` clause**. Doc.

  OpenMP: `omp_get_max_task_priority`

  ```
  int omp_get_max_task_priority()
  ```

- The `OMP_MAX_TASK_PRIORITY` environment variable **controls the use of task priorities** by setting the initial value of the *max-task-priority-var* ICV. The value of this environment variable must be a non-negative integer. Doc.

Possible overhead

The task in general, but especially the **dependencies between tasks, can introduce overhead and reduce performance**.

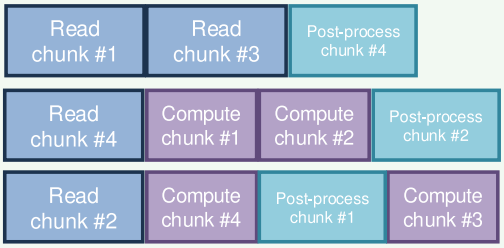
Example 5: dependences

Let the following parallel code:

```
1  #pragma omp parallel default(none) \
2                       shared(fp_read) \
3                       shared(n_io_chunks) \
4                       shared(n_work_chunks) \
5                       shared(a, b, c) \
6                       shared(status_read, status_processing) \
7                       shared(status_postprocessing)
8  {
9      #pragma omp single nowait
10     {
11         for(int64_t i = 0; i < n_io_chunks; ++i) {
12             #pragma omp task depend(out: status_read[i]) \
13                              priority(20)
14             {
15                 (void) read_input(
16                     fp_read, i, a, b, &status_read[i]
17                 );
18             } // End of task reading in a chunk of data
19
20             #pragma omp task depend(in: status_read[i]) \
21                              depend(out: status_processing[i]) \
22                              priority(10)
23             {
24                 (void) compute_results(
25                     i, n_work_chunks, a, b, c,
26                     &status_processing[i]
27                 );
28             } // End of task performing the computations
29
30             #pragma omp task depend(in: status_processing[i])
31                              priority(5)
32             {
33                 (void) postprocess_results(
34                     i, n_work_chunks, c,
35                     &status_postprocessing[i]
36                 );
37             } // End of task postprocessing the results
38         } // End of for-loop
39     } // End of single region
40 } // End of parallel region
41
```

- Row 11. We are going to process `n_io_chunks` of data.
- Rows 12, 20. Refer to read and are compute dependence.
- Rows 21, 30. Refer to compute and are postprocess dependence.

A possible order of execution if we consider 3 threads is:



As long as the dependencies are respected, the loop iterations can be executed in any order. No explicit `flush` (page 88) are required because it is implied before and after every task.

Custom number of iterations

Each task gets assigned a number of iterations which is the minimum between `grainsize` and the total number of iterations.

If a `grainsize` clause is present, the number of logical iterations assigned to each generated task is greater than or equal to the minimum of the value of the *`grain-size`* expression and the number of logical iterations, but less than two times the value of the *`grain-size`* expression.

If the `grainsize` clause has the *`strict`* modifier, the number of logical iterations assigned to each generated task is equal to the value of the *`grain-size`* expression, except for the generated task that contains the sequentially last iteration, which may have fewer iterations. The parameter of the `grainsize` clause must be a positive integer expression. Doc.

OpenMP: `pragma omp taskloop grainsize`

```
1 #pragma omp taskloop grainsize([strict] grain-size)
```

Number of created tasks

To keep the number of created tasks low, the clause `num_tasks` sets the number of tasks that the runtime system can generate.

If `num_tasks` is specified, the `taskloop` construct creates as many tasks as the minimum of the *num-tasks* expression and the number of logical iterations. Each task must have at least one logical iteration. The parameter of the `num_tasks` clause must be a positive integer expression.

If the `num_tasks` clause has the *strict* modifier for a task loop with N logical iterations, the logical iterations are partitioned in a balanced manner and each partition is assigned, in order, to a generated task. The partition size is $\left\lceil \frac{N}{\textit{num-tasks}} \right\rceil$ until the number of remaining iterations divides the number of remaining tasks evenly, at which point the partition size becomes $\left\lfloor \frac{N}{\textit{num-tasks}} \right\rfloor$. Doc.

OpenMP: `pragma omp taskloop num_tasks`

```
1 #pragma omp taskloop num_tasks([strict] num-tasks)
```

If neither a `grainsize` nor `num_tasks` clause is present, the number of loop tasks generated and the number of logical iterations assigned to these tasks is implementation defined.

5.4 Synchronization

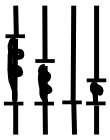
The following section has been enhanced with slides from Senior Principal Engineer Mattson Tim. He's a senior principal engineer at Intel, where he's been since 1993. His profile can be seen here and the slides are available online here. He has also made an interesting YouTube series on the introduction to OpenMP.

OpenMP is a multi-threaded, shared address model. This means that threads communicate by sharing variables. Unfortunately, this can cause some problems such as race conditions (page 47). The solution is to **use synchronization to protect against data conflicts**. The good news is that synchronization can avoid data race problems, but it is also an **expensive method**. So we can use synchronization, but we **need to change the way data is accessed to minimize the need for synchronization**.

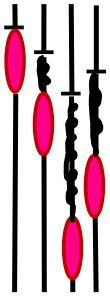
Synchronization brings **one or more threads to a well-defined and known point in their execution**. The two most common forms of synchronization are:

- ***Barrier***: each thread wait at the barrier **until all threads arrive**.
- ***Mutual exclusion***: define a block of code that **only one thread at a time can execute**.

The OpenMP directives are: `critical`, `atomic`, and `barrier`. These are high-level synchronization directives, but there are also low-level synchronization directives, such as `flush` and `locks`, but they are too complex at the moment, we will see them later.



(a) Barrier. Each thread wait at the barrier until all threads arrive.



(b) Mutual exclusion. Define a block of code that only one thread at a time can execute.

Barrier. Each thread waits until all threads arrive. Doc.

OpenMP: `barrier`

```
#pragma omp barrier name
```

An optional `name` may be used to identify the `critical` region. A **thread waits at the beginning of a critical region** until no other thread is executing a critical region (anywhere in the program) **with the same name**. **All unnamed `critical` directives map to the same unspecified name**.

Example 6: synchronization with `barrier` directive

The following example includes several critical directives. The example illustrates a queuing model in which a task is dequeued and worked on. To guard against many threads dequeuing the same task, the dequeuing operation must be in a `critical` section. Because the two queues in this example are independent, they're protected by `critical` directives with different names, *xaxis* and *yaxis*. [9]

```
#pragma omp parallel shared(x, y) private(x_next, y_next)
{
    #pragma omp critical ( xaxis )
        x_next = dequeue(x);
    work(x_next);
    #pragma omp critical ( yaxis )
        y_next = dequeue(y);
    work(y_next);
}

```

Mutual exclusion. Only one thread at a time can enter a *critical* region. Doc.

OpenMP: `critical`

```
#pragma omp critical
```

	`omp critical`	`omp single`
Meaning	Run code segment one by one by all threads	Run code segment once by any thread
Number of times code is executed	Number of threads	Only one
Use case	Avoid race condition	Manage control variables or signals

Table 3: `omp critical` vs `omp single`

Example 7: synchronization with `critical` directive

Note the following code:

```
float res;

#pragma omp parallel
{
    float B;
    int i, id, nthrds, niters = big_number;

    id = omp_get_thread_num();
    nthrds = omp_get_num_threads();
    for (i = id; i < niters; i += nthrds) {
        B = big_job(i);
        #pragma omp critical // threads wait their turn;
        res += consume(B);   // only one at a time
    }                        // calls consume
}

```

Atomic. Provides mutual exclusion, but only when updating a memory location. It **ensures that a particular memory location is accessed atomically**. It is **valid only for the following statement** and not for a structured block.

The statement inside the atomic must be one of the following forms:

- `x binop = expr`
- `x++` or `++x`
- `x--` or `--x`

Where `x` is an lvalue of scalar type and `binop` is a non-overloaded builtin operator. Doc.

OpenMP: `atomic`

```
#pragma omp atomic
```

Example 8: synchronization with `atomic` directive

```
#pragma omp parallel
{
    double tmp, B;
    B = DOIT();
    tmp = big_ugly(B);
    #pragma omp atomic
    X += tmp;
}

```

5.5 Data environment

OpenMP is based on the shared memory programming model, so most **variables are shared by default**. **Global variables are also shared between threads**. But not everything is shared; for example, *stack variables* in functions called from parallel regions are **private**, as are *automatic variables* within a statement block.

Example 9: data sharing

In the following code, the variable `temp` is private (local to each thread) because it is in the stack of the function `work`; meanwhile, the variables `A`, `index`, and `count` are shared by all threads.

```
double A[10];
int main() {
    int index[10];
    #pragma omp parallel
        work(index);
    printf("%d\n", index[0]);
}

void work(int *index) {
    double temp[10];
    static int count;
    /* other code */
}
```

We can refer to these arguments as **Data Scope Attribute Clauses** because the issue is really about the visibility and value of each data in each scope. Although OpenMP shares variables by default, there is (and it is a very common and *best practice*) the option to:

- Selectively **change storage attributes for construct** using the following clauses: `shared`, `private` and `firstprivate`.
- The **final value** of a private inside a parallel loop can be **transmitted to the shared variable outside the loop** with: `lastprivate`.
- The **default attributes can be overridden** with:
  `default(private | shared | none)`

⚠ Note that when we say "copy" we mean the *shallow copy*, not the *deep copy*. So when the following clauses create a local copy, they create a *shallow copy*.

`private` **clause**. The statement `private(var)` creates a **new local copy** of `var` **for each thread**. The value of the private copies is ***uninitialized*** and also the **original variable value remains unchanged after the region**.

OpenMP: `private(...)`

```
#pragma omp parallel directive private(var1-name, var2-name, ...)
```

Example 10: `private` clause and *dirty memory location*

In the following code we try to use the private clause inside a parallel for. The code is very trivial, we set the number of threads to two for better understanding, so we start the parallel code with the for directive and the private clause. It has `private_test` as a private variable. So each thread will copy the variable and the initial value will be undefined.

```
#include <iostream>
#include "omp.h"
#define MAX 6

int main(int argc, char const *argv[])
{
    int i, private_test = 10;
    printf("Memory location of private_test: %p\n",
           &private_test);
    // set limit to 2 threads for better understanding
    omp_set_num_threads(2);
    printf("Master will execute for in parallel!\n");
    #pragma omp parallel for private(private_test)
    for(i = 0; i < MAX; ++i) {
        // initialize private_test
        private_test = i == 0 ? 0 : ++private_test;
        printf(
            "Thread #%i, iter_i: %d, private_test: %d\n",
            omp_get_thread_num(), i, private_test
        );
    }
    printf(
        "private_test outside the parallel region: %d\n",
        private_test
    );
    return 0;
}
```

Unfortunately, when we examine the output, we see a problem. Thread zero (*master*) executes the `for` statement for the first 3 iterations, while thread one (*slave*) executes the `for` statement for the last 3 iterations. The zero thread behaves as expected because we initialize the `private_test` variable to zero on the first `for` iteration (when `i` is 0). The thread one, has made a copy of the variable in another memory location and the value is unknown; so it continues to add a single value to each iteration in a *dirty memory location*. This is a trivial example that highlights the unknown values that we can find inside the private variables if we don't do any initialization.

```
$ g++ -fopenmp example.cpp -o example
$ ./example
Memory location of private_test: 0x7ffecdfa3aa4
Master will execute the for statement in parallel!
Thd #0, i:0, private_test:0,         mem: 0x7ffecdfa3a40
Thd #0, i:1, private_test:1,         mem: 0x7ffecdfa3a40
Thd #0, i:2, private_test:2,         mem: 0x7ffecdfa3a40
Thd #1, i:3, private_test:687869953, mem: 0x76a0297ffdd0
Thd #1, i:4, private_test:687869954, mem: 0x76a0297ffdd0
Thd #1, i:5, private_test:687869955, mem: 0x76a0297ffdd0
private_test outside the parallel region: 10
```

`firstprivate` **clause**. The **variables are initialized from the shared variable**, but as in the `private` clause, the updated value doesn't leave the parallel region.

OpenMP: `firstprivate(...)`

```
#pragma omp parallel directive firstprivate(var1-name, ...)
```

Example 11: `firstprivate` **clause**

A very trivial example to see how the firstprivate clause works. The variable `private_test` is copied to local and initialized with the value outside the parallel region.

```
#include <iostream>
#include "omp.h"
#define MAX 6

int main(int argc, char const *argv[])
    int i, private_test = 10;
    printf("Memory location of private_test: %p\n",
           &private_test);
    // set limit to 2 threads for better understanding
    omp_set_num_threads(2);
    printf("Master will execute for in parallel!\n");
    #pragma omp parallel for firstprivate(private_test)
    for(i = 0; i < MAX; ++i) {
        // initialize private_test
        ++private_test;
        printf(
            "Thd #%i, i:%d, private_test:%d, mem: %p\n",
            omp_get_thread_num(), i,
            private_test, &private_test
        );
    }
    printf(
        "private_test outside the parallel region: %d\n",
        private_test
    );
    return 0;
}
```

Note that the value is not propagated outside the parallel region.

```
$ g++ -fopenmp example.cpp -o example
$ ./example
Memory location of private_test: 0x7ffc5cd153b0
Master will execute for in parallel!
Thd #0, i:0, private_test:11, mem: 0x7ffc5cd15350
Thd #0, i:1, private_test:12, mem: 0x7ffc5cd15350
Thd #0, i:2, private_test:13, mem: 0x7ffc5cd15350
Thd #1, i:3, private_test:11, mem: 0x77001d9ffdd0
Thd #1, i:4, private_test:12, mem: 0x77001d9ffdd0
Thd #1, i:5, private_test:13, mem: 0x77001d9ffdd0
private_test outside the parallel region: 10
```

Example 12: be careful with pointers using the `firstprivate` clause

The following code is very similar to the previous one. The difference here is that the parallel region also gets a pointer. Note that the pointer is in C style, because using the unique pointer technique (suggested in C++) will create an exception, because C++ doesn't allow the copy of a unique pointer. However, each thread creates a shallow copy of the pointer, but not a copy of the value pointed to! In this test, we use the pointer to the value of `private_test` to modify the value of `private_test`.

```
#include <iostream>
#include "omp.h"
#define MAX 6

int main(int argc, char const *argv[]) {
    bool print_flag = true;
    int i, private_test = 10;
    // C pointer, not a good practice in C++...
    // used only for the example
    int *ptr_private_test = &private_test;
    printf("Memory location of private_test    : %p\n",
           &private_test);
    printf("Memory location of ptr_private_test: %p\n",
           &ptr_private_test);
    // set limit to 2 threads for better understanding
    omp_set_num_threads(2);
    printf("Master will execute for in parallel!\n\n");
    #pragma omp parallel for firstprivate(private_test,
    ptr_private_test, print_flag)
    for(i = 0; i < MAX; ++i) {
        if (print_flag) {
            printf("Memory location ptr %p\n",
                   &ptr_private_test);
            print_flag = false;
        }
        // increase value pointed to by ptr
        ++*ptr_private_test;
        // increase simple variable
        ++private_test;
        printf(
            "Thread #%i, i:%d\n- private_test:%d,
    ptr_private_test:%d, mem: %p\n\n",
            omp_get_thread_num(), i, private_test,
            *ptr_private_test, &private_test
        );
    }
    printf(
        "private_test outside the parallel region: %d\n",
        private_test
    );
    return 0;
}
```

Note an interesting observation. The variable `private_test` is incremented at each iteration; in the same way, the value pointed to by the pointer `ptr_private_test` is also incremented. Finally, the variable

`private_test` is modified because the pointer was copied in each thread and each slave, including the master, increased the value. This is a very bad practice and we want to suggest to use unique pointers of C++ or to avoid shallow copies.

```
Memory location of private_test    : 0x7fff3849c224
Memory location of ptr_private_test: 0x7fff3849c228
Master will execute for in parallel!

Memory location ptr 0x7fff3849c1a0
Thread #0, i:0
- private_test:11, ptr_private_test:11, mem: 0x7fff3849c198

Thread #0, i:1
- private_test:12, ptr_private_test:12, mem: 0x7fff3849c198

Thread #0, i:2
- private_test:13, ptr_private_test:13, mem: 0x7fff3849c198

Memory location ptr 0x7b710cfffdb0
Thread #1, i:3
- private_test:11, ptr_private_test:14, mem: 0x7b710cfffda8

Thread #1, i:4
- private_test:12, ptr_private_test:15, mem: 0x7b710cfffda8

Thread #1, i:5
- private_test:13, ptr_private_test:16, mem: 0x7b710cfffda8

private_test outside the parallel region: 16
```

The expected value for `private_test` should remain 10 because the firstprivate doesn't affect the values of the variables after the parallel region, but in this case we are using a pointer and a bad practice.

`lastprivate` **clause**. The variables **update shared variables with the value from the last iteration**, so the order of execution of the threads is important here.

OpenMP: `lastprivate(...)`

```
#pragma omp parallel directive lastprivate(var1-name, ...)
```

Example 13: `lastprivate` **clause**

In the following example, the value of the last iteration is passed (and overwritten) to the original value:

```
#include <iostream>
#include "omp.h"
#define MAX 6

int main(int argc, char const *argv[]) {
```

```
    int i, private_test = 10;
    printf("Memory location of private_test: %p\n",
           &private_test);
    // set limit to 2 threads for better understanding
    omp_set_num_threads(2);
    printf("Master will execute for in parallel!\n");
    #pragma omp parallel for lastprivate(private_test)
    for(i = 0; i < MAX; ++i) {
        // initialize private_test
        private_test = i;
        printf(
            "Thd #%i, i:%d, private_test:%d, mem: %p\n",
            omp_get_thread_num(), i,
            private_test, &private_test
        );
    }
    printf(
        "private_test outside the parallel region: %d\n",
        private_test
    );
    return 0;
}
```

The `private_test` variable initially has a value equal to 10, but with `lastprivate` we have overwritten it.

```
Memory location of private_test: 0x7ffc4c82b8c0
Master will execute for in parallel!
Thd #0, i:0, private_test:0, mem: 0x7ffc4c82b860
Thd #0, i:1, private_test:1, mem: 0x7ffc4c82b860
Thd #0, i:2, private_test:2, mem: 0x7ffc4c82b860
Thd #1, i:3, private_test:3, mem: 0x72c7dddffdd0
Thd #1, i:4, private_test:4, mem: 0x72c7dddffdd0
Thd #1, i:5, private_test:5, mem: 0x72c7dddffdd0
private_test outside the parallel region: 5
```

`default` clause. The default storage attribute is `default(shared)`. To change the default we can write simply the value shared or none inside the brackets:

- `default(share)` is the default choice for OpenMP, so there is no need to use it except for the clause `pragma omp task`.

OpenMP: `default(share)`

```
#pragma omp parallel directive default(share)
```

Against the `private` clauses, if we want to share some variables, we can use `shared` clause.

OpenMP: `shared`

```
#pragma omp parallel directive shared(var1-name, ...)
```

- `default(private)`, each variable in the construct is made private as if specified in private clause.

OpenMP: `default(private)`

```
#pragma omp parallel directive default(private)
```

- `default(none)`, no default for variables in static extent. Must list storage attribute for each variable in static extent. It is a good programming practice.

OpenMP: `default(none)`

```
#pragma omp parallel directive default(none)
```

Example 14: `default(none)`

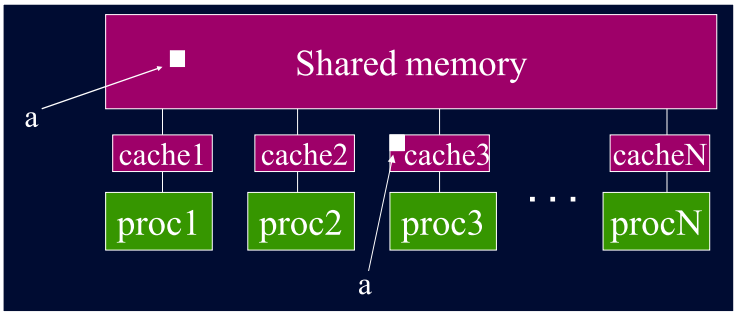
```cpp
#include <iostream>
#include "omp.h"
#define MAX 6

int main(int argc, char const *argv[]) {
    double private_test = 10.0;
    int i;
    // set limit to 2 threads for better understanding
    omp_set_num_threads(2);
    #pragma omp parallel for default(none) private(i,
    private_test)
    for(i = 0; i < MAX; i++) {
        private_test = i;
        printf(
            "Thread #%i, value: %f\n",
            omp_get_thread_num(), private_test
        );
    }
    printf(
        "private_test outside the parallel region: %f\n"
    ,
        private_test
    );
    return 0;
}
```

```
Thread #0, value: 0.000000
Thread #0, value: 1.000000
Thread #0, value: 2.000000
Thread #1, value: 3.000000
Thread #1, value: 4.000000
Thread #1, value: 5.000000
private_test outside the parallel region: 10.000000
```

5.6 Memory model

OpenMP supports a **shared memory model**. It is a model where **all threads share an address space**, but it can get complicated, for example in the following example (picture) we can see a shared space that is also in the cache of process 3; so how can we manage this situation? What are the methods to adapt? Furthermore, what happens when a thread modifies a shared address space? How does the system behave?



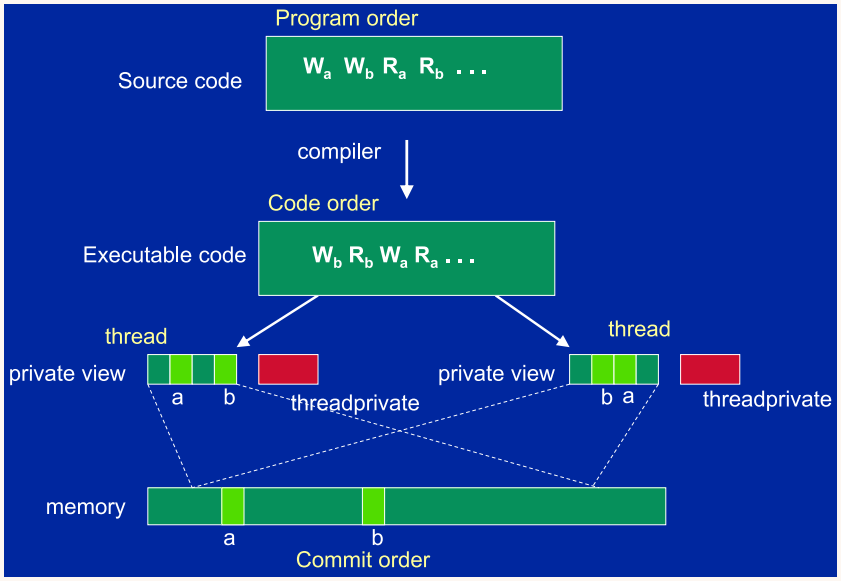
In general, a memory model is defined in terms of:

- **Coherence**: behavior of the memory system when a **single address is accessed by multiple threads**.
- **Consistency**: read, write, or synchronization (RWS) orders with **different addresses and through multiple threads**.

Remark: what the compiler does at the low level

When we write a program and ask the compiler to compile it, it does a lot of "magic" under the hood.

1. Our source code is decoded into very low level operations. These operations respect the order of the high-level operations of our code.
2. The compiler, smarter then us, tries to optimize the code while creating the executable code; so it reorders the low-level operations that are **semantically equivalent** to our program, but often allow to gain performance.
3. During the execution of the code, if it is parallel, some threads are created and each of them has a private memory address.
4. Finally, each thread writes/reads from memory, also called the commit order, using some rules.



Note that the re-ordering are made by:

- **Compile** re-orders *program order* → *code order*.
- **Machine** re-orders *code order* → *memory commit order*.

At any given time, the private view seen by a thread may differ from the view in shared memory. For this reason, there are **consistency models** that **define constraints on the order of RWS** (Reads, Writes, Synchronizations) **operations**.

In general, a multiprocessor adopts the **Sequential Consistency Model**. It says that given n operations (RWS), they are sequentially consistent if:

- They **remain in *program order* for each processor**.
- They are **seen to be in the same overall order by each of the other processors**.

Also, in a sequential consistency model, program order is the same as code order and commit order.

OpenMP uses a **Relaxed Consistency Model** where the **compiler cannot reorder synchronization operations with read or write operations on the same thread**. The consequences are:

- ✓ All threads have the same view of memory at specific points in the code, called ***Consistency Points***.
- ✓ Between two consistency points, each thread has its own temporary view of memory, which may be different from the other temporary views of other threads.

✓ Data are read-only, this guarantee to avoid consistency issues.

✗ Shared data that need to be modified can create possible race conditions.

`flush` **directive**. Defines a *sequence point* at which a **thread is guaranteed to see a consistent view of memory** with respect to the `flush-set`. The `flush-set` means ***all thread visible variables*** for a `flush` construct without an argument list, and it also means a list of variables when the `flush(list)` construct is used. The action of `flush` is to **guarantee** that:

- All read/write **operations that overlap** the `flush-set` and **occur *before*** the `flush` will be **completed before the `flush` is executed**.
- Any read/write **operations that overlap** the `flush-set` and **occur *after*** the `flush` will **not be performed until after the `flush`**.
- Flushes with overlapping `flush-set`s can not be reordered.

In other words, the `flush` **forces data to be updated in memory so that other threads see the most recent value**. Doc.

OpenMP: `flush`

```
1 #pragma omp flush flush-set
```

A flush operation is **implied** by default by OpenMP synchronizations:

- At entry/exit of `parallel`/`critical` regions;
- At implicit and explicit `barrier`s;
- At exit from work-sharing constructs, unless `nowait` is specified.

And it is **not implied**:

- At entry of work-sharing constructs;
- At entry and exit of `master`.

5.7 Nested Parallelism

How does OpenMP create multiple threads?

OpenMP uses a **fork-join model** of parallel execution. When a thread encounters a `parallel` construct, the **thread creates a team composed of itself and some additional** (possibly zero) **number of threads**. The encountering **thread becomes the *master* of the new team**. The **other threads of the team are called *slave*** threads of the team.

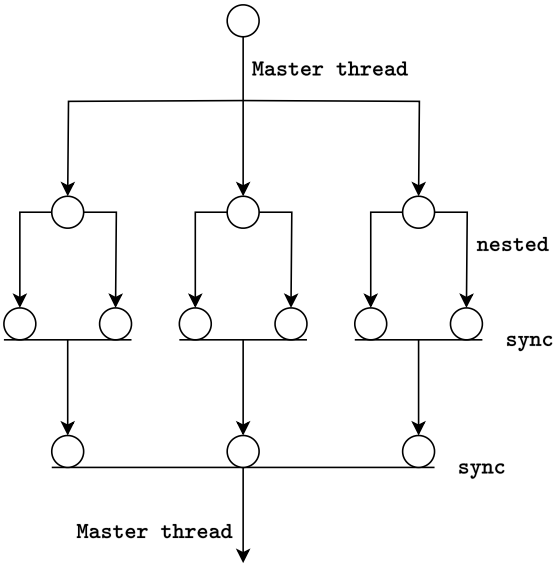


Figure 21: OpenMP fork-join model.

What is the life history of each thread (*slave*) created?

All team members execute the code inside the `parallel` construct. When a thread finishes its work within the parallel construct, it waits at the implicit barrier at the end of the parallel construct. When all team members have arrived at the barrier, the threads can leave the barrier. The *master* thread continues execution of user code beyond the end of the `parallel` construct, while the *slave* threads wait to be summoned to join other teams.

OpenMP **parallel regions can be nested inside each other**. If nested parallelism is:

- **Disabled**, then the **new team** created by a thread encountering a parallel construct inside a parallel region **consists only of the encountering thread**.
- **Enabled**, then the **new team may consist of more than one thread**.

How does OpenMP manage the available threads? Thread Pool

The OpenMP runtime library maintains a pool of threads that can be used as *slave* threads in parallel regions. When a thread encounters a parallel construct and needs to create a team of more than one thread, the thread will check the pool and grab idle threads from the pool, making them *slave* threads of the team. The *master* thread might get fewer *slave* threads than it needs if there is not a sufficient number of idle threads in the pool. When the team finishes executing the parallel region, the *slave* threads return to the pool.

Summary

1. **Parallel construct**. Our main thread starts to execute our code. It encounters a `parallel` construct.
2. **Pool verification**. The OpenMP library checks its pool of threads. If within its pool of available threads have something of disposable, it allocates the *slave* (thread requested) to the *master* (thread that want to create the team). We refer to a single thread, but obviously this can be extended to multiple thread request (e.g. *master* thread requests 3 threads to OpenMP). Finally, the number of requested *slave*s cannot always be satisfied; OpenMP guarantees the best, so it continues to give the requested threads to the applicants. If it cannot satisfy the request, it returns the maximum number of *slave*s it can satisfy (e.g. *master* requests 4 threads, but OpenMP has a pool of only 2 threads available; therefore it returns 2 *slave*s).
3. **Assign and start execution**. Ideally, OpenMP returns the number of threads requested by the *master*. Otherwise, it returns the maximum number. The team now consists of the main thread, called the *master*, and its *slave*s. Each member of the team executes the code specified by the programmer within the parallel construct.
4. **End of execution of a thread**. A thread of the team finishes its work. It can finally rest, and its state changes from "running" to "waiting". It waits for its other thread friends. Each thread has an implicit barrier at the end of the parallel construct.
5. **Any thread finish**. Finally, each thread finishes its work. The *master* releases each member of the team to the OpenMP library. OpenMP updates its thread pool with the returning threads, and the *master* thread continues its life independently from the released threads.

The previous flow works if and only if nested parallelism is enabled. Otherwise, if a *master* asks to create a new team, it will ignore the request and tell the requester to continue alone.

Implementation

Nested parallelism can be enabled or disabled by passing true or false as arguments to the runtime function: Doc.

OpenMP: `omp_set_nested`

```
void omp_set_nested(int nested)
```

We can set a **default number of threads at different levels of nested parallelism** with:

```
OMP_NUM_THREADS = [list, of, integers]
```

If the nesting level is deeper than the number of entries in the list, the last value is used for all subsequent nested parallel region.

- `OMP_MAX_ACTIVATE_LEVELS` defines the upper **limit on the number of active parallel regions that may be nested**.
- `OMP_THREAD_LIMIT` **avoids** that recursive applications **create too many threads**.

Finally, since we are in nested parallelism, the thread number returns the thread number partially and not globally, we need other useful functions:

- Returns the maximum number of OpenMP threads available in contention group: Doc.

  OpenMP: `omp_get_thread_limit`

  ```
  int omp_get_thread_limit()
  ```

- Returns the maximum number of nested active parallel regions when the innermost parallel region is generated by the current task. Doc.

  OpenMP: `omp_get_max_active_levels`

  ```
  int omp_get_max_active_levels()
  ```

- Limits the number of nested active parallel regions when a new nested parallel region is generated by the current task. Doc.

  OpenMP: `omp_set_max_active_levels`

  ```
  void omp_set_max_active_levels(int max_levels)
  ```

- Returns the number of nested parallel regions on the device that enclose tha task containing the call. Doc.

OpenMP: `omp_get_level`

```
int omp_get_level()
```

- Returns the number of active, nested parallel regions on the device enclosing the task containing the call. Doc.

OpenMP: `omp_get_active_level`

```
int omp_get_active_level()
```

- Returns, for given nested level of the current thread, the thread number of the ancestor of the current thread. Doc.

OpenMP: `omp_get_ancestor_thread_num`

```
int omp_get_ancestor_thread_num(int level)
```

- Returns, for a given nested level of the current thread, the size of the thread team to which the ancestor of the current thread belongs. Doc.

OpenMP: `omp_get_team_size`

```
int omp_get_team_size(int level)
```

5.8 Cancellation

As of OpenMP 4.0, a thread can be cancelled using a combination of two commands:

- `cancel` construct cancels the innermost enclosing region of the specified type.

  In other words, it **allows us to cancel the current thread**. But be careful! The directive only allows to **set the cancellation flag to a value of one**. So it is necessary to also use the `cancellation point` command or the thread must hit a barrier (implicit or explicit).

  The syntax of the `cancel` construct is as follows: Doc.

  OpenMP: `pragma omp cancel`

  ```
  1 #pragma omp cancel construct-type-clause
  ```

  Where *`construct-type-clause`* is one of the following: `parallel`, `sections`, `for`, `taskgroup`.

- The `cancellation point` construct introduces a user-defined cancellation point at which implicit or explicit tasks check whether cancellation of the innermost enclosing region of the specified type has been enabled.

  In other words, **when a thread encounters the cancellation point, the cancellation flag has been checked**. The thread execution is stopped. This is an explicit cancellation request because there are other points where cancellation flag is checked, such as at another cancel region or at a barrier. Doc.

  OpenMP: `pragma omp cancellation point`

  ```
  1 #pragma omp cancellation point
  ```

Since there is an **overhead in checking for cancellation**, it can be enabled manually using the `OMP_CANCELLATION` environment variable (false by default).

⚠ Immediate cancellation is not guaranteed

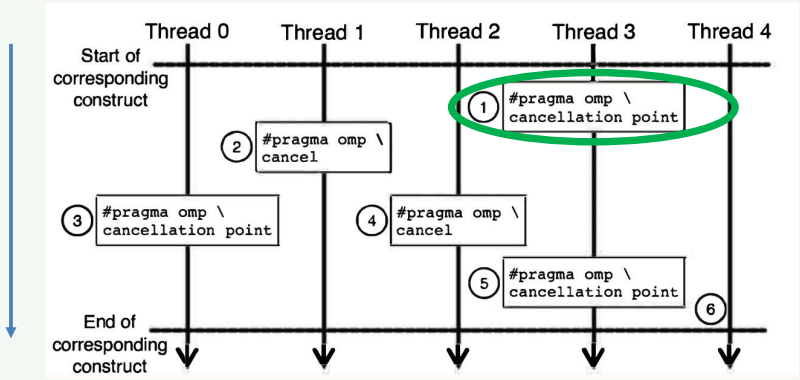
OpenMP does not guarantee that cancellation will result in immediate termination.

✔ When to use?

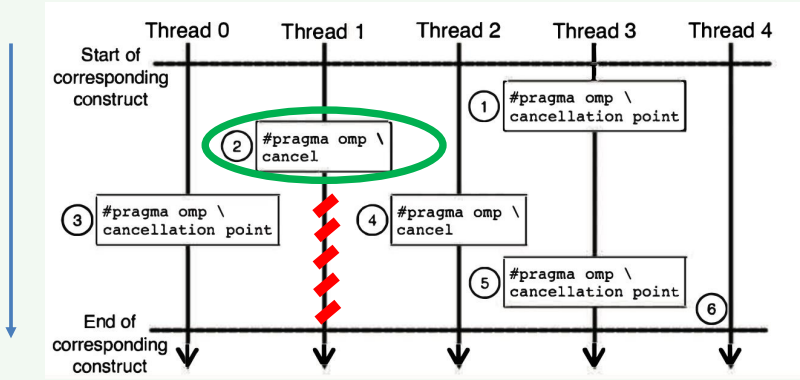
When there is a **need to stop the execution**, such as in the **divide-and-conquer algorithms** (for example, to stop a search when we find the element), or when we **need to handle some errors**.

Example 15: thread cancellation

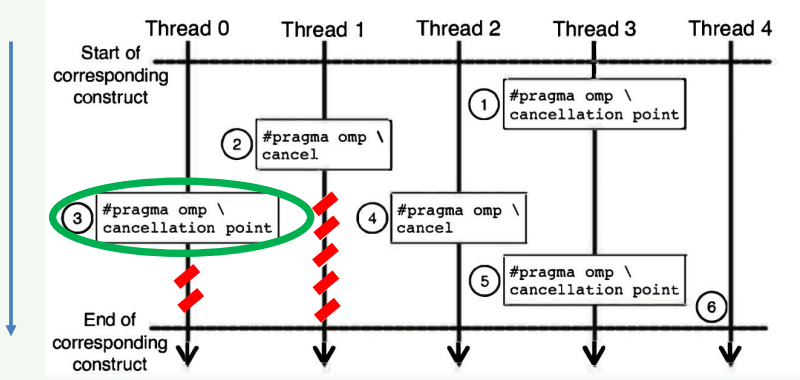
Assume that from a certain point in our algorithm there are 5 threads alive (including the *master*), and the construct pipeline is as follows.



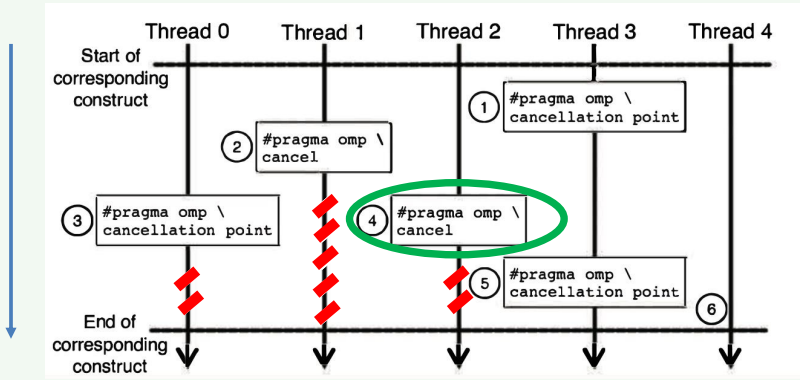
Thread 3 hits a cancellation point, but there is no cancel directive before it, so the cancellation flag is set to zero by default and the thread hasn't been broken.



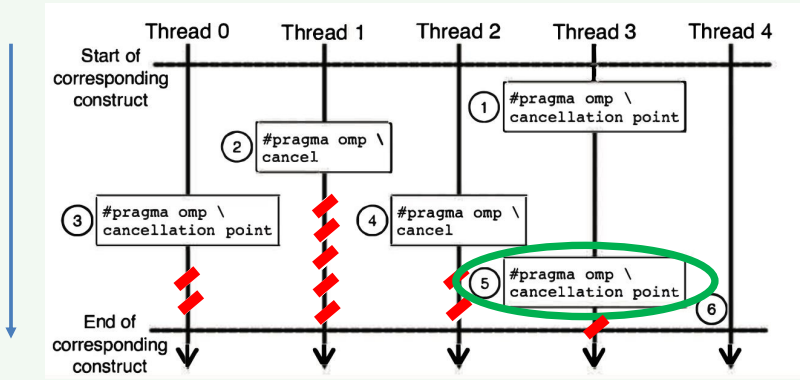
Thread 1 requests the cancellation and waits for a synchronization point, such as a barrier or an cancellation point. From that point on, it is idle and waits.



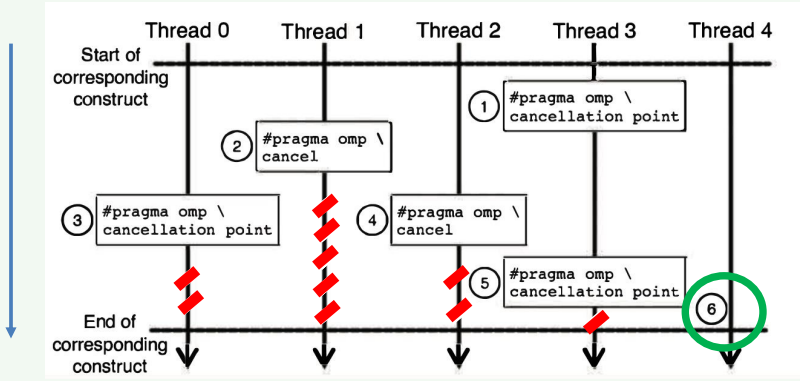
Thread 0 calls a cancellation point to check the flag and terminates because thread 1 (last step) has set the cancellation flag to 1. At this point, thread 1, which was waiting for a synchronization point, also terminates.



At the same time, thread 2 is cancelled because the cancel directive checks the cancellation flag first.



At this point the thread 3 has been terminated.



Finally, thread 4 never encounters a cancellation point and finishes execution normally.

5.9 SIMD Vectorization

A **SIMD processor exploits data parallelism by providing instructions that operate on blocks of data** (called **vectors**). SIMD provides **data parallelism at the instruction level** and can be combined with other OpenMP constructs to achieve multi-level parallelism.

What compilers must do to understand whether a loop can be vectorized

SIMD instructions use **SIMD registers**. The compilers deal with **several issues to determine whether a loop can be vectorized by SIMD instructions**. It does:

- An analysis of the dependencies across iterations;
- An alias analysis of pointers;
- An analysis of the data layout/alignment issues;
- An analysis of conditional executions;
- Checks of the loop bounds that must not be multiple of vector length.

Other compiler problems: Loop Peeling and Loop Tail

Also, the **loop iterations at the beginning and end may not be vectorized** (loop peeling, tail). This is because when a compiler tries to optimize a loop using vectorization (i.e., applying the same operation to multiple data points simultaneously to speed up execution), it often encounters problems with the iterations at the beginning and end of the loop. These iterations may not fit neatly into the vectorized operations because the total number of iterations of the loop may not be a perfect multiple of the vector length.

Essentially, the **compiler may have to split the loop into three parts**:

1. **Loop Peeling**. Handle the initial few iterations that don't align with vector boundaries.
2. **Vectorized Main Loop**. Process the majority of the loop iterations using vectorized operations.
3. **Loop Tail**. Handle the remaining iterations that can't be processed with vector operations due to their small number.

This approach ensures that the loop is as optimized as possible, even if some parts of it cannot be vectorized efficiently.

Use OpenMP SIMD

The `simd` construct can be applied to a loop to indicate that the loop can be transformed into a SIMD loop (i.e., multiple iterations of the loop can be executed simultaneously using SIMD instructions). Doc.

OpenMP: `pragma omp simd`

```
#pragma omp simd
```

The **loop is divided into chunks**, and **all iterations are executed by a single thread using SIMD vector instructions**. The **chunks should fit into a vector register** for **performance**, and **each iteration is executed by a *SIMD lane***. The compiler will generate SIMD instructions, it is **up to the user to ensure this maintains correct behavior**.

Possible clauses

- Data scope clauses (page 78) can be used in a `simd` directive.
- A `collapse` clause can be **used to fuse two perfectly nested loops, but the complexity can be increase**.
- The `simdlen(`*`size`*`)` clause suggests a **preferred vector length**. Maybe the code will work better with a specific vector length, but the compiler is free to ignore it (is only a suggestion make by the programmer). It can also hurt performance but the results remain correct.
- The `safelen(`*`size`*`)` clause sets an **upper limit to the vector length that the compiler cannot exceed**.

For SIMD

The `omp for simd` directive **distributes the iterations of one or more associated loops across the threads** that already exist in the team **and indicates that the iterations executed by each thread can be executed concurrently using SIMD instructions**.

OpenMP: `pragma omp for simd`

```
#pragma omp for simd
```

The **number of threads and scheduling policy greatly affect performance**. If the **number of threads increases, work for each thread is smaller**. **Each thread should work with a chunk corresponding to the vector length**. Ideally, it is correct to distribute iterations among threads in a team, then each thread uses SIMD instructions.

The clause `schedule` **avoid performance degradation specifying the scheduling type and chunk size for the loop iterations**. The `static` schedule divides the iterations into chunks of *`chunk-size`*, distributing them to the threads.

OpenMP: `pragma omp declare simd`

```
#pragma omp for simd schedule(simd:static, chunk-size)
```

Finally, we can also **declare a function to be compiled for calls inside a SIMD loop**.

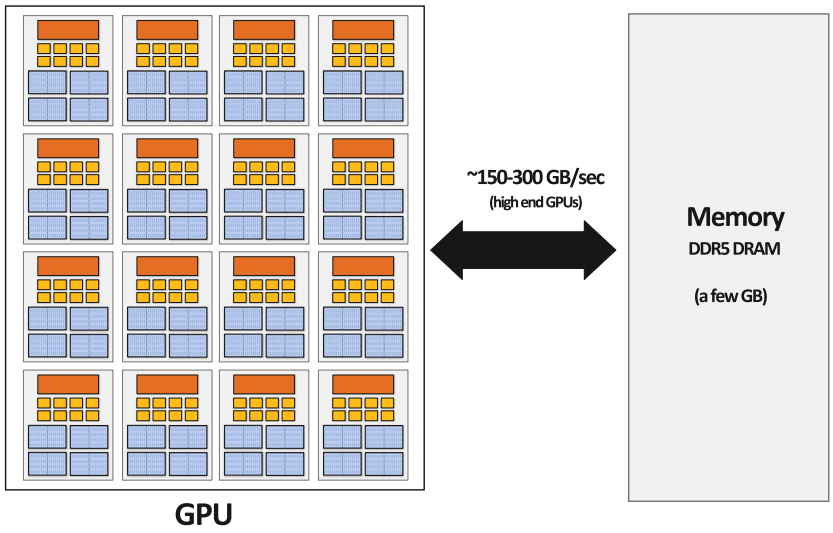
OpenMP: `pragma omp declare simd`

```
#pragma omp declare simd
    /* function definition */
```

6 GPU Architecture

6.1 Introduction

In the image below, we can see a very basic GPU architecture:



- **GPU Structure** (left). The GPU is made up of a **grid of** smaller blocks, each representing a **core**. All the blocks form a **multi-core structure** that allows the GPU to handle many tasks simultaneously.

  Within a **single core**, the GPU uses **SIMD** (Single Instruction, Multiple Data) **execution**. This means that many execution units within a core can simultaneously execute the same instruction on different pieces of data. This results in highly efficient execution of parallel tasks such as rendering graphics or running simulations.

  In addition, **each core supports multi-threaded execution**, which allows multiple threads to be processed simultaneously. This further enhances the GPU's ability to perform multiple tasks simultaneously.

- **Memory Connection** (right). The GPU is connected to DDR5 DRAM, a type of dynamic random access memory. Fast memory is essential to handle the large amounts of data that GPUs process. The more powerful the DRAM, the faster the data transfer.

Initially, GPUs were designed with a specific purpose: to render graphics quickly and efficiently. However, their role has expanded significantly over the years.

General-Purpose computing on Graphics Processing Units (GPGPU) was originally designed to **render graphics**, but GPUs have evolved to perform a wide range of computations beyond traditional graphics tasks. GPGPU takes advantage of the parallel processing capabilities of GPUs to perform computations typically handled by the CPU.

6.2 GPU compute mode

GPU compute mode refers to **GPU hardware that is optimized for general-purpose computing rather than graphics rendering**. This mode allows users to run non-graphics programs on the GPU's programmable cores, taking advantage of the GPU's parallel processing capabilities for tasks such as scientific simulations, data analysis, and machine learning.

Example 1: how to run code on a GPU (prior to 2007)

Now let us see how to run a simple code on a GPU. Suppose a user wants to draw a picture on a GPU:

- **GPU Shader Program Binaries**. The application, via the graphics driver, supplies the GPU with shader program binaries. These are compiled programs that the GPU will execute to perform rendering tasks.
- **Graphics Pipeline Parameters**. The application sets various parameters for the graphics pipeline, such as the output image size, to control how the rendering should be processed.
- **Vertex Buffer**. The application provides the GPU with a buffer of vertices. Vertices are data points that define the shape of the objects to be rendered.
- **Draw Command**. The application sends a draw command to the GPU using the function call `drawPrimitives(vertex_buffer)`. This command instructs the GPU to start rendering using the provided vertex data.

The stages of the graphics pipeline:

1. **Input Vertex Buffer**: The initial stage where the vertex data is input to the pipeline.
2. **Vertex Generation**: Vertices are generated or fetched from the vertex buffer.
3. **Vertex Processing**: The vertices undergo various transformations and shading calculations.
4. **Primitive Generation**: The processed vertices are used to generate geometric primitives (such as triangles).
5. **Fragment Generation (Rasterization)**: The primitives are converted into fragments (potential pixels).
6. **Fragment Processing**: Fragments undergo shading and texturing calculations to determine their final color and properties.
7. **Pixel Operations**: Final operations are performed on the fragments, such as depth testing and blending.
8. **Output Image Buffer**: The processed fragments are written to the output image buffer, resulting in the final rendered image.

Some history

Before 2007 the **only way to interface with GPU hardware was through the graphics pipeline**. This meant that GPUs were ***designed and used specifically for tasks related to graphics rendering***. The pipeline stages (such as vertex processing, fragment generation, and pixel operations) were all designed to transform vertex data into pixels displayed on the screen.

Because they were optimized to handle parallel tasks associated with rendering images, their architecture and interfaces were tightly coupled with graphics APIs.

By 2007, the concept of using GPUs for **General-Purpose computing on Graphics Processing Units (GPGPU)** was emerging. Thanks mainly to the introduction of a new architecture signed NVIDIA called Tesla and CUDA, a parallel computing platform and programming model (also OpenCL was emerging).

The **NVIDIA Tesla architecture**, introduced with the GeForce 8800 GPU in 2006, marked a significant shift in GPU design by unifying graphics and computing capabilities. This architecture featured a scalable parallel array of processors that could be programmed in `C` or via graphics `APIs2`. The Tesla architecture enabled **flexible**, **programmable graphics and high-performance computing**, making it possible to use GPUs for a wide range of applications beyond traditional graphics rendering. This unification enabled massive multithreading and parallel processing, dramatically improving performance for computationally intensive tasks. [7]

From this point on, the programmable cores of the GPU are used:

- Application could allocate buffers in GPU memory and copy data to/from buffers;
- Application (via the graphics driver) provides a single kernel program binary to the GPU;
- Application tells GPU to run kernel in SPMD fashion.

6.3 CUDA

6.3.1 Basics of CUDA

What is CUDA?

Compute Unified Device Architecture (CUDA), is a **parallel computing platform and Application Programming Interface** (API) **model** created by NVIDIA. It allows developers to utilize NVIDIA GPUs for general-purpose processing, enabling them to perform a wide range of computations more efficiently than with traditional CPU processing alone.

CUDA was introduced with the NVIDIA Tesla architecture, which marked a significant shift in GPU capabilities, enabling general purpose computing on GPUs. **CUDA is designed to be similar to the `C` programming language**, making it familiar to many developers. It allows programmers to write code that runs on GPUs using the compute-mode hardware interface.

CUDA's abstractions are relatively low-level and closely match the performance characteristics and capabilities of modern GPUs. This design goal helps maintain a low abstraction layer, ensuring that developers can take full advantage of the hardware's potential.

Note that **Open Computing Language (OpenCL)** is an open standards version of CUDA that runs on both CPUs and GPUs from multiple vendors. **While CUDA runs only on NVIDIA GPUs, OpenCL is designed to be more versatile and work on hardware from multiple vendors**.

CUDA Thread Hierarchy

CUDA organizes threads into a hierarchical structure to efficiently manage parallel computations on GPUs. To understand how it works, let's look at it from the deepest level up:

1. **Threads**. Threads (or **CUDA Threads**) are the **smallest unit of execution** in CUDA. Each thread runs a single instance of a kernel function[8].

   Threads are **identified by their unique thread IDs**, which are used to calculate memory addresses and control decisions.

2. **Thread Blocks**. Threads are grouped into blocks (called also **CUDA Blocks**). Each block can contain multiple threads that execute concurrently.

   Threads within the same block can communicate and share data through shared memory and synchronization primitives like barriers and atomic operations.

   The *maximum number* of threads per block is limited by the GPU architecture, typically up to 1024 threads per block.

[8] A kernel function in CUDA is a function that runs on the GPU (device) but is called from the CPU (host). These functions are executed by many parallel threads on the GPU

3. **Grids**. Blocks are organized into a grid (called also **CUDA Grids**). A grid is a **collection of blocks that execute the same kernel function**.

   All **threads in a grid share the same global memory space**.

   The grid can be multi-dimensional, allowing for flexible organization of blocks to match the problem's dimensions.

Note that CUDA uses the `x`, `y`, and `z` dimensions for threads, blocks, and grids because this **multi-dimensional structure** aligns well with many common computational problems. Many computational problems naturally fit into a multi-dimensional space. For example, image processing involves 2D data, and volumetric simulations involve 3D data.

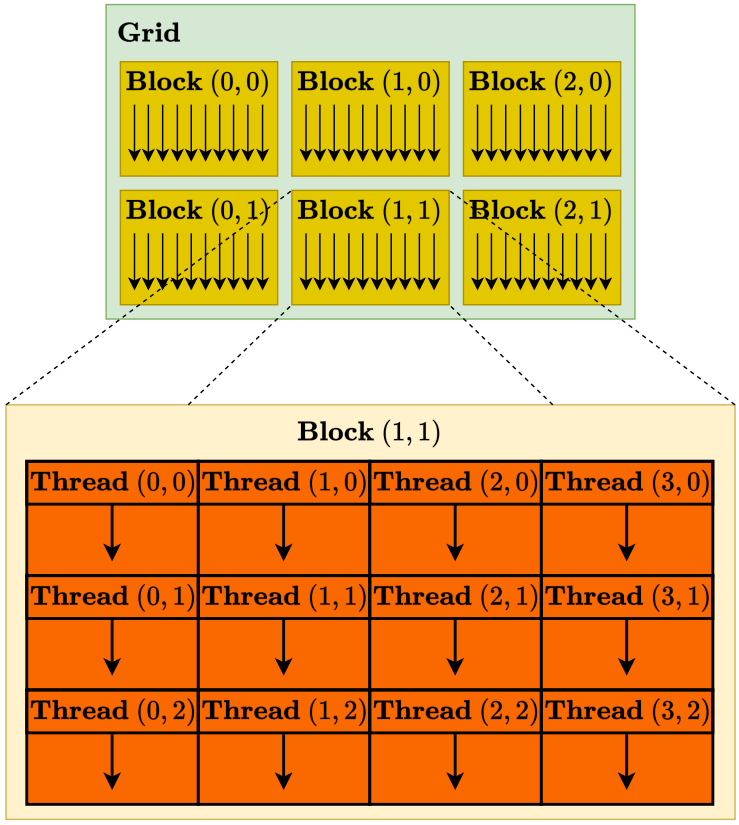


Figure 22: The figure shows an example of a CUDA grid consisting of 6 blocks, each block consisting of 12 threads. The number of threads and the number of blocks per column/row are customizable, this is just an example. In total, there are 72 CUDA threads (12 threads per block times 6 blocks in the grid).

CUDA Kernels

A **CUDA kernel** is a **function that gets executed on the GPU**. It contains the parallel portion of the application, which is executed by multiple threads in parallel. Unlike regular C/C++ functions that run on a single thread, CUDA kernels can run thousands of threads simultaneously.

⚖ Device vs Host

In CUDA, there is a term to identify the code that runs on the CPU and GPU.

Definition 1: CUDA Host (CPU)

The **CUDA Host** **refers to the CPU** and its associated memory.
It is **responsible for managing the overall program execution**. This includes allocating memory, launching CUDA kernels, and transferring data between the CPU and the GPU.
It is typically written in the standard programming language used (e.g., C, C++, Java, Python, etc.) and contains the logic for setting up and controlling the execution of CUDA kernels on the device.

Definition 2: CUDA Device (GPU)

The **CUDA Device** **refers to the GPU** and its associated memory.
It is **responsible for running parallel code** (CUDA kernels) and **performs the bulk of the computation**. This takes advantage of the parallel processing power of the GPU.
The device code (kernels) is written in the CUDA programming language and is designed to run on the multiple parallel cores of the GPU.

</> Basic CUDA syntax

Here is a sample code written in CUDA:

```
#include <cuda_runtime.h>
#include <iostream>
#define Nx 12
#define Ny 6

// Kernel definition
__global__ void MatAdd(
    float A[Ny][Nx],
    float B[Ny][Nx],
    float C[Ny][Nx]
) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    // guard against out of bounds array access
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main() {
    // Define the hierarchy
    dim3 threadsPerBlock(4, 3);
    dim3 numBlocks(
        Nx / threadsPerBlock.x,
        Ny / threadsPerBlock.y
    );
    // Kernel invocation
    // Assume matrices A, B, C are already allocated (dim Nx x Ny)
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```

Rows 3-4 Variables `Nx` and `Ny` define the dimensions of the matrices. In this case, they are 12×6.

Row 21 The `threadsPerBlock` function specifies that each block will have 4 threads in the x dimension and 3 threads in the y dimension. In other words, it defines how a block should be composed (see the figure 22).

Row 22 The `numBlocks` function defines the number of blocks required to cover the entire matrix. The number of blocks is calculated by dividing the matrix dimensions by the number of threads per block in each dimension. In other words, it defines how a grid should be composed (see the figure 22).

Row 28 The execution configuration syntax `<<<...>>>` is required to specify the number of threads per block and the number of blocks per grid (both numbers can be of type `int` or `dim3`).

The function invocation starts the `MatAdd` kernel with the given grid and block dimensions. This kernel runs on the GPU and performs the matrix addition.

Row 7 Defines a kernel function called `MatAdd` to be executed on the GPU. The `__global__` keyword indicates that this is a kernel function.

Rows 12-13 Computes the global column index i for each thread. This combines the block index and the thread index within the block to get the overall position.

It also does the same with the j index.

Row 15 The `if` condition ensures that the thread does not access out-of-bounds elements in the matrices.

Row 16 Finally, it performs the element-wise addition of matrices `A` and `B` and stores the result in matrix `C`.

As we said in Figure 22 (page 103), the example spawns 72 CUDA threads.

6.3.2 Memory model

The CUDA memory model consists of several types of memory, each with different characteristics and uses. In general, the **Host (CPU) and the Device (GPU) have different address spaces**, each one has its private memory address.

For example, the `cudaMemcpy` function in CUDA is used to **copy data** between different memory spaces, specifically **between host (CPU)** memory **and device (GPU) memory**.

```
// allocate buffer in host mem
float* A = new float[N];

// populate host address space pointer A
for (int i = 0; i < N; ++i) {
    A[i] = (float)i;
}

// allocate buffer in device (GPU) address space
int bytes = sizeof(float) * N;
float* deviceA;
cudaMalloc(&deviceA, bytes);

// populate deviceA
cudaMemcpy(deviceA, A, bytes, cudaMemcpyHostToDevice);
// deviceA:
//      Destination memory address (either on the host or device).
//
// A:
//      Source memory address (either on the host or device).
//
// bytes:
//      Number of bytes to copy.
//
// cudaMemcpyHostToDevice:
//      Type of memory copy operation.
//      Copies data from host memory to device memory.
```

Note that directly accessing `deviceA[i]` is an invalid operation, because we cannot manipulate the contents of `deviceA` directly from host, since `deviceA` is not a pointer to the host's address space.

Types of CUDA device memory models visible to kernels

1. **Per-thread Private Memory**. This memory is **private to each thread**. Each thread has its own memory space that other threads cannot access.

   Usage. It is **ideal for storing variables that are only relevant to individual threads** and do not need to be shared with other threads.

   Access. It has **fast access**, limited by the number of registers available.

2. **Per-block Shared Memory**. This memory is **shared by all threads within a block**. It allows threads within the same block to cooperate by sharing data.

   Usage. It is useful for tasks where **threads within a block need to communicate or share intermediate results**. It is often used to optimize memory access patterns and reduce global memory accesses.

   Access. It is **much faster than global memory, but limited in size**. Access is almost as fast as registers when used properly.

3. **Device Global Memory**. This memory is **accessible to all threads across all blocks**. It provides a large amount of memory, but has higher latency and lower bandwidth than shared memory.

   Usage. It is **suitable** for storing large amounts of **data that must be accessed by threads in different blocks**.

   Access. It has the **slowest access** of the three types, but is necessary for large data storage and inter-block communication.

6.3.3 NVIDIA V100 Streaming Multiprocessor (SM)

The NVIDIA V100 is a powerful GPU designed for data centers, primarily used for Artificial Intelligence (AI), High Performance Computing (HPC), and data science. Meanwhile, the **NVIDIA V100 Streaming Multiprocessor (SM)** is a key component of the V100 GPU architecture.

How is architecture composed?

See Figure 23 (page 109) for a graphical representation.

- **Warp Selector and Fetch/Decode**:

  The *Warp Selector* and *Fetch/Decode* units are responsible for **managing the execution of warps** (groups of threads) and **decoding instructions**.

- **Functional Units**:

  - **SIMD fp32 functional unit** (Yellow).
    It **handles single-precision floating-point operations**. Control is shared across 16 units, allowing for 16 multiply-add (`MUL-ADD`) operations per clock cycle. This translates to one 32-wide SIMD operation every two clocks.
  - **SIMD int functional unit** (Orange).
    It **manages integer operations**, also shared across 16 units, with the same performance characteristics as the fp32 unit (16 × `MUL-ADD` per clock).
  - **SIMD fp64 functional unit** (Brown).
    It is **responsible for double-precision floating-point operations**. Control is shared across 8 units, allowing for 8 `MUL-ADD` operations per clock cycle, equating to one 32-wide SIMD operation every four clocks.
  - **Tensor core unit** (Red).
    It is **specialized for tensor operations**, which are crucial for deep learning and AI workloads.
  - **Load/store unit** (Green).
    It **handles memory operations**, such as loading data from and storing data to memory.

- **Warp Scheduler**:

  The diagram below shows the scheduling of warps (Warp 0, Warp 4, Warp 60, etc.) across the functional units, indicating how different warps are processed in parallel.

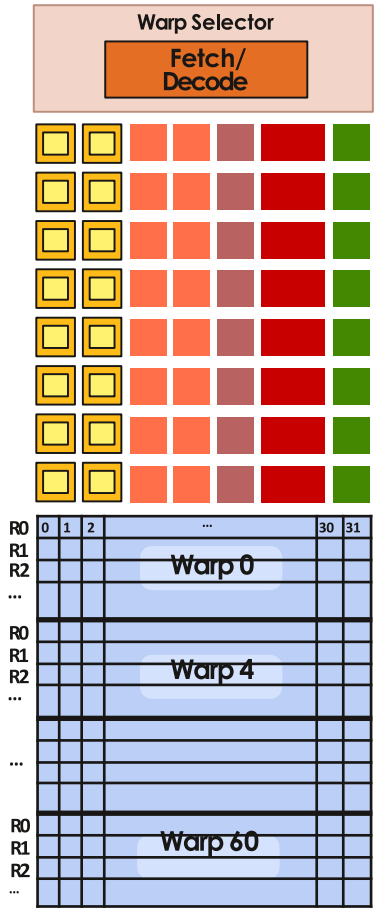


Figure 23: A "sub-core" of the NVIDIA V100 Streaming Multiprocessor (SM) architecture.

What is a Warp?

A **Warp** is a **group of 32 threads that execute the same instruction at the same time**. **Threads within a block are divided into warps**. For example, a block of 256 threads would have 8 warps (256 threads / 32 threads per warp). **Each SM** in the V100 **can schedule and interleave the execution of up to 16 warps**. This means that multiple warps can be executed simultaneously, improving the overall throughput of the GPU. Finally, each warp has some registers to store the data needed by the threads.

How is a Warp executed?

Threads within a warp execute the same instruction simultaneously, taking advantage of **SIMD execution** to improve performance.

When threads within a warp do not share the same instruction, it results in divergent execution, which can degrade performance due to the need to serialize instructions. However, the **check** of the same instruction by the 32 threads is done **dynamically by the GPU hardware**. Finally, although not part of CUDA, understanding warps is critical to optimizing CUDA programs on modern NVIDIA GPUs.

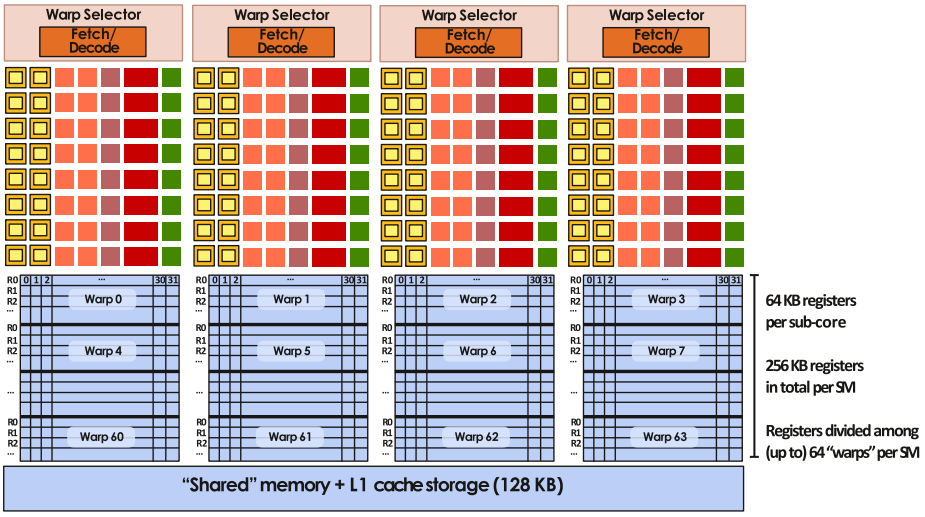


Figure 24: A NVIDIA V100 Streaming Multiprocessor (SM) architecture.

6.3.4 Running a CUDA program on a GPU

The following section is dedicated to understanding the execution of the "convolve" kernel on a fictitious dual-core GPU. In other words, we present an example of a kernel function execution (and therefore GPU-executed) that implements a convolution operation. It also aims to show why memory management is also fundamental inside the GPU.

The kernel execution requirements for this explanation are:

- Each **thread block** must execute **128 CUDA threads**.
- Each thread block must allocate $130 \times$ `sizeof(float)` $= 520$ bytes of shared memory. In other words, **each thread block requires 520 bytes of shared memory**.

Let's take an array of size N as **input** to the kernel function. When the kernel function is executed (launched), it generates thousands of thread blocks because the array size N is assumed to be very large.

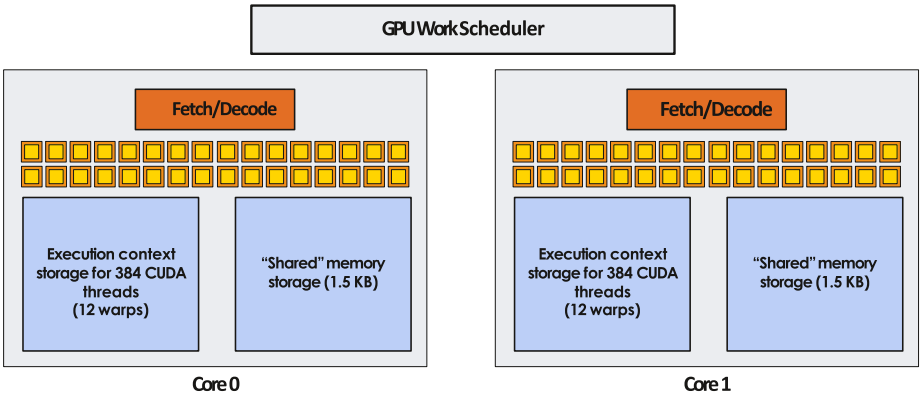
```
1 #define THREADS_PER_BLK 128
2 convolve<<<(N/THREADS_PER_BLK), THREADS_PER_BLK>>>(N, input_array,
      output_array);
```

Where:

- `N/THREADS_PER_BLK` is the number of blocks per grid.
- `THREADS_PER_BLK` is the number of threads per block, 128 CUDA threads in our case.

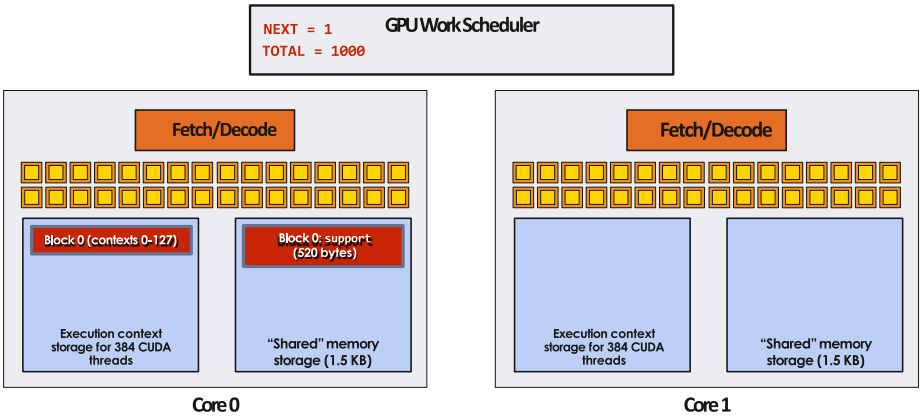
The **main task** of the *GPU Work Scheduler* is to **manage the two available cores** (let's say Core 0 and Core 1), where each core has its own **Fetch/Decode units**, **execution context storage** for 384 CUDA threads (12 warps), and **shared memory storage** (1.5 KB).

Note that this architecture has fictitious cores that are smaller than V100 SM cores, with fewer execution units, less support for active warps, and less shared memory.

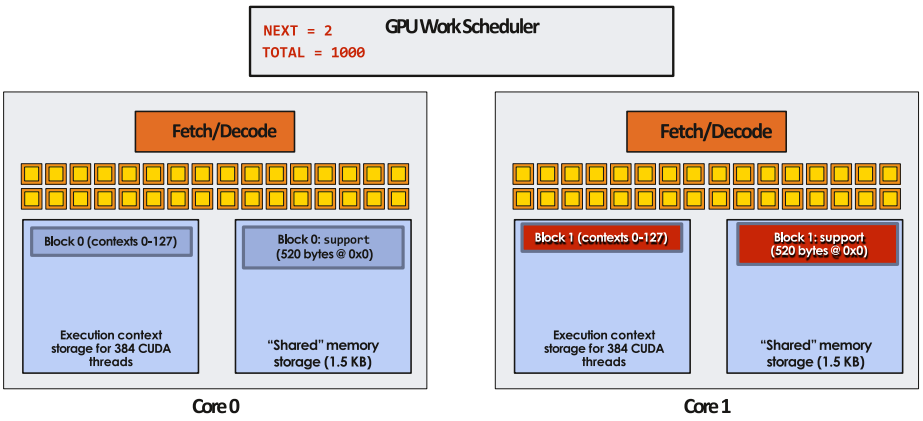


Execution

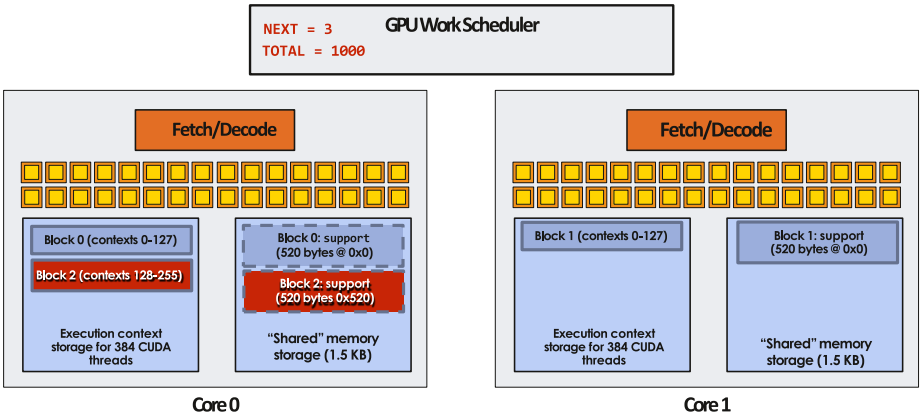
1. The **host** (CPU) **sends a command to the CUDA device** (GPU), which is the execution of the kernel function.
   - **Execute**: convolve (kernel function)
   - **Args**:
     - `N` $= 128'000$
     - `input_array`
     - `output_array`
   - **Number of blocks**: 1000. Note that the number of blocks is given by the formula:
     $$\frac{\texttt{N}}{\texttt{THREADS\_PER\_BLK}}$$
     Therefore, the size of the array is easily calculated as:
     $$\frac{\texttt{N}}{128} = 1000 \Rightarrow \texttt{N} = 128 \times 1000 = 128'000$$
2. The *scheduler* maps *block 0* to *Core 0*, reserving execution contexts for 128 threads and 520 bytes of shared memory to meet requirements.



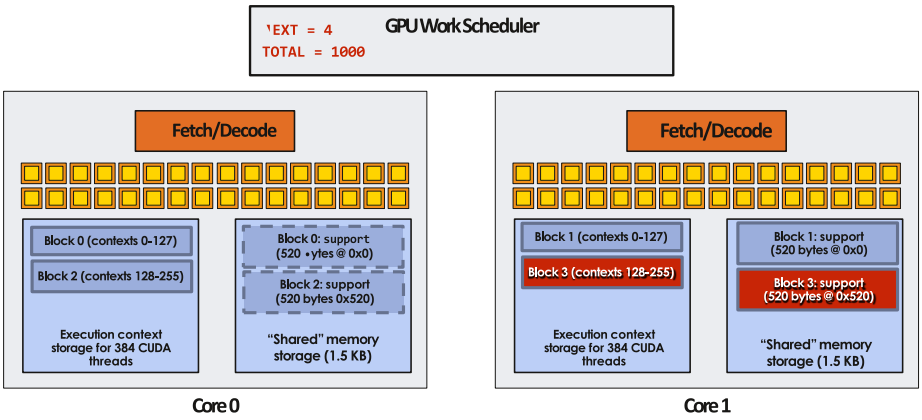
3. The *scheduler* continues to map blocks to available execution contexts, so it now maps *block 1* to *Core 1*. This shows **interleaved mapping** (where the scheduler maps blocks to available execution contexts across different cores, ensuring efficient use of resources).



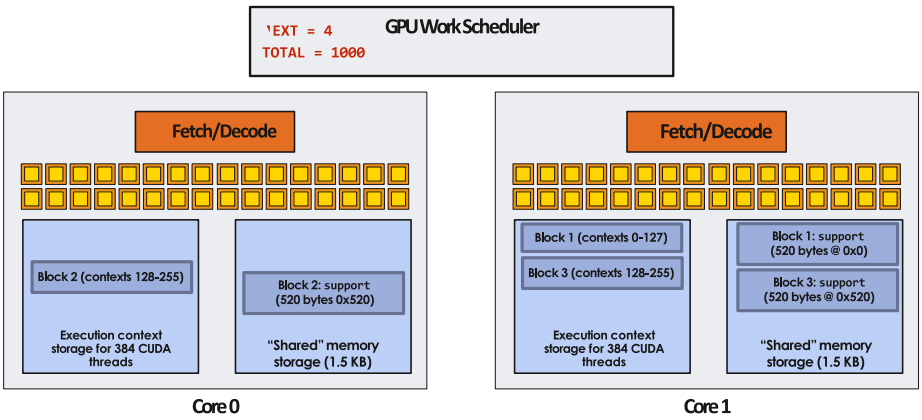
4. As in the previous step, there is a interleaved mapping phenomena. The scheduler maps *block 2* to *Core 0*. But now the shared memory (*Core 0*) is saturated because three concurrent blocks allocate 520 `bytes` $\times 3 =$ 1.56 `KB` $>$ limit (1.5 `KB`).



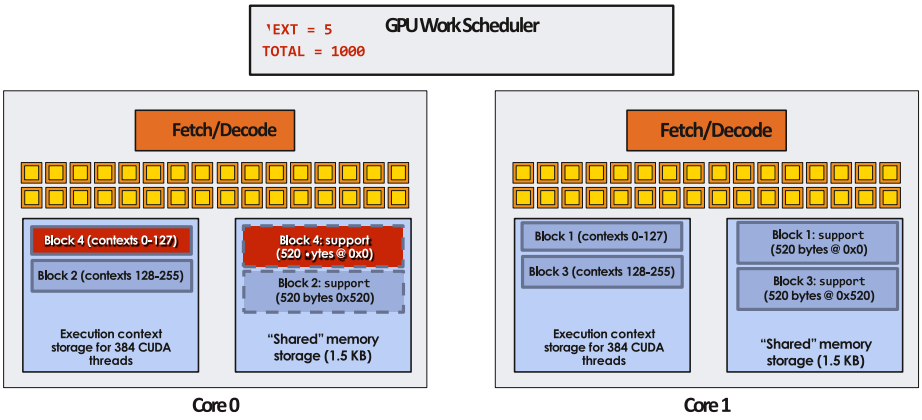
5. The scheduler assigns *block 3* to *Core 1*. But now the shared memory of *Core 1* is also **saturated**, because three concurrent blocks allocate 520 `bytes` $\times$ 3 = 1.5 `KB` $>$ limit (1.5 `KB`). So we can say that only two thread blocks fit on one core.



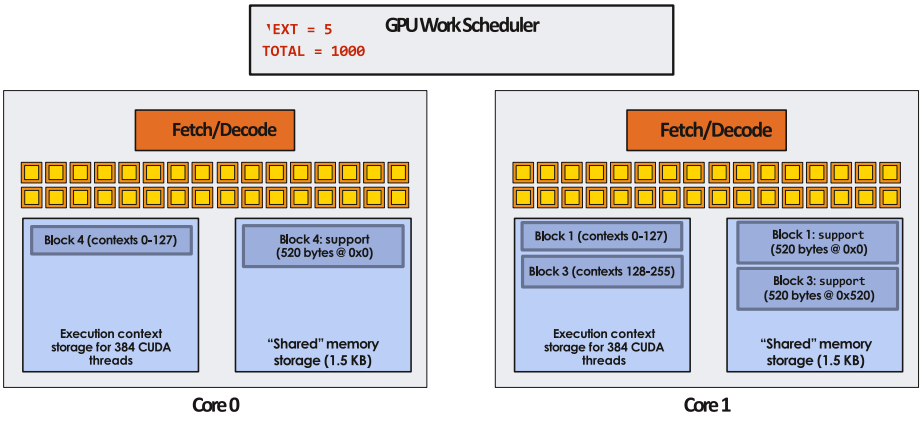
6. The scheduler waits for a task to complete on a block. The following figure shows *block 0* completing on *Core 0*. Now *Core 0* is ready to host execution blocks again.



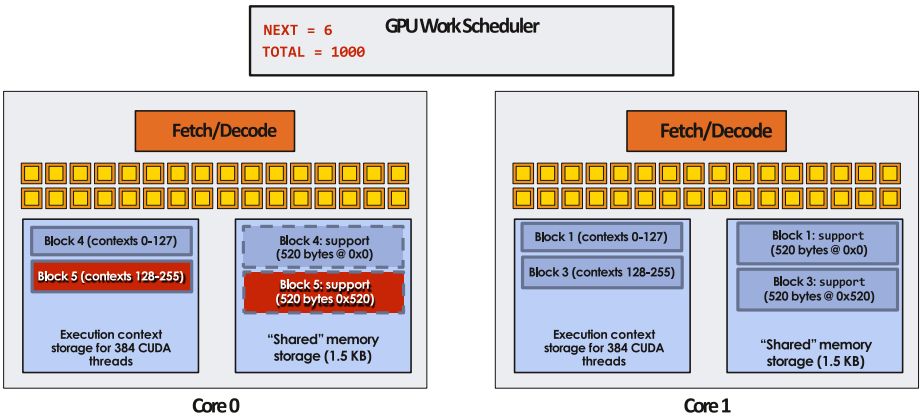
7. When the task is complete, the scheduler assigns *block 4* to *Core 0*.



8. Thread *block 2* completes on *Core 0*.



9. Finally, thread *block 5* is scheduled on *Core 0*.



The explanation is intended to illustrate a phenomenon where the GPU scheduler has to manage limited shared memory resources across multiple thread blocks.

- **Shared Memory Saturation**: When a GPU core's shared memory is fully occupied by existing thread blocks, new thread blocks cannot be scheduled until sufficient shared memory is freed up by the completion of some of the current thread blocks.
- **Idle Periods**: While the scheduler is waiting for shared memory to become available, the **GPU cores may be idle**. This doesn't mean the entire GPU is idle, but certain cores may not have new blocks to execute until resources are freed up.
- **Resource Contention**: This example shows how shared memory contention can affect the scheduling efficiency of a GPU. **Efficient use of shared memory is critical to maximizing GPU performance**.
- The concept of **resource contention**, whether it be shared memory, registers or other resources, is well known in parallel computing and GPU programming. It highlights the **importance of optimizing memory usage to avoid bottlenecks and ensure efficient execution**.

The example demonstrates how GPUs must juggle limited resources while maximizing throughput, a key aspect of parallel computing. By showing that the scheduler must wait before allocating new blocks, it emphasizes the **importance of careful resource management in kernel design and execution**.

6.3.5 Implementation of CUDA abstractions

We assume that we have a fictitious **Streaming Multiprocessor (SM) core** (Figure 23 page 109, same as NVIDIA V100, page 108) with only **four warps** of parallel execution in hardware. So there are 4 warps times 32 threads, and **128 threads** can be executed in parallel each time.

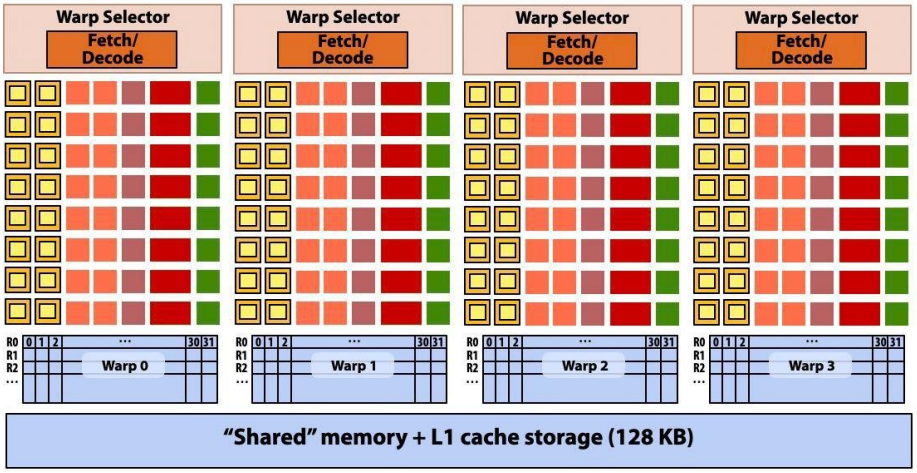


Figure 25: Streaming Multiprocessor (SM) core, with 4 warps and 128 threads in total.

Why allocating all threads in a block might be inefficient

Now imagine we want to run a CUDA program where a thread **block** consists of **256 CUDA threads**, even though our GPU architecture can *only* execute 128 threads at a time. A naive implementation might execute the entire CUDA block by executing four warps (threads 0-127) to completion, and then execute the next four warps (threads 128-255) to completion. This **sequential execution of warps can lead to inefficiencies** because **there may be idle periods where some warps are stalled waiting for memory accesses or synchronization points**.

Use interleaving execution

A good alternative is to use interleaved execution. **Interleaved execution** means that the **GPU schedules warps so that they overlap**. While some warps are waiting for memory access or synchronization, **others can execute**. This overlapping helps to **hide latencies** and **keep the GPU cores busy**.

However, **CUDA kernels can create dependencies between threads in a block**. To manage these dependencies, the programmer can use the function `__syncthreads()` to synchronize threads within a block. `__syncthreads()` ensures that **all threads in the block reach the synchronization point before any thread can continue**. This means that threads 128-255 cannot continue until threads 0-127 reach the synchronization point.

Why interleaving execution is optimal when there are dependencies

If we run four warps to completion before starting the next four, we **may not be using shared resources** (such as shared memory and execution units) efficiently. This can also **lead to scenarios where some warps are stalled indefinitely waiting for other warps to reach synchronization points**, causing potential **deadlocks** or inefficiencies.

By interleaving the execution of warps, the GPU can better manage thread dependencies, hide latencies, and make more efficient use of its resources.

```
#define THREADS_PER_BLK 256

__global__ void convolve(int N, float* input, float* output)
{
    __shared__ float support[THREADS_PER_BLK * 2];
    int index = blockIdx.x * blockDim.x + threadIdx.x;

    support[threadIdx.x] = input[index];
    if (threadIdx.x < N) {
        support[
            THREADS_PER_BLK + threadIdx.x
        ] = input[index + THREADS_PER_BLK];
    }

    __syncthreads();

    float result = 0.0f; // thread-local
    for (int i = 0; i < 5; i++)
        result += support[threadIdx.x + i];

    output[index] = result;
}
```

Summary

1. Thread **blocks** can be **scheduled in any order by the system**.
   - The system assumes no dependencies between blocks.
   - Blocks are logically concurrent, similar to ISPC tasks (Implicit SPMD Program Compiler, page 30).
2. **CUDA threads in the same block run concurrently** (live at the same time).
   - When a block begins executing, all threads exist and have register state allocated.
   - A **CUDA thread block is an SPMD** (Single Program, Multiple Data) **program**, similar to an ISPC gang of program instances.
   - Threads in a thread block are concurrent, cooperating "workers".

3. **CUDA implementation**.

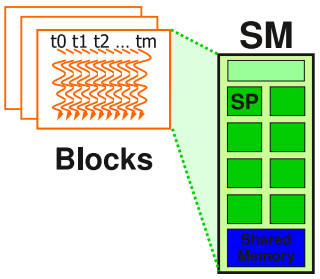
   - An NVIDIA GPU warp has performance characteristics *similar* to an ISPC gang of instances.
   - All warps in a thread block are scheduled onto the same SM (Streaming Multiprocessor), allowing for high-bandwidth, low-latency communication through shared memory variables.
   - When all threads in a block complete, block resources (shared memory allocations, warp execution contexts) become available for the next block.

6.3.6 Advanced thread scheduling

The *main goal* of this section is to show how a CUDA kernel uses hardware execution resources: thread block allocation to execution resources, execution resource capacity constraints, and zero-overhead thread scheduling.

In general, **CUDA thread blocks execute independently and can run in any order**. The hardware is **free to assign blocks to any processor at any time**. This flexibility allows the GPU to optimize resource utilization and balance the load. A kernel (the function that runs on the GPU) scales to any number of parallel processors. This means that the same code can run efficiently on GPUs with different numbers of cores.

Thread blocks are the basic unit of work in CUDA and are assigned to SMs in block granularity (as we saw in Chapter 6.3.4). This means that a **thread block cannot be split across multiple SMs, but is executed entirely within a single SM**. Each SM has a *limit* on the number of threads and thread blocks it can support simultaneously. For example, the **Volta SM can handle up to 2048 threads**. The number of blocks an SM can hold depends on the number of threads per block:



- If a block has 256 threads, up to 8 blocks can fit ($256 \times 8 = 2048$ threads).
- If a block has 512 threads, only 4 blocks can fit ($512 \times 4 = 2048$ threads).

The **SM manages the indexes of the threads and blocks assigned to it**, enabling scheduling and execution.

Von Neumann model with SIMD units

The **Von Neumann Model** consists of a Control Unit, ALU (Arithmetic Logic Unit), Registers, Memory, and I/O components that work in a sequential manner. However, when we **integrate SIMD** (Single Instruction, Multiple Data) units into the model, we add the **ability to process multiple data items simultaneously using a single instruction**.

As we have explained in the section 2.1.3 page 24, SIMD allows the same operation to be performed on multiple pieces of data in parallel. This means a Control Unit (CU) sends the same instruction to multiple ALUs, each working on different data at the same time.

Von Neumann model with SIMD units in GPUs

The **architecture uses SIMD units to execute multiple threads in parallel**, making GPUs highly efficient at tasks involving large data sets, such as image processing, matrix multiplication, and other data-parallel computations.

Warps (groups of 32 threads) are **executed in a SIMD fashion**. **All threads in a warp perform the same operation, but on different pieces of data**. This is critical for speeding up computations that need to process large amounts of data in parallel. SIMD capabilities allow GPUs to efficiently handle large numbers of parallel tasks, making them far more powerful than traditional CPUs for certain workloads, such as graphics rendering and scientific computing.

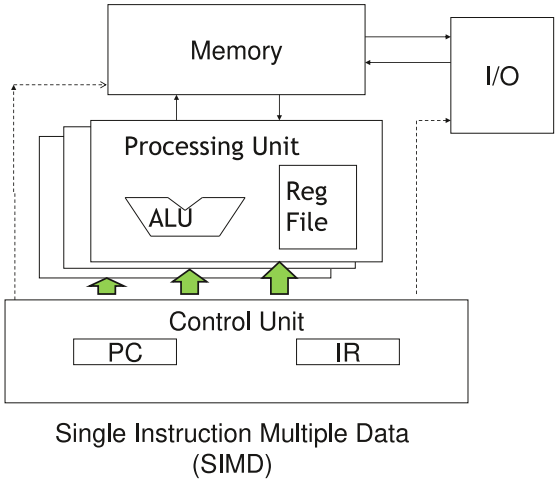


Figure 26: Von Neumann model with SIMD units.

However, these features are implementation choices, not part of the CUDA programming model. Therefore, future GPUs may have different numbers of threads in each warp.

Example 2: Warp

If 3 blocks are assigned to a Streaming Multiprocessor (SM) and each block has 256 threads, how many warps are there in an SM?
Since each warp is made up of 32 threads, if a block is made up of 256 threads, then 8 warps are required for each block:

$$\frac{\text{Threads per block}}{\text{Threads per warp}} = \frac{256}{32} = 8$$

Since the number of blocks to be allocated is 3, if each block requires 8 warps, there are $8 \times 3 = 24$ warps inside a single SM.

Zero-Overhead Warp Scheduling [11]

Zero-Overhead Warp Scheduling is a feature of NVIDIA's GPUs that allows **efficient management of thread execution** without significant performance penalties. This mechanism is obviously **implemented in the Streaming Multiprocessor architectures**.

- **Warp Status**. Each warp has a state that can be:
  - **Eligible**: the warp is ready to execute its next instruction because all the *necessary operands are available* (ready for consumption).
  - **Not Eligible**: the warp cannot execute its next instruction because it is *waiting for operands or other resources*.
  - **Busy**: the warp is *currently running and executing instructions*.

  Each warp also has an associated **priority value**.
- The scheduler has a **Prioritized Scheduling Policy**, so it selects:
  1. **Highest Priority and Eligible**: the scheduler first selects warps that have the highest priority and are eligible for execution.
  2. **Eligible Warps**: if no high-priority warps are eligible, the scheduler then selects from the pool of eligible warps.
- **Execution Context Switching**. After that the scheduler choose a warp to run, *how it can change the execution context from a warp to another?*
  - Before giving the answer, it is necessary to understand that the **Execution Context contains the state of all threads in a warp**, such as Program Counters, register values, and other information necessary for execution. This context is **stored in hardware resources dedicated to each warp**.
  - Thus zero-overhead warp scheduling takes advantage of the power of the hardware. When the warp **scheduler decides to switch** from one warp to another (to hide latency or because a warp is waiting for data), it does not need to save and restore context to and from memory. Instead, **it simply switches the execution state to the context of another warp, which is already stored in dedicated hardware**.

  This means that the switching process is very fast and has virtually no overhead, hence the term "zero-overhead"!

  In other words, **since all the necessary state information is already stored in the hardware, this switch is instantaneous and doesn't involve the overhead of saving/loading data to/from memory**.

Advantages

- **Efficiency**. By leveraging hardware resources for context storage, CUDA ensures that switching between warps incurs virtually no overhead.
- **Latency Hiding**. This efficient scheduling helps hide latencies and keeps the GPU's computational resources fully utilized, leading to high performance.

Example 3: Matrix Multiplication on Volta Architecture

This example illustrates the impact of different block sizes on thread utilization when performing matrix multiplication using NVIDIA's Volta GPU architecture.
With the term **Block Granularity** we refer to the **number of threads per block**. **Choosing the right block size is crucial for maximizing the efficiency of GPU resources**.

- In general, each Streaming Multiprocessor (SM) on the Volta architecture can handle up to 2048 threads.
- 4×4 Threads per Block:
  - **Threads per Block**: 16 threads (4×4).
  - **Blocks per SM**: the GPU can accommodate up to 32 blocks per SM.
  - **Utilization**: 16 threads per block mean 2048 threads would fill 128 blocks. However, since an SM can only accommodate up to 32 blocks at a time, only 512 threads will be utilized per SM (16 threads/block $\times$ 32 blocks), leading to under-utilization of available threads!
- 8×8 Threads per Block:
  - **Threads per Block**: 64 threads (8×8).
  - **Blocks per SM**: with 2048 threads per SM, up to 32 blocks of 64 threads each can be allocated per SM (2048 threads per SM $\div$ 64 threads per block).
  - **Utilization**: this setup can utilize the full capacity of 2048 threads per SM, provided other resource limitations (such as shared memory or registers) are not a constraint.
- 30×30 Threads per Block:
  - **Threads per Block**: 900 threads (30×30).
  - **Blocks per SM**: with 2048 threads per SM, only two blocks of 900 threads each can be accommodated, resulting in 1800 threads (2×900), which is less than the SM's maximum capacity.
  - **Utilization**: this configuration also leads to under-utilization because it does not utilize the full 2048 thread capacity.

The example on page 123 shows some important points to emphasize:

- **Choosing Block Size**: choosing the optimal block size is critical to maximizing GPU efficiency. In the previous example, a block size of 8×8 allows full utilization of the SM's thread capacity.
- **Resource Constraints**: when choosing a block size, we must also consider other resource limitations, such as shared memory and registers, which can affect the number of threads and blocks that can be scheduled on an SM.

6.3.7 Memory and Data Locality in Depth

In CUDA programming, understanding memory and data locality is crucial for achieving high performance.

Why Memory Hierarchy and Data Locality Matter

We propose a piece of code of a kernel function where there is a computation of the blur in an image:

```
1  // get the average of the surrounding 2xBLUR_SIZE x 2xBLUR_SIZE box
2  for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow)
3  {
4      for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol)
5      {
6          int curRow = Row + blurRow;
7          int curCol = Col + blurCol;
8          // Verify we have a valid image pixel
9          if(curRow > -1 && curRow < h && curCol > -1 && curCol < w)
10         {
11             pixVal += in[curRow * w + curCol];
12             // keep track of number of pixels
13             // in the accumulated total
14             pixels++;
15         }
16     }
17 }
18
19 // write our new pixel value out
20 out[Row * w + Col] = (unsigned char)(pixVal / pixels);
```

The **code accesses global memory for input matrix elements**. This is evident from the line 11 where `in` is likely a pointer to the global memory holding the image data. Therefore, each thread accesses global memory to read the pixel values within a 2xBLUR_SIZE x 2xBLUR_SIZE box around the current pixel.

- Each **memory access is 4 bytes** per floating point addition.
- The **memory bandwidth is 4 bytes per second per FLOP**, 4B/s of memory bandwidth/FLOPS.

Assuming a GPU with a peak floating point rate of 1,600 GigaFLOPS and a DRAM bandwidth of 600 GB/s, then 1,600 GigaFLOPS would require 6,400 GB/s of memory bandwidth to achieve the peak FLOPS. However, the 600 GB/s memory bandwidth limits execution to 150 GFLOPS, which is only 9.3% of the peak floating point execution rate. **To get close to the 1,600 GigaFLOPS, it is necessary to drastically cut down memory accesses**.

In other words, there is an **evident Memory Bandwidth Bottleneck**. Even with a high peak floating-point rate, the actual performance can be limited by memory bandwidth (to achieve 1,600 GigaFLOPS, the kernel would require 6,400 GB/s of memory bandwidth, but the available bandwidth is only 600 GB/s).

Therefore, it is important to understand that **minimizing global memory access and optimizing data locality are essential strategies in CUDA programming to achieve high performance**. This is because accessing global memory often results in high latency, which can significantly degrade the performance of our GPU application.

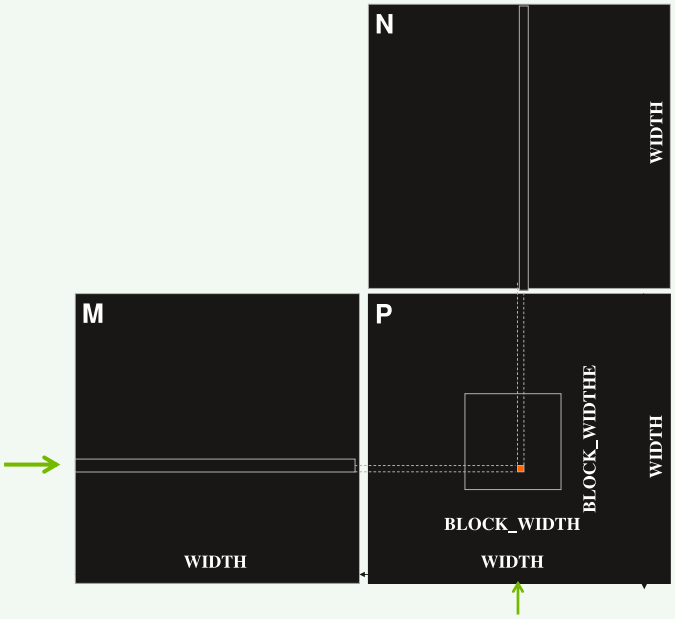
- **Global memory access**. Global memory is the largest and slowest memory available on a GPU. **Frequent accesses to global memory can cause significant performance bottlenecks due to high latency**.
- **Bandwidth and Latency**. While global memory provides high bandwidth, the latency associated with **accessing it can slow overall performance**, especially when multiple threads are accessing it simultaneously.
- **Memory hierarchy**. GPUs have a hierarchical memory structure that includes registers, shared memory, and global memory. **Using a proper memory structure** can improve performance and reduce memory bandwidth. We discussed these topics in Section 6.3.2, page 107.
- **Data Locality and Caching**. Optimizing data locality (keeping frequently accessed data close to where it is processed) is key to improving performance. **Using shared memory to cache data that is accessed multiple times by threads can significantly reduce the need for global memory accesses**.

Example 4: Matrix Multiplication

Matrices:

- Matrix M
- Matrix N
- Matrix P, the resulting matrix.

In the image below, the matrices are divided into blocks with dimensions labeled as `WIDTH` and `BLOCK_WIDTH`, indicating the block-based approach to matrix multiplication.



Arrows show the direction of data flow, indicating how elements from matrices M and N are multiplied to form matrix P.
The code used to perform the matrix multiplication is as follows:

```
__global__ void MatrixMulKernel(
    float* M, float* N, float* P, int Width
) {
    // Calculate the row index of the P element and M
    int Row = blockIdx.y * blockDim.y + threadIdx.y;

    // Calculate the column index of P and N
    int Col = blockIdx.x * blockDim.x + threadIdx.x;

    int RowTimesWidth = Row * Width;

    if ((Row < Width) && (Col < Width)) {
        float Pvalue = 0;
        // Each thread computes one element
        // of the block sub-matrix
        for (int k = 0; k < Width; ++k) {
            Pvalue += M[RowTimesWidth + k] *
                      N[k * Width + Col];
        }
        P[RowTimesWidth + Col] = Pvalue;
    }
}
```

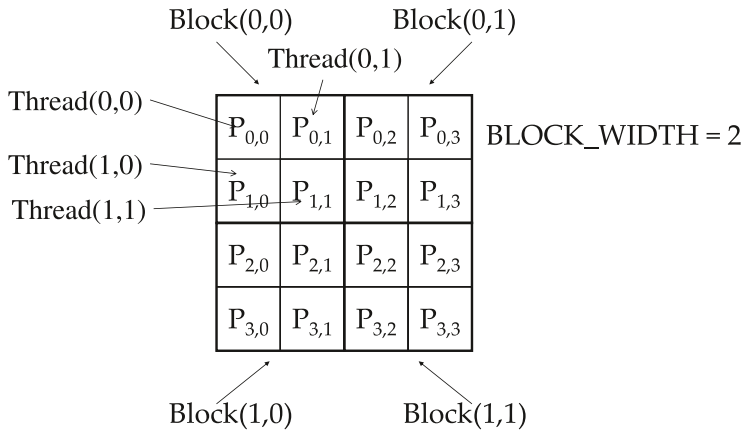
- The `MatrixMulKernel` function is defined to run on the GPU (indicated by `__global__`). It takes pointers to matrices M, N, P, and an integer `Width` representing the dimensions.
- Calculate Row Index (row 5). It calculates the row index of the current element in matrix P that the thread is responsible for.

- Calculate Column Index (row 8). It calculates the column index of the current element in matrix P.
- The if condition ensures that the thread operates within the bounds of the matrices.
- If the condition is true, initializes the `Pvalue` and iterates over the row of M and column of N. Each iteration accumulates the product of corresponding elements from M and N.
- Finally, stores the computed value in matrix P.

Since the previous example shows problems with **global memory access** (because accessing elements of matrices M, N, and P involves global memory), **optimization** (it doesn't load elements into shared memory to reduce global memory accesses), and **data locality** (because consecutive threads don't access successive memory locations), we propose an alternative.

Alternative (better) implementation of the previous example

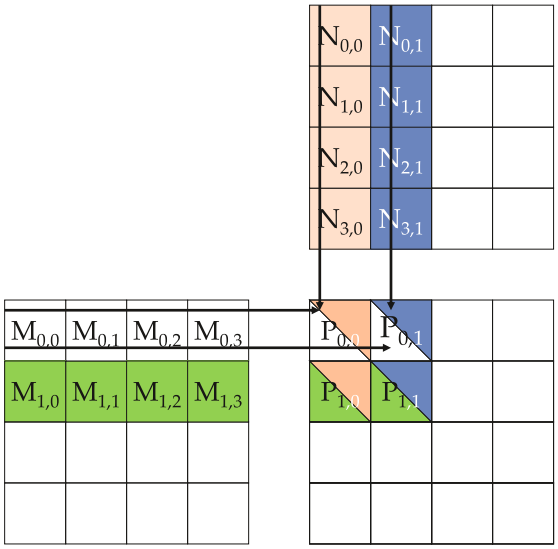
The following figure shows how threads are mapped to elements of the output matrix P during matrix multiplication. It shows how the matrix is divided into blocks and how threads handle these blocks.



- The 4×4 matrix P is divided into smaller 2×2 blocks. This division helps in managing data more efficiently and improves memory access patterns.
- The `BLOCK_WIDTH` is 2, indicating that each block contains 2×2 elements.
- Each block of the matrix P is assigned to a specific thread.
  - Block(0,0) is handled by Thread(0,0).
  - Block(0,1) is handled by Thread(0,1).
  - Block(1,0) is handled by Thread(1,0).
  - Block(1,1) is handled by Thread(1,1).

The **mapping ensures that each thread is responsible for computing the values of a specific block** in matrix P.

In the following figure, we illustrate an example of how elements $P_{0,0}$ and $P_{0,1}$ in matrix P are calculated using elements from matrices M and N.



- To compute $P_{0,0}$, are used elements from:
  - The first row of matrix M $(M_{0,0}, M_{0,1}, M_{0,2}, M_{0,3})$
  - The first column of matrix N $(N_{0,0}, N_{1,0}, N_{2,0}, N_{3,0})$

  The computation involves multiplying corresponding elements and summing the results:

$$\begin{aligned} P_{0,0} &= (M_{0,0} \times N_{0,0}) + (M_{0,1} \times N_{1,0}) + \\ &\quad (M_{0,2} \times N_{2,0}) + (M_{0,3} \times N_{3,0}) \end{aligned}$$

- To compute $P_{0,1}$, are used elements from:
  - The first row of matrix M $(M_{0,0}, M_{0,1}, M_{0,2}, M_{0,3})$
  - The second column of matrix N $(N_{0,1}, N_{1,1}, N_{2,1}, N_{3,1})$

  The computation involves multiplying corresponding elements and summing the results:

$$\begin{aligned} P_{0,1} &= (M_{0,0} \times N_{0,1}) + (M_{0,1} \times N_{1,1}) + \\ &\quad (M_{0,2} \times N_{2,1}) + (M_{0,3} \times N_{3,1}) \end{aligned}$$

And ***why is this related to storage and data locality?*** For three reasons:

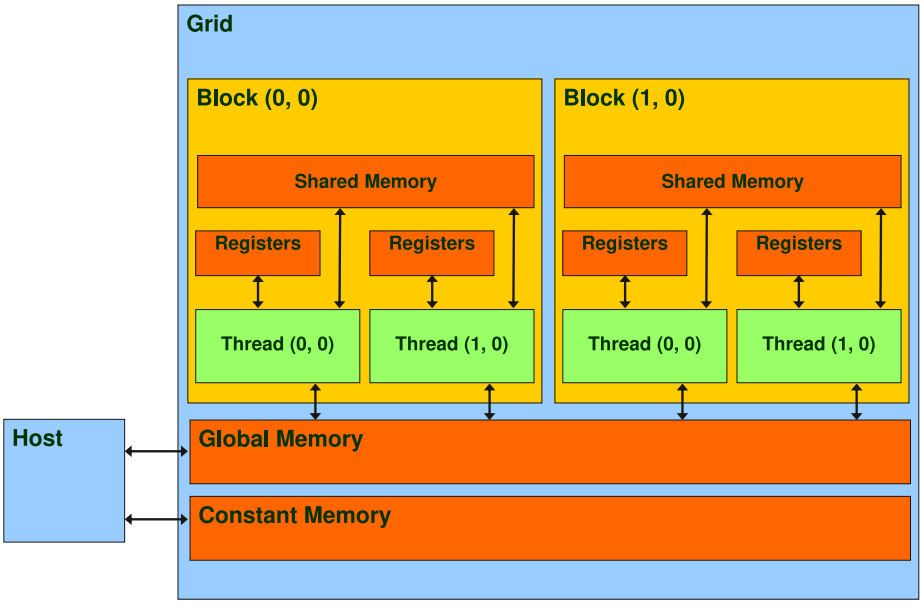
✔ **Memory Access Patterns**. By dividing the matrix into **blocks and assigning specific threads to handle these blocks**, the computation can take advantage of memory locality. **Threads working on the same block will likely access contiguous memory locations**, which improves memory access efficiency.

✔ **Shared Memory Usage**. Instead of repeatedly accessing global memory, **threads can load the necessary block data into shared memory once**. All **threads in a block can then work on the data in shared memory**, significantly reducing global memory access and thus improving performance.

✔ **Optimization for Performance**. Reducing the number of global memory accesses by optimizing data locality and using shared memory can drastically improve the performance of the matrix multiplication kernel. Efficient **mapping of threads to data and making use of memory hierarchy are key strategies for achieving high performance in CUDA applications**.

Memory/Data Locality are fundamental, now we go deep

Before explaining the keywords used by CUDA, it is important to understand the **programmer's view** of CUDA memory.



- **Host** (CPU) **Memory**. Connected to the GPU, but *distinct* from GPU memory.
- **Global Memory**. Accessible by *all threads* and the *host*. It's the *largest* but also the *slowest* type of memory.
- **Constant Memory**. Read-only memory accessible by *all* threads.
- **Grid**. Contains multiple blocks, each consisting of threads (we have already seen and discussed this in Section 2, page 102).
- **Shared Memory**. Faster than global memory, shared among threads within the same block.

- **Registers**. The fastest memory, local to each thread.

CUDA has specific declarations for different types of memory:

- **Local Variable** (`int LocalVar`): stored in registers, scope and lifetime are per thread.
- **Shared Variable** (`__shared__ int SharedVar`): stored in shared memory, scope and lifetime are per block.
- **Global Variable** (`__device__ int GlobalVar`): stored in global memory, scope is the entire grid, and the lifetime is the duration of the application.
- **Constant Variable** (`__constant__ int ConstantVar`): stored in constant memory, scope is the grid, and the lifetime is the duration of the application.

Variable Declaration	Memory	Scope	Lifetime
`int LocalVar`	Register	Thread	Thread
`__shared__ int SharedVar`	Shared	Block	Block
`__device__ int GlobalVar`	Global	Grid	Application
`__constant__ int ConstantVar`	Constant	Grid	Application

Table 4: CUDA Memory Types, Scope, and Lifetime.

There are two observations to make:

1. The `__device__` keyword is **optional** when used with `__shared__` or `__constant__`.
2. The **automatic variables** (those that are automatically managed by the compiler) **reside in the registers** because they are the fastest type of memory available to threads (very close to the processor).

   However, there's an **exception for per-thread arrays**: when we declare an array inside a kernel function, this array is unique to each thread and can be relatively large. Due to their size and potential complexity, these **per-thread arrays cannot fit in registers and are stored in global memory instead**.

And how do I decide where to put the variables?

It depends on the implementation. The good question to ask is: *can the host access the declared variable?*

- If the **host can access the variable**, it should be a `global` or `constant` variable.
  - Declared outside of a function;
  - Accessible by both host (CPU) and device (GPU).

- If the **host cannot access the variable**, it should be a `register` or `shared` variable.
  - Declared inside the kernel function;
  - If it is `shared`, it is accessible to all threads within the same block;
  - On the other hand, if it is `register` (local), it is only accessible to individual threads, suitable for frequently accessed variables.

In-depth analysis of shared memory in CUDA

Some features of shared memory in CUDA:

1. **Shared Memory in Each Streaming Multiprocessor (SM)**. Each SM in a CUDA GPU has its own shared memory.

   Shared memory is significantly faster than global memory, both in terms of latency and throughput.

2. **Scope and Lifetime**. The scope of shared memory is limited to the block; only threads within the same block can access the same shared memory.

   The lifetime of shared memory is tied to the thread block's lifetime. Once the block finishes execution, the shared memory is released.

3. **Access**. Shared memory is accessed using memory load/store instructions.

   It acts as a scratchpad memory in the computer architecture, allowing threads to quickly share and exchange data.

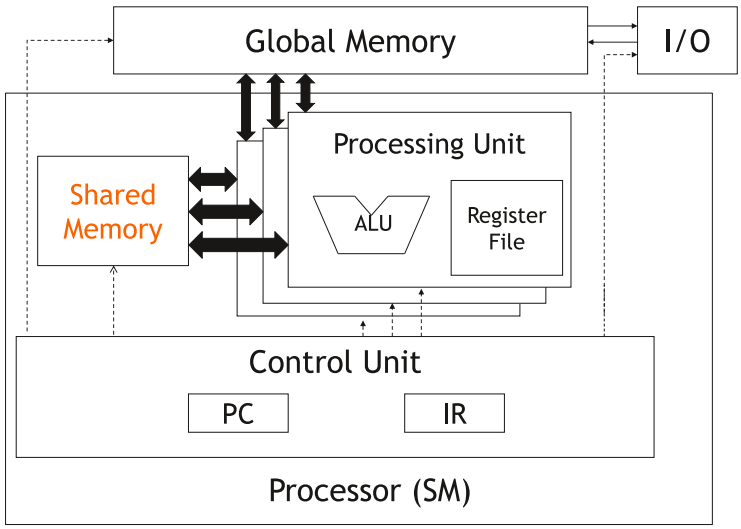


Figure 27: Hardware View of CUDA Memories.

- **Global Memory**. A large memory space accessible by all threads and the host (CPU). Used for data storage and retrieval but has higher latency compared to shared memory.
- **Shared Memory**. A smaller, faster memory space within each SM. Used for data sharing among threads in the same block.
- **Processing Unit**.Contains Arithmetic Logic Units (ALUs) and a Register File for performing computations. Registers are the fastest type of memory, used for storing per-thread data.
- **Control Unit**. Manages the execution of instructions, including the Program Counter (PC) and Instruction Register (IR).
- **I/O**. Represents input/output operations.

6.3.8 Tiling Technique

The **Tiling Technique**, also known as ***blocked matrix multiplication*** or ***tiling***, is a strategy to **enhance the performance of matrix computations by optimizing memory access patterns**. It leverages shared memory in CUDA to reduce the number of global memory accesses, improving efficiency and throughput. A lot of parallel algorithms adopt the tiling technique.

What is Tiling?

Tiling can be thought of as the **process of breaking a large matrix into smaller sub-matrices** (called *tiles* or blocks) **that can fit into faster**, **limited shared memory**. The main goal of tiling is to:

- **Minimize** slower **global memory accesses**;
- **Maximize** the **use of faster shared memory**.

Instead of a large matrix, we can think of the global memory contents as tiles and focus the computation of CUDA threads on one or a small number of tiles at a time.

Great analogy to understand the basic concept of Tiling

Reducing the number of vehicles in a congested traffic system can significantly improve the delays experienced by all vehicles. This is analogous to carpooling for commuters. We can image:

- **Drivers**: represent *threads accessing their memory data operands.*
- **Cars**: represent *memory access requests.*

Just as carpooling reduces the number of cars on the road, tiling reduces the number of global memory accesses by loading data into shared memory. The result is a reduction in traffic (memory access requests) and an obvious improvement in overall efficiency.

Unfortunately, just like in real life, there are some problems. For example, there are the **challenges of carpooling**. In fact, some carpools are easier to organize than others because the participants need to have similar work schedules. So certain vehicles may be more suitable for carpooling. However, other **commuters may have different needs**, **so the number of carpools may increase and there is a risk of increasing traffic again**.

Similar challenges exist in tiling calculations. Some computations may be easier to tile based on data access patterns and the nature of the computation. **Organizing data and computations efficiently to fit into tiles may be more challenging for certain algorithms**. So what is the **general euristic to adopt in order to use the tiling technique** or not? In general, it is:

✔ **Good to use** tiling when people have similar schedule. In computing, is good **when threads have *similar access timing***.

✖ **Bad to use** tiling when people have very different schedule. In computing, is bad **when threads have very *different timing***.

But synchronization is also important. Just as workers' schedules must be aligned for effective carpooling, **threads** in parallel computing **must be synchronized** for efficient execution. Efficient data access and memory usage depend on synchronized operations to reduce latency and improve performance. The following figure shows synchronization between multiple threads.

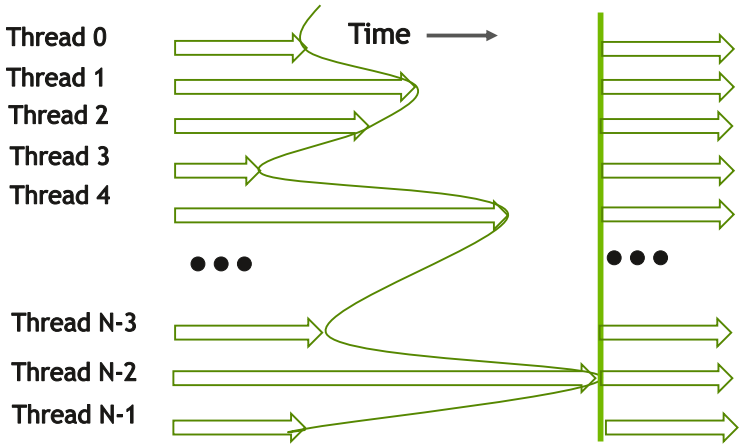


Figure 28: Barrier Synchronization for Tiling.

On the left are multiple threads (`Thread 0`, `Thread 1`, `Thread 2`, ..., `Thread N-1`) progressing over time. The wave represents when each thread reaches its barrier on the code; at that point, the thread must wait until all threads have arrived before it can continue. This ensures that all threads are synchronized at certain points during execution. Execution of all threads resumes when the last thread (in the picture, `Thread N-2`) reaches the barrier.

Synchronization is fundamental in the tiling technique because it **ensures that all threads have their share of data loaded into shared memory and are ready to proceed before they perform any computations**. After the computation, they use another synchronization barrier to ensure that all threads have completed their work before moving to the next tile.

Summary - How it works

1. **Identify a Tile**. Determine a section of global memory content that multiple threads will access. Dividing the workload into smaller tiles allows for efficient memory access and utilization of shared memory.
2. **Load the Tile**. Transfer the tile from global memory to on-chip memory (shared memory). Loading data into shared memory reduces the latency associated with accessing global memory.
3. **Barrier Synchronization**. Ensures that all threads are ready to begin the computation phase. It also ensures that all threads have the necessary data before starting the computation.
4. **Access Data**. Multiple threads access their data from the on-chip memory. Threads perform computations using the data stored in shared memory, benefiting from its faster access time.
5. **Barrier Synchronization**. Ensure all threads have completed the current phase (computations).
6. **Next Tile**. Move on to the next tile and repeat the process.

Advantages

- **Improved Memory Access Patterns**. By loading data into shared memory, the number of global memory accesses is reduced, leading to better performance.
- **Higher Computational Throughput**. Tiling helps achieve higher computational throughput by leveraging the faster shared memory.

6.3.8.1 Tiled Matrix Multiplication

In the following section, we present an **example of matrix multiplication using the tiling technique** to illustrate the efficiency improvements brought about by the tiling technique.

Traditional approach

Classical matrix multiplication has the following characteristics

- Each thread accesses a row of matrix M and a column of matrix N.
- Each thread block accesses a strip of matrix M and a strip of matrix N.

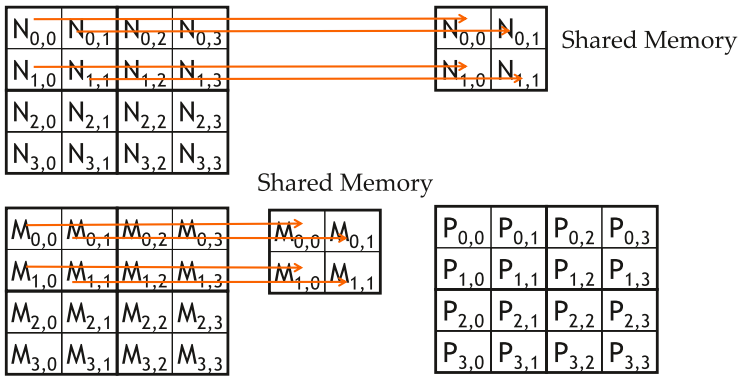
In the traditional approach, threads access data directly from global memory, which can be inefficient due to high latency.

Tiled Matrix Multiplication

1. As a **first step**, since we know the problem and (ideally) how the solution is implemented, we can try to brainstorm on how to apply tiling. In general, matrix operations lend themselves well to the tiling technique. In the matrix multiplication:
   - The execution of each thread can be broken into phases.
   - The data accesses by the thread block in each phase are focused on one tile of M and one tile of N.
   - The tile is of `BLOCK_SIZE` elements in each dimension.

2. As second step, the threads in a block participate in **loading items into shared memory using the tiling technique**.

   As we can see in the following figure, each thread in a block contributes to loading elements from matrices M and N into shared memory. Each thread is responsible for loading one element from the M matrix and one element from the N matrix into shared memory. This parallel loading ensures that data is moved quickly and efficiently from global memory to faster shared memory.

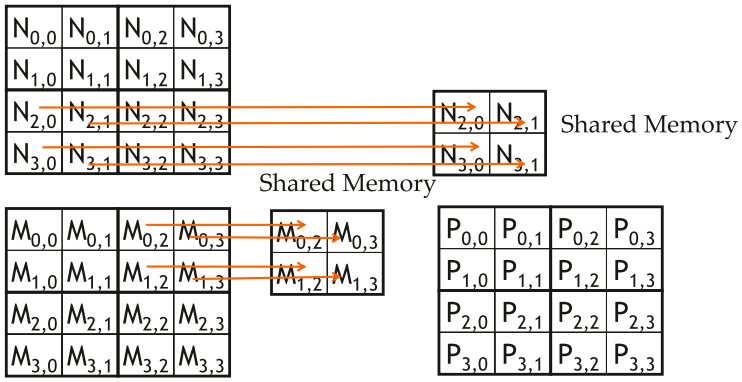


   The matrices M and N are the input and P is the result matrix. Each element is indexed by i and j (e.g., M_{ij}).

   - Elements of M are loaded into shared memory by the threads.
   - Similarly, elements of N are loaded into shared memory.

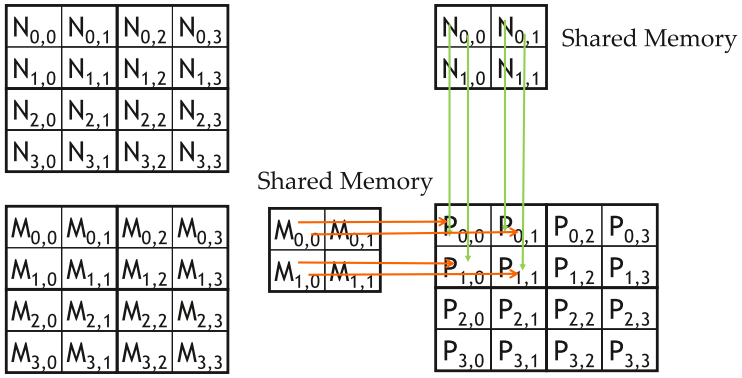
   This particular phase focuses on the initial loading of elements for the tile corresponding to block (0,0). Each thread in block (0,0) loads one element from M and one from N, populating the shared memory with the necessary data for the computation.

   These steps are performed for each block of the matrices. In the following figure, we can see another step of loading into shared memory.

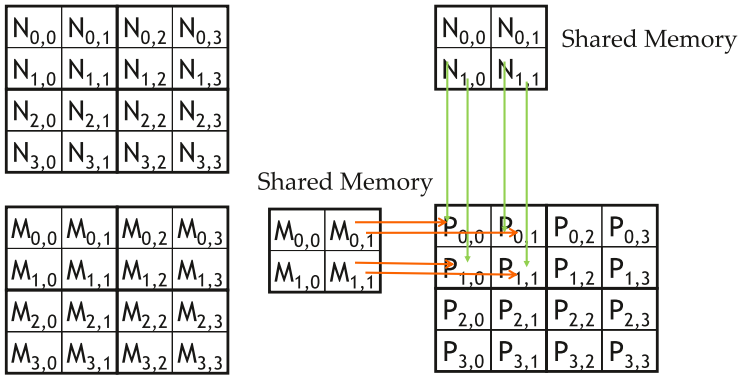


3. After loading into shared memory, there is the loading step, done in two iterations, into the result matrix. This is only a graphical representation, because in reality this step can be merged with the execution step. This step prepares the operands to be used in the execution step.

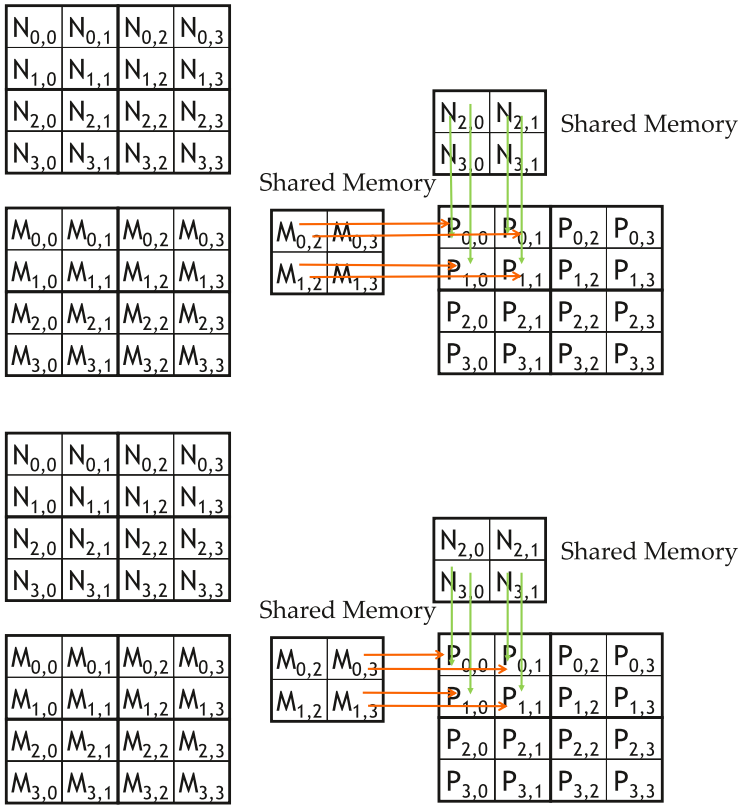
   (a) In the first iteration, the elements from the shared memory $N_{0,0}, N_{0,1}$ are used against the element from the shared memory $M_{0,0}, M_{1,0}$.



   (b) In the second iteration, the elements from the shared memory $N_{1,0}, N_{1,1}$ are used against the element from the shared memory $M_{0,1}, M_{1,1}$.



This step is done for each block of the matrices. In the figure on the next page, we can see two more steps of the load block.



4. After each load, there is the **execution phase**. In the following table, we highlight how four threads in a block perform matrix multiplication using shared memory.

	Phase 0			Phase 1		
$thread_{0,0}$	**$M_{0,0}$** ↓ $Mds_{0,0}$	**$N_{0,0}$** ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$+ $Mds_{0,1}$*$Nds_{1,0}$	**$M_{0,2}$** ↓ $Mds_{0,0}$	**$N_{2,0}$** ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$+ $Mds_{0,1}$*$Nds_{1,0}$
$thread_{0,1}$	**$M_{0,1}$** ↓ $Mds_{0,1}$	**$N_{0,1}$** ↓ $Nds_{1,0}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$+ $Mds_{0,1}$*$Nds_{1,1}$	**$M_{0,3}$** ↓ $Mds_{0,1}$	**$N_{2,1}$** ↓ $Nds_{0,1}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$+ $Mds_{0,1}$*$Nds_{1,1}$
$thread_{1,0}$	**$M_{1,0}$** ↓ $Mds_{1,0}$	**$N_{1,0}$** ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$+ $Mds_{1,1}$*$Nds_{1,0}$	**$M_{1,2}$** ↓ $Mds_{1,0}$	**$N_{3,0}$** ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$+ $Mds_{1,1}$*$Nds_{1,0}$
$thread_{1,1}$	**$M_{1,1}$** ↓ $Mds_{1,1}$	**$N_{1,1}$** ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}$*$Nds_{0,1}$+ $Mds_{1,1}$*$Nds_{1,1}$	**$M_{1,3}$** ↓ $Mds_{1,1}$	**$N_{3,1}$** ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}$*$Nds_{0,1}$+ $Mds_{1,1}$*$Nds_{1,1}$

time →

The values $\texttt{Mds}_{i,j}$ are the values loaded in the load step from global to shared memory of the M matrix. The same reasoning applies to `Nds`.

Once the data is loaded into shared memory, each thread uses the loaded values to update the partial product value `PValue` for the resulting matrix P:

- $\texttt{thread}_{0,0}$:

```
1 PValue0,0 += Mds0,0 * Nds0,0 + Mds0,1 * Nds1,0
```

- $\texttt{thread}_{0,1}$:

```
1 PValue0,1 += Mds0,0 * Nds0,1 + Mds0,1 * Nds1,1
```

- $\texttt{thread}_{1,0}$:

```
1 PValue1,0 += Mds1,0 * Nds0,0 + Mds1,1 * Nds1,0
```

- $\texttt{thread}_{1,1}$:

```
1 PValue1,1 += Mds1,0 * Nds0,1 + Mds1,1 * Nds1,1
```

5. **Barrier Synchronization step**. To synchronize all threads in a block, we use the function `__syncthreads()`. This function acts as a barrier synchronization, ensuring that all threads in a block reach that point before any thread can proceed.

   Barrier synchronization is particularly useful for coordinating the execution of tiled algorithms in phases. It is useful in:

   - ✔ **Loading Phase**. Ensures that **all elements of a tile are loaded into shared memory before any computation begins**.
   - ✔ **Computation Phase**. Ensures that **all threads have completed their computation on the current tile before moving to the next phase or tile**.

6.3.8.2 Implementation Tiled Matrix Multiplication

After a great explanation of how to apply the tiling technique to the matrix multiplication operation, we present the CUDA implementation. As we said, the goal of tiled matrix multiplication with CUDA is to optimize the matrix multiplication process by taking advantage of shared memory (small, fast memory space accessible to all threads within the same block).

Introduction to the implementation

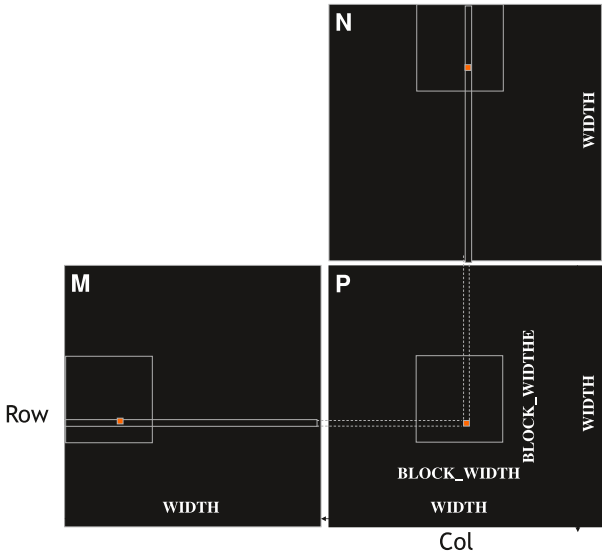
The main core of the implementation is about **tile indexing**. It can be 1 or 2 dimensional. In the following figures, to understand the logic, we show 2 dimensional indexing, which is more natural.

- When each thread loads the input from the original matrix (M or N), it needs an index.

```
int Row = by * blockDim.y + ty;
int Col = bx * blockDim.x + tx;
// 2D indexing for accessing Tile 0:
    M[Row][tx]
    N[ty][Col]
```

  At each iteration, each thread in a block considers the same row of the matrix M and the same column of the matrix N. To distribute the workload among the threads, we assign each thread to each column of the matrix M (using the unique index `tx`, the index of the thread in the CUDA block), and with the same reasoning, we assign each thread to each row of the matrix N. The row in the matrix M is fixed, but the column is taken entirely by the assignment of all threads in the block (using `tx`).

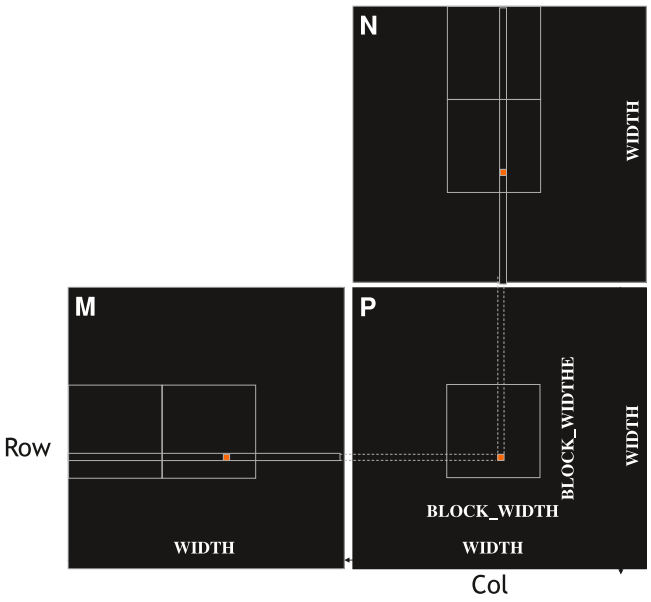
  In the following figure, we see that the row in the matrix M is fixed, but the column is fully occupied thanks to the assignment of all threads in the block (using tx).



When the first phase is finished, we move on to the second tile. For this reason, we use a kind of **offset** given by the formula $p \times$ `TILE_WIDTH` (`TILE_WIDTH` = `BLOCK_WIDTH`), where p is the number of the phase (at the beginning zero, then one, and so on).

In the following image, we see that at phase 1, to load the tile 1, we use the formula:

```
// 2D indexing for accessing Tile 1:
M[Row][1 * TILE_WIDTH + tx]
N[1 * TILE_WIDTH + ty][Col]
```



CUDA code

```
#define TILE_WIDTH 16

__global__ void MatrixMulKernel(
    float* M, float* N, float* P, int Width
) {
    __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
    __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x; int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;

    int Row = by * blockDim.y + ty;
    int Col = bx * blockDim.x + tx;
    float Pvalue = 0;

    // Loop over the M and N tiles
    // required to compute the P element
    int bound = Width / TILE_WIDTH;
    for (int p = 0; p < bound; ++p) {
        // Collaborative loading of M and N tiles
        // into shared memory
        ds_M[ty][tx] = M[Row * Width + p * TILE_WIDTH + tx];
        // = M[Row][p * TILE_WIDTH + tx]
        ds_N[ty][tx] = N[(p * TILE_WIDTH + ty) * Width + Col];
        // = N[p * TILE_WIDTH + ty][Col]
        __syncthreads();

        for (int i = 0; i < TILE_WIDTH; ++i) {
            Pvalue += ds_M[ty][i] * ds_N[i][tx];
        }
        __syncthreads();
    }
    P[Row * Width + Col] = Pvalue;
}
```

- **Index variables**:
  - `bx` and `by` are the **block indices** in the x and y directions.
  - `tx` and `ty` are the **thread indices** within a block.
  - `Row` and `Col` are the **row and column indices of the element in the output matrix**.
- **Main Loop**: `ph` (phase) determines which tile is currently being processed.
- **Loading Tiles into Shared Memory**:
  - Each thread loads one element from the current tile of M and N into shared memory.
  - `ds_M[ty][tx]` loads an element from M.
  - `ds_N[ty][tx]` loads an element from N.
  - `__syncthreads()` is called to ensure all threads have loaded their elements before proceeding.

- **Matrix Multiplication within a Tile**. Once the tiles are loaded into shared memory, each thread computes the **partial dot product for the corresponding element** in the output matrix. The nested loop (second for loop) accumulates the product of elements from M and N.
- Finally, after all tiles have been processed, the **final value is stored** in the output matrix P.

Final consideration

A bigger block is better and the reason is simple. **Bigger tiles mean more threads per block**. For example:

- `TILE_WIDTH` of 16 results in $16 \times 16 = 256$ threads per block.
- `TILE_WIDTH` of 32 results in $32 \times 32 = 1024$ threads per block.

Therefore, the *workload* per phase is:

- `TILE_WIDTH` = 16:
  - Each block performs 512 (2×256) float loads from global memory.
  - Executes **8192** ($256 \times (2 \times 16)$) **multiply-add operations** (16 floating point operations for each memory load).
- `TILE_WIDTH` = 32:
  - Each block performs 2048 (2×1024) float loads from global memory.
  - Executes **65536** ($1024 \times (2 \times 32)$) **multiply-add operations** (32 floating point operations for each memory load).

Although a larger block might be better, shared memory is not infinite. It depends on the implementation. If we take a classic Streaming Multiprocessor (SM) with 16KB of shared memory, when we do a memory usage analysis:

- `TILE_WIDTH` = 16:
  - Uses **2KB** ($2 \times 256 \times 4$B) **of shared memory per block**.
  - Allows up to **8 blocks per SM** in parallel execution (8×256 threads = 2048 threads).

  This allows up to 4096 (8×512) pending loads (2 per thread, 256 per block).
- `TILE_WIDTH` = 32:
  - Uses **8KB** ($2 \times 32 \times 32 \times 4$B) **of shared memory per block**.
  - Allows up to **2 blocks per SM** in parallel execution (limited by thread count). If a GPU limits the thread count to 1536 threads per SM, the number of blocks per SM is reduced to one!

Using `__syncthreads()` ensures that all threads reach a barrier before continuing, which can **temporarily reduce the number of active threads**. **More blocks per SM can be beneficial to balance memory usage and thread count**.

In summary, **choosing the right tile size and efficiently managing shared memory and threads are critical to optimizing GPU performance**. This balance affects how many operations can be performed simultaneously and how effectively memory is used.

6.3.8.3 Any size matrix handling

The following paragraph focuses on the challenge of **handling matrix multiplication for matrices of arbitrary size** using tiled matrix multiplication. Often, real-world applications require support for matrices that aren't always square or multiples of the tile width (`TILE_WIDTH`). So *what is the strategy for these situations?*

⚠ Limitations of the Tiled Matrix Multiplication presented

- The base kernel can only handle square matrices whose dimensions are multiples of `TILE_WIDTH`. This is a major limitation, since non-square matrices also exist.
- A possible solution would be to apply padding, but padding these matrices to match tile sizes can result in **significant overhead** in terms of space and data transfer time.

The basic idea is that instead of padding, a different technique is proposed to efficiently handle matrices of arbitrary size and prevent access to invalid memory locations.

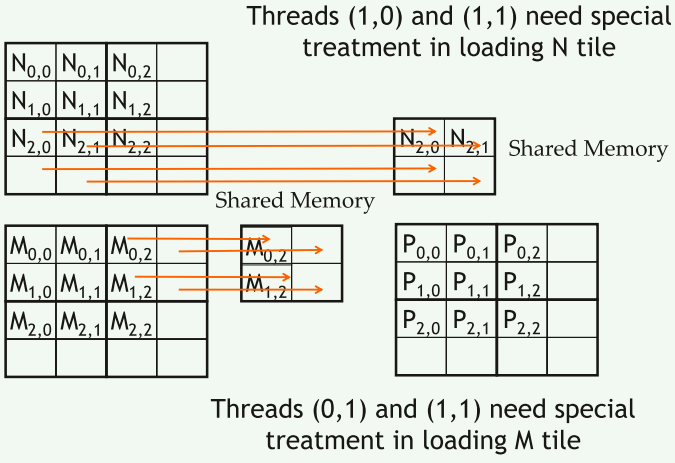
Example 5: graphical example

Imagine we have a 3×3 matrix and a `TILE_WIDTH` of 2. The grid and block configuration may result in some threads accessing positions outside the bounds of the 3×3 matrix.
The basic logic will be:

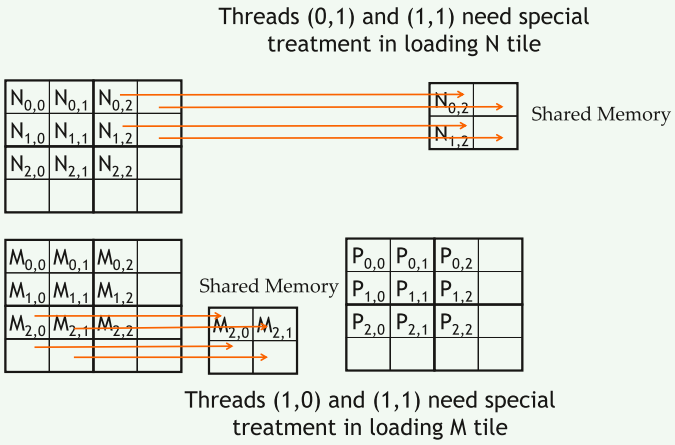
- Threads within the valid range of the matrix will load elements normally.
- Threads outside the valid range need conditions to avoid accessing or writing to out-of-bound elements.

Graphically, the special cases that we need to deal with will be:

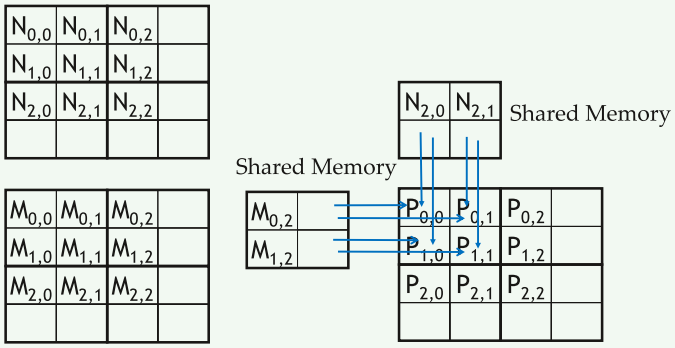
- When we load elements from the matrix into shared memory:



And:



- If we use the out-of-bound elements to calculate the result:



✔ A simple solution

The main simple and efficient solution is the **conditional loading**. When a thread is to load an input element, **check if the element index is within valid bounds**:

- If valid: proceed to **load the element**.
- If invalid: do not load the element, but **write a 0** instead.

Assigning a **0 value ensures that the multiply-add step does not affect the final value of the output element**. Also, this simple check helps **avoid errors from invalid memory access and ensures correctness**.

Why do we have to load a zero? We cannot skip this part directly?

1. **Safety and Validity**. When a thread loads a 0 instead of an out-of-bound element, it **ensures that the memory access is valid and does not cause runtime errors or fetch garbage data**. This is crucial for maintaining the stability of the program.

2. **Maintaining Computational Integrity**. The 0 value acts as a **neutral element in multiplication**, meaning it does not alter the final sum during the multiply-add operations. For example:

   ```
   sum += tileM[ty][k] * tileN[k][tx]
   ```

   If `tileM[ty][k]` or `tileN[k][tx]` is 0, the sum remains unaffected by that particular operation, preserving the integrity of the calculation.

3. **Facilitating Parallel Execution**. In CUDA, all threads in a warp (a group of 32 threads) execute the same instruction simultaneously. If some threads are out-of-bound, they still participate in the synchronization and memory access patterns, **ensuring the warp executes efficiently without branching or divergence**.

   By loading 0s, these threads can reach synchronization points (such as `__syncthreads()`) together with other threads that load valid data, ensuring all threads in the block stay in sync.

   A **Warp Divergence** occurs when **threads in a warp follow different execution paths due to conditional branches**. This means some threads are active while others are idle, leading to inefficiency. The consequences of warp divergence can have a **severe impact on performance**.

   - *Performance Impact*:
     - Divergence causes **some threads to stall while others execute**, reducing the overall throughput.
     - **All threads in the warp must eventually reconverge to execute the same instructions again**, prolonging the execution time for that warp.
   - *Synchronization Issues*:
     - Skipping `__syncthreads()` or similar synchronization points can lead to incorrect behavior. Threads that do not wait might access incomplete or inconsistent data in shared memory, leading to incorrect results.
     - **Ensuring that all threads in a warp (or block) reach synchronization points is crucial for data consistency**.

4. **Shared Memory Utilization**. All threads, including those loading 0s, contribute to filling the shared memory tiles. This **ensures that when the matrix multiplication is performed, the shared memory contains a complete tile** (with zeros in out-of-bound areas). This approach guarantees that the shared memory tiles are properly used for the block's calculations, and threads with 0 elements help maintain the structure and coherence of these tiles.

Implementation

The code snippet is as follows:

```
for (int p = 0; p < (Width - 1) / TILE_WIDTH + 1; ++p) {
    if (Row < Width && p * TILE_WIDTH + tx < Width) {
        ds_M[ty][tx] = M[Row * Width + p * TILE_WIDTH + tx];
    } else {
        ds_M[ty][tx] = 0.0;
    }
    if (p * TILE_WIDTH + ty < Width && Col < Width) {
        ds_N[ty][tx] = N[(p * TILE_WIDTH + ty) * Width + Col];
    } else {
        ds_N[ty][tx] = 0.0;
    }
    __syncthreads();

    // Ensuring valid computations and
    // writing to the output matrix
    if (Row < Width && Col < Width) {
        for (int i = 0; i < TILE_WIDTH; ++i) {
            Pvalue += ds_M[ty][i] * ds_N[i][tx];
        }
    }
    __syncthreads();
}

if (Row < Width && Col < Width) {
    P[Row * Width + Col] = Pvalue;
}
```

- Matrix M. Each thread loads:
  - An element in the position `M[Row][p * TILE_WIDTH + tx]` if the row is less than the width of the matrix (`Row < Width`) and the tile number (`p * TILE_WIDTH`) plus the index number of the thread `tx` is less than the width of the matrix (`p * TILE_WIDTH + tx < Width`).
  - Otherwise, load 0.
- Matrix N. Each threads loads:
  - An element in the position `N[p * TILE_WIDTH + ty][Col]` if the column is less than the width of the matrix (`Col < Width`) and the tile number (`p * TILE_WIDTH`) plus the index number of the thread `ty` is less than the width of the matrix (`p * TILE_WIDTH + ty < Width`).
  - Otherwise, load 0.
- Boundary condition `(Width - 1) / TILE_WIDTH + 1`:
  - `Width - 1`: adjusts the maximum index for a zero-based index system. Essentially, it ensures we account for the last valid index in the matrix.
  - `/ TILE_WIDTH`: divides the total width by the tile width, determining how many full tiles fit within the width.

– `TILE_WIDTH + 1`: ensuring that any remaining part of the matrix that doesn't fit perfectly into the last tile is also processed. If the width isn't an exact multiple of the tile width, the +1 ensures that an additional tile is considered to cover the remaining part of the matrix.

6.3.9 Optimizing Memory Coalescing

In this chapter, we introduce the importance of memory coalescing for effectively using memory bandwidth in CUDA.

Introduction

Memory bandwidth is important in CUDA because each thread may need to access memory, and efficient memory access is critical to overall performance. Ideally, when multiple threads in a CUDA program access memory, their accesses should be *coalesced*. This means that **memory accesses from multiple threads are combined into a single memory transaction**.

The memory type inside the GPU is DRAM. Since DRAM accesses are not as fast as local registers, an optimization is required. For this reason, the technique of *DRAM bursts* is introduced:

- It allows a block of data to be transferred in one go. When CUDA threads coalesce memory accesses, the entire burst can be used effectively, resulting in *fewer memory transactions and higher memory bandwidth utilization.*
- By coalescing memory accesses into fewer DRAM bursts, the *latency (delay) associated with memory access is reduced.* This is because once a burst is initiated, the additional data within the burst can be accessed quickly.
- CUDA developers use a variety of techniques to ensure that thread memory accesses are concatenated to take full advantage of DRAM burst capabilities. These include *aligning data structures and managing memory access patterns.*

For **example**, consider a CUDA kernel where each thread accesses successive memory locations. When these accesses are combined, a single DRAM burst can fetch multiple data points needed by multiple threads, *significantly speeding up the computation.*

What is a DRAM burst?

A **DRAM (Dynamic Random Access Memory) burst** is a way of **reading data from or writing data to memory**. When accessing DRAM, **data** isn't **retrieved** one byte at a time, but rather **in bursts**. A DRAM burst involves the **transfer of multiple bytes of data in a single**, **continuous sequence**, rather than in separate, discrete chunks.

- **Non-Burst Timing**. There are **gaps between data transfers**, resulting in inefficiencies.
- **Burst Timing**. Represents a **continuous sequence of data transfer**, illustrating the efficiency of burst mode. **Burst bytes are transmitted to the processor, but may be discarded if accesses are not sequential**.

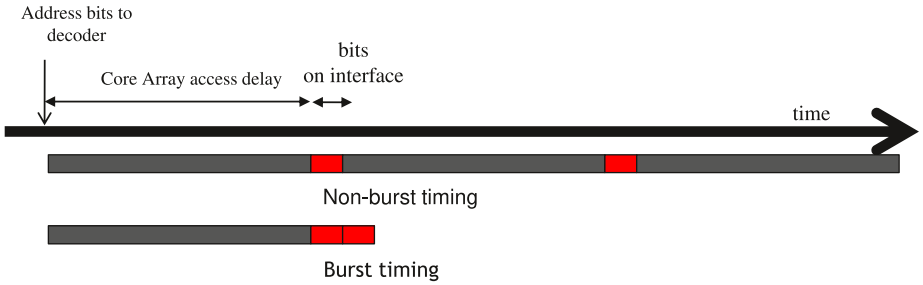


Figure 29: The figure shows a comparison of non-burst and burst timing in DRAM systems.

Burst mode minimizes the gaps between data transfers, allowing for faster and more efficient data flow. This is in contrast to non-burst mode, where data transfers are inefficient due to gaps.

In more detail, burst mode can be **described by the following features**:

- **Burst operation**. **When accessing a specific memory address, the DRAM retrieves an entire block of adjacent data, called a burst**. This allows faster data access than retrieving individual bytes one at a time.
- **Burst Length**. The length of the burst indicates **how many bytes are transferred in one operation**. Common burst lengths are 4, 8, or 16 bytes, but larger lengths are used depending on the system and application.
- **Memory efficiency**. This method improves memory access efficiency. **Once the initial memory location is accessed, the DRAM can transfer the rest of the data in the burst with less overhead**.
- System Usage. Burst transfers are particularly useful in applications where large blocks of data must be moved quickly, such as graphics processing where textures and images are frequently accessed.

Example 6: great analogy with burst technology

We can think of burst mode as a bookshelf, where instead of taking out one book at a time, we take out a whole section at once. This way, we get more books (data) in one go, reducing the time it takes to reach for each book individually.

The following figure shows a system view of the DRAM burst. Each address space is divided into burst sections, and when one location is accessed, all other locations in the same section are also delivered to the processor.

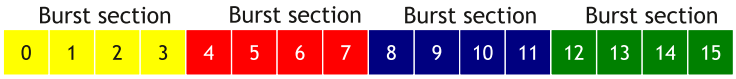


Figure 30: In the figure, a 16-byte address space is divided into 4-byte burst sections. In practice, address spaces are much larger (e.g., 4 GB) and burst sections are also larger (e.g., 128 bytes or more).

⚖ Coalesced access vs un-coalesced access

Memory Coalescing refers to the **process of combining** (or *coalescing*) **multiple memory accesses by different threads into a single memory transaction**. This is important in the context of GPUs because they run many threads in parallel, and efficient memory access can significantly improve performance.

When threads in a warp (a group of threads running in parallel) **access successive memory locations, these accesses are combined into a single memory transaction**. This means that all requested data is retrieved in one go.

When threads access memory locations that are scattered or misaligned, multiple memory transactions are required, leading to inefficiencies. This phenomenon is called **Un-Coalesced Access**.

Un-Coalesced Accesses occur when the **memory requests of multiple threads in a warp do not fit properly into a single memory transaction**. This inefficiency results in *increased memory latency* and *lower overall performance*. It is manifested when:

- **Non-Sequential Memory Access**. When threads access memory addresses that are not contiguous, the **memory controller must handle each access individually or in smaller, less efficient chunks**.
- **Crossing Burst Boundaries**. When memory accesses span multiple DRAM burst sections, **multiple memory transactions are required**. Each burst section may only be able to handle a portion of the requested data, resulting in additional overhead.
- **Thread misalignment**. If the **data being accessed by threads is misaligned with the natural boundaries of memory bursts**, the accesses cannot be merged effectively. This often happens with poorly structured data layouts.

The un-coalesced accesses are a **penalty for memory optimization because the memory controller cannot combine these accesses into a single transaction because they are spread out**. Instead, it must handle multiple transactions, each fetching only a few bytes relevant to the threads.

The **consequences of un-coalesced access** are:

- **Increased DRAM transactions**. **Each un-coalesced access requires a separate DRAM burst**, increasing the number of memory transactions.
- **Wasted Bandwidth**. **Not all bytes in a DRAM burst are utilized**. For example, if each burst fetches 32 bytes and only 8 bytes are used by threads, the remaining 24 bytes are wasted.
- **Higher Latency**. More transactions means more latency because **each transaction requires setup and transmission time**.

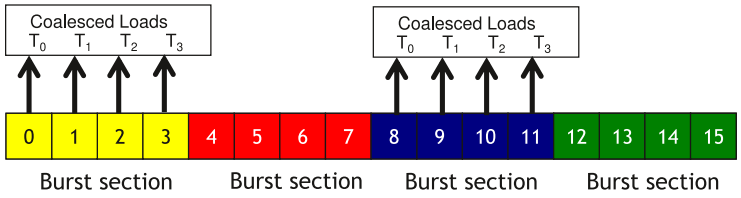


Figure 31: When all threads of a warp execute a load instruction, if all accessed locations are in the same burst section, only one DRAM request is made and the access is fully merged.

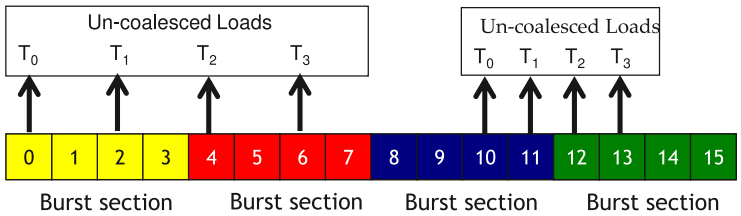


Figure 32: If the accessed locations are spread across burst boundaries, coalescing fails (un-coalesced accesses), multiple DRAM requests are made. This results in some garbage bytes that are not used by threads.

Example 7: great analogy to un-coalesced access

Think of un-coalesced access as trying to get multiple items from different aisles in a supermarket, one at a time. We end up walking back and forth more and taking longer to collect all the items than if we collected items from a single, well-organized aisle.

Example 8: difference from coalesced and un-coalesced access

Suppose a warp of 32 threads accesses an array in global memory. If each thread accesses elements sequentially (thread 0 accesses element 0, thread 1 accesses element 1, and so on), **a single memory transaction retrieves all 32 elements**.
In contrast, if the same warp accesses memory in a non-sequential manner (thread 0 accesses element 0, thread 1 accesses element 4, thread 2 accesses element 8, and so on), **multiple memory transactions are required**, reducing efficiency.

How do we guarantee coalesced access?

Accesses in a warp are to consecutive locations **if the index in an array access is in the form of**:

```
(expression with terms independent of threadIdx.x) + threadIdx.x
```
(12)

This formula ensures coalesced memory access in GPU programming.

- `threadIdx.x`: represents the **thread index** within a warp.
- `Expression Independent of threadIdx.x`: this part of the formula ensures that the **base index is the same for all threads within a warp**. It's crucial for aligning memory access.

But *why does it work?* For two main reasons:

1. **Consecutive Access**: if the base index (the part of the expression that is independent of `threadIdx.x`) is the same for all threads, then adding `threadIdx.x` to that base index means that each thread is accessing a consecutive location.
2. **Memory Coalescing**: using this formula ensures that all threads within a warp are accessing adjacent memory locations, resulting in coalesced access.

Before we continue, we need to understand **how a matrix is stored in memory**. It depends on the language, the implementation, and so on, but in general, when we say *linear memory space*, a 2-dimensional matrix is stored as in Figure 33 (page 157).

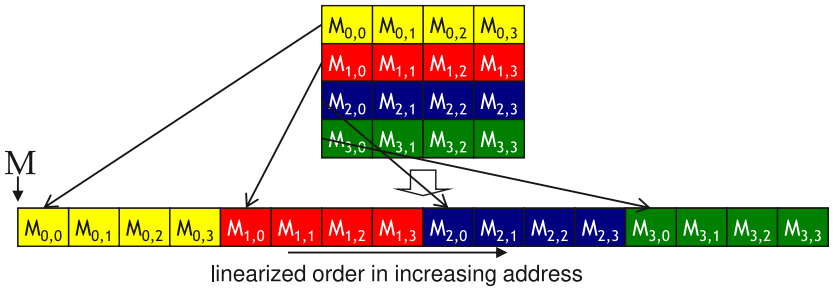


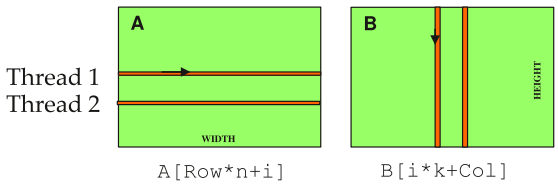
Figure 33: Linear memory space.

In GPU programming, matrix multiplication involves two main access patterns:

- Matrix `A` (left operator): the access pattern is `A[Row * n + i]`
  - `i` is the **loop counter** in the inner product loop of the kernel code;
  - `Row` is the **current row being processed**;
  - `n` is the **number of columns** in matrix `A`.
- Matrix `B` (right operator): the access pattern is `B[i * k + Col]`
  - `i` is the **loop counter**;
  - `k` is the **number of columns** in matrix `B`;
  - `Col` is calculated as:

    ```
    blockIdx.x * blockDim.x + threadIdx.x
    ```

    This means that **threads access elements in matrix B based on their column index**.



- Matrix `B`. The accesses are **Coalesced**. As we can see from the figure 34, the accesses are coalesced. This is because matrix `B` is stored in memory in a linear, one-dimensional array. For a 2D matrix, the elements are stored row by row.

  In a warp, threads are indexed consecutively, like `threadIdx.x` $= 0, 1, \ldots$. When `i` is fixed for a loop iteration, successive threads access successive memory locations in matrix `B`. For example:

  - Thread 0 accesses `B[i * k + 0]`
  - Thread 1 accesses `B[i * k + 1]`
  - Thread 2 accesses `B[i * k + 2]`

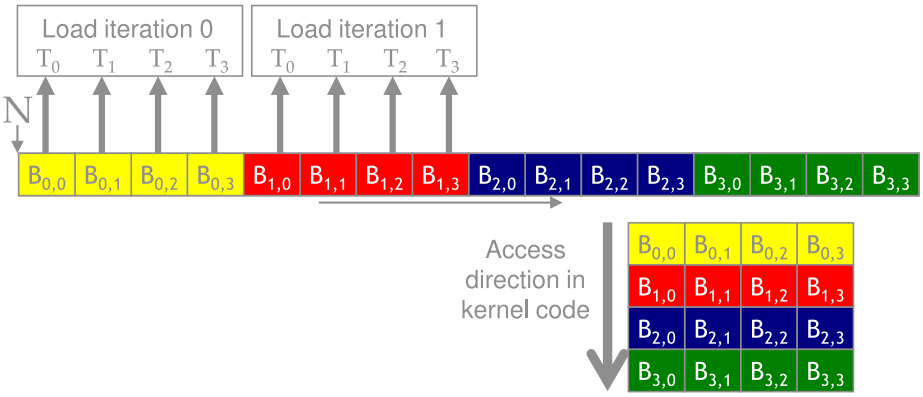


Figure 34: Coalesced accesses.

The main reason for coalesced access in matrix `B` is to **align thread indexes with contiguous memory addresses**. This optimizes memory transactions, making data retrieval more efficient and increasing overall performance.

- Matrix `A`. The accesses are **Un-Coalesced**. The figure shows why the accesses for matrix A are not coalesced.

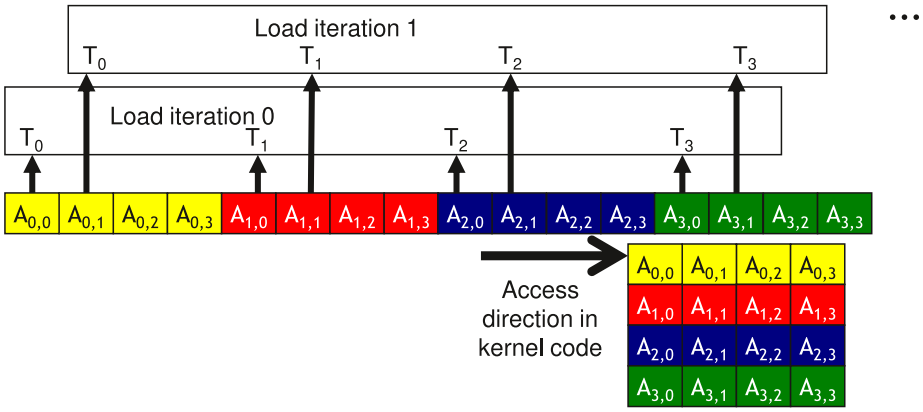


Figure 35: Un-Coalesced Accesses.

The access pattern `A[Row * n + i]` means that each thread in a warp accesses different rows of the matrix `A`. If we consider consecutive threads (e.g. thread 0, thread 1, thread 2, thread 3), they access elements in different rows.

Threads in a warp access elements vertically, not horizontally. For example, if `Row` is fixed for a given iteration of the outer loop, `i` varies. This means that thread 0 accesses `A[Row * n + 0]`, thread 1 accesses `A[Row * n + 1]`, and so on. This results in non-consecutive memory locations being accessed by consecutive threads.

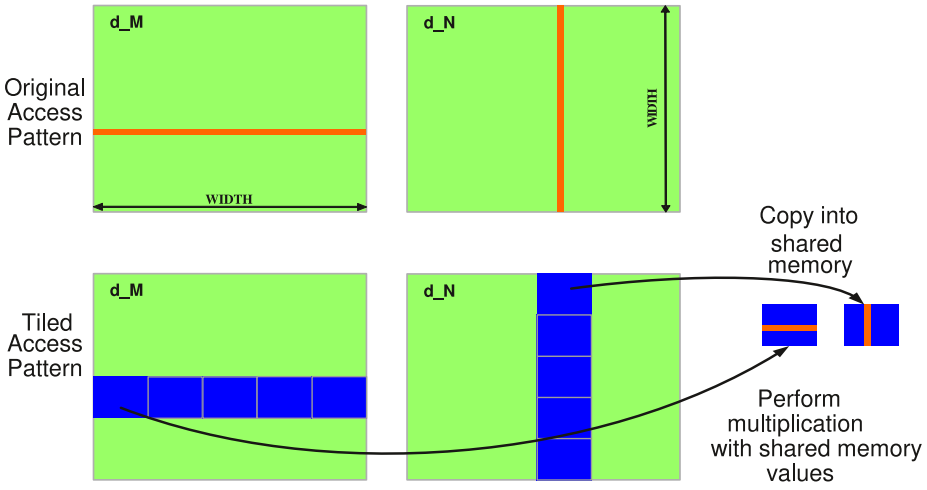
How to optimize the matrix multiplication to avoid un-coalesced accesses

Corner Turning is a **technique used in matrix multiplication to optimize memory access patterns**, especially in parallel computing environments like GPU. The goal is to achieve memory coalescing, which improves performance by ensuring that memory accesses are aligned and contiguous.

Corner Turning works as follows:

- **Column-Major Layout**. Normally, matrices are stored in a row-major layout (elements are stored row by row). In corner turning, the **second matrix** (`B`) is **stored in a column-major layout** (elements are stored column by column).
- **Tiled Access**. The **matrices are divided into smaller submatrices or tiles**. Each thread loads a tile of the first matrix (`A`) and a tile of the second matrix (`B`) into shared memory.
- **Memory Coalescing**. **By loading tiles into shared memory, threads can access consecutive memory locations within the tiles**. This ensures that memory accesses are coalesced, reducing memory latency and increasing bandwidth utilization.

In practice, it works as shown in the following figure:



- Original Access Pattern:
  - Matrix `d_M`. The original access pattern is **horizontal**. **Each thread accesses consecutive elements in a row**.
  - Matrix `d_N`. The original access pattern is **vertical**. **Each thread accesses elements in different rows**.

- Tiled Access Pattern:
  - Matrix `d_M`. The matrix is **divided into smaller horizontal tiles**. Each tile is a small submatrix that fits into shared memory.
  - Matrix `d_N`. Similarly, this matrix is **divided into smaller vertical tiles**, which are also small enough to fit into shared memory.
- Shared Memory Operations:
  - **Loading Tiles into Shared Memory**. Both tiles from `d_M` and `d_N` are loaded into shared memory. This allows for *more efficient access patterns.*
  - **Matrix Multiplication**. The **multiplication of these tiles is performed using the values stored in shared memory**. This step ensures that the data access is coalesced and reduces latency.

By dividing matrices into tiles and loading them into shared memory, **threads can access successive memory locations within a tile**. This ***ensures that memory accesses are concatenated***, reducing memory transactions and increasing efficiency.

In addition, using shared memory for these tiles reduces the need to repeatedly access global memory, which is slower. Shared memory access is much faster, further improving the performance of matrix multiplication.

From a code point of view, we see:

- Original Pattern:

```
Matrix d_M (Row-major):
Thread 0: M[0][0], M[0][1], ...
Thread 1: M[1][0], M[1][1], ...

Matrix d_N (Column-major):
Thread 0: N[0][0], N[1][0], ...
Thread 1: N[0][1], N[1][1], ...
```

- Tiled Pattern:
  1. Load Tiles into Shared Memory:

```
Tile from d_M: [[M00, M01], [M10, M11]]
Tile from d_N: [[N00, N10], [N01, N11]]
```

  2. Perform Multiplication. Threads use values from shared memory for the multiplication, ensuring coalesced access and reduced memory latency.

7 CUDA

7.1 Introduction

CUDA, which stands for **Compute Unified Device Architecture**, is a **parallel computing platform and application programming interface (API) model created by NVIDIA**. It allows developers to use the power of GPUs (Graphics Processing Units) for general-purpose processing, which enables substantial performance improvements for computationally intensive tasks.

Why CUDA?

GPUs, originally designed to render graphics, have evolved into **highly efficient and powerful processors capable of handling thousands of threads simultaneously**. This transformation has made GPUs, and by extension CUDA, incredibly valuable for applications requiring massive parallelism, such as scientific simulations, machine learning, and deep learning.

Why can we not just use the CPU?

Understanding the fundamental differences between CPU and GPU architectures is key to appreciating CUDA's advantages:

- CPU (Central Processing Unit):
  - Designed for sequential processing.
  - Features powerful Arithmetic Logic Units (ALUs) with low latency.
  - Utilizes large hierarchical caches to optimize access to frequently used data.
  - Employs advanced control mechanisms, such as branch prediction and data forwarding, to minimize delays.
- GPU (Graphics Processing Unit):
  - Optimized for parallel processing.
  - Contains a large number of simpler, pipelined ALUs designed for high-throughput computations, despite having longer latency.
  - Relies on smaller caches to facilitate high memory throughput.
  - Uses simpler control mechanisms, enabling efficient context switching and handling many threads concurrently.

CUDA leverages these GPU characteristics to execute programs with a parallel-first approach, breaking down tasks into smaller, manageable pieces and processing them simultaneously. This approach leads to significant speedups compared to traditional CPU-only processing, making CUDA a pivotal technology for high-performance computing.

GPGPU programming paradigms

CUDA helped bring General Purpose computing on Graphics Processing Units (GPGPU). Initially designed to accelerate rendering and processing of graphics, GPUs have evolved into versatile processors capable of handling a wide range of computational tasks beyond graphics. This paradigm shift has allowed GPUs to be used for general-purpose computing, providing significant performance improvements for applications requiring high parallelism and computational power (the definition and introduction of GPGPU can be found on page 99).

The GPU (called a device) acts as a co-processor for the CPU (host):

- CPU (**host**):
  - *General-purpose* processor.
  - Handles diverse tasks, running an operating system, and executing a sequence of stored instructions.
  - Efficient for tasks requiring low-latency access to cached data.
- GPU (**device**):
  - Specialized for *high-throughput computation*.
  - Contains a large number of processing elements for parallel tasks.
  - Relies on dedicated, high-bandwidth memory.
  - Best for data-parallel tasks, hiding memory latency effectively.

The relationship of these two concepts:

- Both CPU and GPU have their own memory spaces.
- Optimal performance achieved through cooperation.
- CPU handles complex logic and serial tasks; GPU manages parallel processing.

About the architecture, the differences are:

- CPU (**host**):
  - Composed of control units, ALUs, cache, and DRAM.
  - Efficient for tasks requiring complex control and low-latency data access.
- GPU (**device**):
  - Made up of DRAM and a grid of processing units.
  - High-throughput design for parallel processing tasks.

GPGPU (General-Purpose computing on Graphics Processing Units) **programming paradigms are various approaches or models used to leverage the GPU for general-purpose computation**. These paradigms **allow programmers to write code that runs efficiently on GPUs**. The main paradigms are:

1. **Applications**. End-user programs that take advantage of GPU performance. Designed to take advantage of the GPU's processing power to accelerate performance (such as video processing software, scientific simulations, and machine learning models).
2. **Libraries**. Pre-built functions and routines optimized for GPUs. In other words, a collections of pre-written code and routines that are optimized to run on GPUs. Using these libraries, developers can avoid writing low-level GPU code. Examples of libraries are cuBLAS (for linear algebra), cuFFT (for Fast Fourier Transforms), and TensorFlow (for machine learning).
3. **Compiler Directives**. Special instructions embedded in the source code that guide the compiler on how to optimize and parallelize the code for GPU execution. OpenACC and OpenMP are commonly used directives.
4. **Programming Languages**. Languages or extensions of languages specifically designed for GPGPU programming. They provide the syntax and constructs needed to write code that executes on the GPU. Examples are CUDA and OpenCL (Open Computing Language).

Given an application code, in general:

- **Serial Parts run on the CPU** (host). These are the parts of the code that need to be executed sequentially and can benefit from the sophisticated control mechanisms of the CPU.
- **Computation-Intensive and Data-Parallel Parts offloaded to the GPU** (device). These parts can be executed in parallel, making use of the GPU's many cores for faster computation.

The flow inside the GPU can be described in three general steps:

1. **Copy Input Data from CPU Memory to GPU Memory**. The data is transferred over the PCI bus. This step is critical to prepare the data so that the GPU can access and process it.
2. **Load GPU Program and Execute, Caching Data on Chip for Performance**. The GPU program (kernel) is loaded and executed. Data is cached on the GPU chip to improve performance during execution. This step takes advantage of the parallel processing power of the GPU.
3. **Copy Results from GPU Memory to CPU Memory**. After calculation, the results are transferred back to the CPU memory via the PCI bus. This step is necessary to integrate the results into the larger application or for further processing on the CPU.

Common GPGPU issues and solutions

- Data Movement between CPU and GPU is the main Bottleneck.
  - **Low Bandwidth**. Data transfers between the CPU and GPU are relatively slow because they use the PCI Express (PCIe) bus, which has a bandwidth of about 12-14GB/s. This is much lower compared to the internal memory bandwidth of the CPU and GPU.
  - **Relatively High Latency**. Data transfer times can be significant, introducing delays that affect overall performance.
  - **Data Transfer can take more time than the actual computation**. In some cases, moving data between the CPU and GPU can take longer than the time it takes to perform the actual computations on the GPU. This is a critical point to consider when designing GPU-accelerated applications.
- Issues Porting CPU Applications to GPGPU.
  - **Ignoring Data Movement can destroy GPU Performance Benefits**. When converting CPU applications to run on a GPU, it's crucial to manage data transfers efficiently. Failing to do so can negate the performance advantages of using a GPU.
  - **Solutions to Hide/Automate Data Transfer**. Some programming techniques and tools can help hide or automate the data transfer process. These solutions can optimize performance by reducing the manual overhead of managing data movement.

In other words, efficient data management is the key to performance. In fact, the performance of GPGPU applications depends heavily on how well data is managed between the CPU and the GPU. Efficient data transfer strategies are essential to unleash the full computing power of GPUs.

7.2 CUDA Basics

CUDA, developed by NVIDIA, is a parallel computing platform and programming model that enables developers to harness the immense computational power of NVIDIA GPUs (Graphics Processing Units).

CUDA allows programmers to write code for GPUs using familiar programming languages like C, C++, and Fortran. This accessibility lowers the learning curve, enabling more developers to take advantage of GPU acceleration without requiring extensive knowledge of graphics programming.

The CUDA programming model evolves with the underlying hardware architecture, ensuring that developers can maximize performance gains from the latest GPU advancements. By writing a program for a single thread and instantiating it across many parallel threads, developers can achieve significant speedups for data-parallel tasks.

The most important function: the kernel

A **kernel** is a function that runs on the GPU. When a kernel is launched, **thousands of threads execute its code simultaneously**. The programmer specifies the number of threads, each acting independently on different data elements. This approach leverages Single Instruction, Multiple Data (SIMD) and Single Program, Multiple Data (SPMD) parallelism.

Example 1: comparison between C and CUDA C program

```c
void vsum(int* a, int* b, int* c) {
    int i;
    for (i=0; i<N; i++) {
        c[i] = a[i] + b[i];
    }
}

void main() {
    int va[N], vb[N], vc[N];
    ...
    vsum(va, vb, vc);
    ...
}

```

```c
__global__ void vsum(int* a, int* b, int* c) {
    int i = ... // get unique thread ID
    c[i] = a[i] + b[i];
}

void main() {
    int va[N], vb[N], vc[N];
    ...
    vsum<<<N, 1>>>(va, vb, vc);
    ...
}

```

In the CUDA version, the `vsum` function is defined as a kernel using the `__global__` keyword, and it is launched on the GPU with a specified number of threads. Each thread processes a different element of the input arrays independently, showcasing CUDA's ability to handle parallel tasks efficiently.

7.2.1 GPGPU Best Practices

NVIDIA provides a guide to help developers get the most out of NVIDIA CUDA GPUs:

CUDA C++ Best Practices Guide

Among the suggestions, NVIDIA explains a process for accelerating an application with NVIDIA GPUs called the *Cyclical Process for Accelerating Applications with NVIDIA GPUs*.

The **Cyclical Process for Accelerating Applications with NVIDIA GPU**s is a **systematic approach designed to optimize and enhance the performance of applications by leveraging the computational power of NVIDIA GPUs**. This process ensures that applications can take full advantage of the parallel processing capabilities of GPUs for improved performance and efficiency. The process consists of four main stages: Assess, Parallelize, Optimize, and Deploy.

1. **Assess**.
   (a) **Locate Bottlenecks**. Identify the parts of our application where performance is limited. This involves profiling our application to understand where most of the computation time is spent.
   (b) **Estimate Parallelization Benefits**. Determine the potential performance improvements from parallelizing the application. Evaluate how much of the workload can be efficiently offloaded to the GPU.
2. **Parallelize**.
   (a) **Apply Libraries, Compiler Directives, or CUDA**. Utilize GPU-optimized libraries (such as cuBLAS for linear algebra operations), apply compiler directives (like OpenACC) to guide the parallelization, or write custom CUDA kernels to parallelize the identified computational bottlenecks.
3. **Optimize**.
   (a) **Apply Optimizations**. Fine-tune our GPU code to improve performance. This can include optimizing memory access patterns to reduce latency, ensuring efficient data transfer between the CPU and GPU, and maximizing thread utilization to fully exploit the GPU's parallel architecture.
   (b) **Measure Performance Improvements**. Continuously profile and benchmark our application to monitor the impact of the optimizations and ensure that they lead to significant performance gains.

4. **Deploy**.
   (a) **Compare with Performance Estimations**. Verify that the performance improvements achieved through parallelization and optimization meet the initial estimations. This step ensures that the application performs as expected in a real-world environment.
   (b) **Move to Production**. Once satisfied with the performance, deploy the optimized application to production. This involves integrating the GPU-accelerated code into the main application and ensuring it runs efficiently in the production environment.

The process is depicted as a *continuous cycle*, indicating that **optimization and assessment are ongoing activities**. This **iterative approach ensures that the application remains optimized and continues to benefit from GPU acceleration over time**.

By following this cyclical process, developers can systematically **identify performance bottlenecks**, **efficiently parallelize** and **optimize** their **applications**, and **achieve significant performance improvements using NVIDIA GPUs**. This methodical approach helps ensure that the application takes full advantage of GPU capabilities, leading to faster, more efficient computing.

7.2.2 Compilation

The compilation of a CUDA file follows a structured process that involves both the CPU and GPU components.

1. `.cu File`. The source file containing CUDA kernels and the rest of the application code is usually saved with a `.cu` extension.
2. **CUDA Kernels**. The CUDA-specific code within the `.cu` file is processed by the NVIDIA CUDA Compiler (NVCC). This includes all the functions intended to run on the GPU.
2. **Rest of the Application**. The non-CUDA parts of the code, intended to run on the CPU, are processed by the host CPU compiler.
3. **NVCC Compiler**. NVCC compiles the CUDA kernels into CUDA object files, which are specific to the GPU.
3. **CPU Compiler**. The CPU compiler processes the remaining application code into CPU object files, which are specific to the CPU.
4. **Linker**. The linker combines the CUDA object files and CPU object files into a single executable that can run on both the CPU and GPU.

An example of a compilation is the command:

```
1 nvcc -arch=sm_70 -o hello-gpu 01-hello/01-hello-gpu.cu -run
```

That demonstrates how to compile and execute a CUDA file using NVCC. The `-arch=sm_70` flag specifies the architecture of the GPU, in this case, the Volta architecture (`sm_70`).

Useful Compilation Flags for NVCC:

Flags	Description
`-x`	Treat all input files as `.cu` files.
`-Xcompiler`	Pass a host compiler flag that is not supported by NVCC.
`-g`	Include host debugging symbols.
`-G`	Include device debugging symbols.
`-lineinfo`	Include line information with symbols.

Table 5: Some compilation flags.

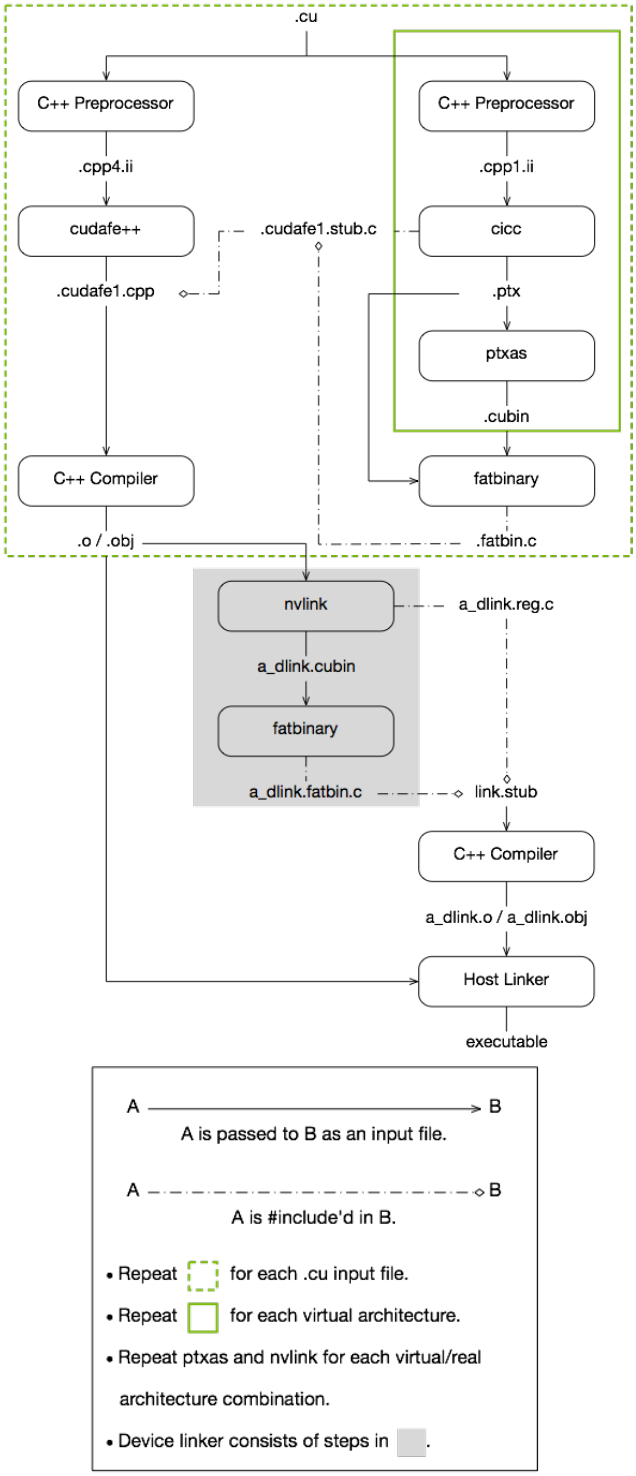


Figure 36: CUDA Compilation Trajectory, source: NVIDIA CUDA Compiler Driver NVCC

7.2.3 Debugging

Given the complexity and parallel nature of GPU programming, effective debugging tools are essential to ensure code correctness and optimize performance. NVIDIA provides a suite of **debugging tools** specifically designed for CUDA applications, helping developers identify and resolve issues in their code.

- `cuda-memcheck` is a comprehensive tool that detects memory-related errors in CUDA applications. It helps identify issues such as memory leaks, out-of-bounds accesses, and race conditions. Its capabilities:
  - **Memory Leaks**. Finds memory that is allocated but not freed, preventing unnecessary resource consumption.
  - **Memory Errors**. Detects accesses to invalid memory locations, which can cause unpredictable behavior.
  - **Race Conditions**. Identifies scenarios where multiple threads access shared data concurrently, potentially leading to incorrect results.
  - **Illegal Barriers**. Spots improper use of synchronization barriers in parallel code.
  - **Uninitialized Memory**. Flags the use of uninitialized memory, which can lead to unreliable outcomes.
- `cuda-gdb` is an extension of the GNU Debugger (GDB) tailored for debugging CUDA applications. It offers a familiar debugging environment for those already comfortable with GDB. Its capabilities:
  - **Setting Breakpoints**. Allows developers to pause execution at specific points in the code to inspect the state of the program.
  - **Inspecting Memory**. Enables examination of variables and memory contents to ensure they hold expected values.
  - **Stepping Through Code**. Provides the ability to step through code line by line, both on the CPU and GPU, to trace the execution flow.

With the debugging tools, there are also some profiling tools provided by NVIDIA.

CUDA profiling[9] is the process of measuring various aspects of a CUDA program's performance to **identify inefficient code segments**, **resource bottlenecks**, and **opportunities for optimization**. Profiling helps developers understand how their applications are utilizing GPU resources and make informed decisions to enhance performance.

- **NVPROF** is a command-line profiler provided by NVIDIA that **gives detailed timing information for each CUDA kernel and memory operation**. Its capabilities:

[9] **Profiling** is a **process** used in software development **to analyze and measure the performance characteristics of a program**. The goal of profiling is to identify parts of the code that are consuming the most resources or taking the most time, so that developers can optimize these areas and improve the overall performance of the application.

  - **Kernel Execution Time**. Measures the time taken by each CUDA kernel to execute.
  - **Memory Transfers**. Monitors the time spent on data transfers between the CPU and GPU.
  - **API Calls**. Tracks the duration of CUDA API calls.
  - **Occupancy**. Analyzes the utilization of GPU resources, helping to identify potential underutilization or over-subscription of resources.

  Developers use NVPROF to run their applications and generate detailed profiling reports, which can then be analyzed to pinpoint performance bottlenecks.

- **NVIDIA Visual Profiler (NVVP)** is a **graphical profiling tool** that provides a visual representation of the application's performance, making it easier to identify and address bottlenecks. Its capabilities:

  - **Timeline Visualization**. Displays a timeline of kernel executions, memory transfers, and API calls, allowing developers to see the sequence and overlap of events.
  - **Detailed Metrics**. Offers in-depth metrics on kernel performance, memory usage, and other critical aspects of CUDA applications.
  - **Optimization Suggestions**. Provides recommendations for optimizing code based on the profiling results.

  NVVP is particularly useful for visualizing complex interactions within CUDA applications and understanding how different parts of the code affect overall performance.

- **NSIGHT** is an integrated development environment that includes **advanced profiling and debugging features** for CUDA applications. Its capabilities:

  - **Advanced Profiling**. Combines the features of NVPROF and NVVP, offering a comprehensive set of profiling tools within a single interface.
  - **Interactive Analysis**. Allows developers to interactively analyze performance data and make real-time adjustments to their code.
  - **Unified Development**. Integrates debugging and profiling, providing a seamless environment for developing and optimizing CUDA applications.

  NSIGHT is ideal for developers who need a powerful, all-in-one tool for debugging and profiling their CUDA applications.

- **Third-Party Profiling Tools**:

  - **TAU** (Tuning and Analysis Utilities). A performance analysis tool that can be used to profile CUDA applications along with other types of programs. **TAU provides detailed performance data and visualization capabilities**.

  - **VampirTrace**. **Captures performance data and visualizes it to help optimize parallel applications**. It supports a range of profiling features for CUDA and other programming models.

Another powerful profiling system is Nsight. **Nsight Systems** offers both a **command-line profiler and a graphical user interface (GUI)**, providing comprehensive insights into the execution of CUDA applications. It helps identify performance bottlenecks, understand GPU utilization, and improve overall application efficiency.

- **Nsight Command-Line Profiler**. `nsys` can be used to profile an accelerated application by launching it and gathering detailed statistics about its performance. The information reported are:
  - **GPU Activity**. Insights into how the GPU is utilized during the application's execution.
  - **CUDA API Calls**. Details on the usage of CUDA API functions.
  - **Memory Activity**. Information about memory operations, including transfers between CPU and GPU.

  An example command is:

```
nsys profile --stats=true -o vector-add-no-prefetch-report ./
    vector-add-no-prefetch
```

  This command profiles the application `vector-add-no-prefetch` and generates a report with detailed statistics.

- **Nsight Systems GUI**. Provides a visual representation of the profiling data collected by `nsys`, making it easier to analyze and interpret the results. Its capabilities:
  - **Timeline View**. Displays a comprehensive timeline of CPU and GPU activities, showing how different tasks are executed over time.
  - **Detailed Metrics**. Offers in-depth metrics on kernel performance, memory usage, and other critical aspects.
  - **Interactive Analysis**. Allows developers to zoom into specific regions of the timeline and examine detailed activities.

7.2.4 CUDA Kernel

Launching a CUDA kernel involves **defining a function that will run on the GPU and then executing this function from the host (CPU) code**.

1. **Defining functions**.
   - CPU Function:

```
void CPUFunction() {
    printf("This function is defined to run on the CPU.");
}
```

     This function runs on the CPU and prints a message.
   - GPU Function:

```
__global__ void GPUFunction() {
    printf("This function is defined to run on the GPU.");
}
```

     This function is defined to run on the GPU, indicated by the `__global__` qualifier, and it prints a message when executed.

2. **Launching the Kernel**. In the main function, we launch the GPU function using a special syntax:

```
int main() {
    CPUFunction();

    // Launch the GPU kernel
    // with 1 block and 1 thread
    GPUFunction<<<1, 1>>>();
    cudaDeviceSynchronize();
}

```

   - The `CPUFunction()` is called normally, as it's a CPU function.
   - The `GPUFunction<<<1, 1>>>();` syntax launches the GPU kernel with a specified execution configuration: `<<<1, 1>>>` means 1 block and 1 thread per block.
   - `cudaDeviceSynchronize()` is called to ensure that the CPU waits for the GPU to finish executing the kernel before continuing.

Just like with variables (table 4, page 131), CUDA provides specific keywords that define the scope and lifetime of functions.

Function Declaration	Executed on	Callable from
`__device__ type DeviceFunc()`	Device	Device
`__global__ void KernelFunc()`	Device	Host
`__host__ type HostFunc()`	Host	Host

Table 6: CUDA function qualifiers.

How can we customize the kernel?

Kernel configuration in CUDA involves specifying **how many blocks and threads will be used** to execute a kernel on the GPU. The key parameters are:

- **Number of Blocks**. Specifies **how many blocks** of threads will be launched on the GPU.
- **Number of Threads per Block**. Specifies **how many threads** will be **in each block**.

For a *1D hierarchy*, we use simple integer numbers to specify these values because a 1D grid or block can be represented by a single dimension:

```
// Example of launching a kernel with 1 block and 256 threads in 1D
KernelFunc<<<1, 256>>>();
```

For *higher dimensions* (2D or 3D), CUDA uses the `dim3` **type to represent the grid and block dimensions**. `dim3` is a CUDA-specific type that can hold three unsigned integer values, representing the dimensions in the x, y, and z directions. This provides a flexible way to configure more complex thread hierarchies:

```
// Example of launching a kernel with a 2D grid and blocks
dim3 gridDim(16, 16); // 16 blocks in x and y dimensions
dim3 blockDim(16, 16); // 16 threads per block in x and y
    dimensions
KernelFunc<<<gridDim, blockDim>>>();
```

CPU-GPU Synchronization

In CUDA, **kernel executions are asynchronous**, meaning that the **CPU can continue to execute other instructions while the GPU is running the kernel**. However, there are times when synchronization is required to ensure that the CPU is waiting for the GPU to complete its tasks.

- **Asynchronous Kernel Launch**. When a kernel is launched, the CPU can proceed with subsequent instructions without waiting for the GPU to finish.

```
KernelFunc<<<1, 256>>>();
// The CPU immediately continues executing the next
    instructions
```

- **Synchronizing CPU and GPU**. To synchronize, CUDA provides runtime functions such as `cudaDeviceSynchronize()`, which ensures that the CPU waits until the GPU has completed all preceding tasks.

```
KernelFunc<<<1, 256>>>();
// Wait for the GPU to finish
cudaDeviceSynchronize();
```

7.3 Execution Model

The **CUDA execution model** **defines how parallel computing tasks are organized and executed on NVIDIA GPUs**. It consists of both software and hardware components that work together to efficiently process large amounts of data in parallel.

Software Hierarchy

- **Grid**. A grid is a **collection of thread blocks that execute a given kernel function**. Each grid is associated with a specific kernel launch. Grids can be one-dimensional, two-dimensional, or three-dimensional, depending on the problem's complexity.
- **Thread Block**. A thread block is a **group of threads that execute together and can cooperate by sharing data through shared memory**. Each block is independent, allowing the scheduler to execute blocks in any order. Thread blocks can also be organized in one, two, or three dimensions.
- **Thread**. The **smallest unit of execution in CUDA**. Each thread executes the **same code but operates on different data elements**. Threads within a block can communicate and synchronize with each other, enabling collaborative computation.

Hardware Hierarchy

- **GPU**. The **physical hardware that executes the parallel tasks**. A GPU consists of **multiple Streaming Multiprocessors (SMs)**.
- **Streaming Multiprocessor (SM)**. Each SM can **execute multiple threads concurrently**. Thread blocks are assigned to SMs for execution. SMs manage and execute warps of threads.
- **GPU Core**. **Individual cores within the SMs that perform the actual computation**. Cores **execute threads in groups called warps**, typically comprising 32 threads.

What happens if the kernel startup doesn't match the input dataset?

Execution configuration mismatches occur when the number of threads configured for a kernel launch does not perfectly match the size of the dataset being processed. This can lead to *inefficiencies* or the need for *additional handling in the kernel code*.

- **Optimal Thread Configuration**: **multiple of 32 threads**. It's generally desirable to configure blocks with a multiple of 32 threads for performance reasons. This is because GPUs execute threads in groups of 32, known as warps. Ensuring the number of threads per block is a multiple of 32 can prevent under-utilization of GPU resources.

  As the warp dimension changes, the optimal thread configuration should also change.

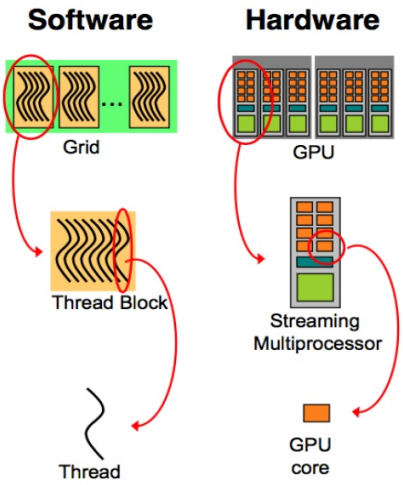


Figure 37: Graphical representation of the CUDA execution model.

- **Dealing with Non-Multiple of 32 threads**. If the dataset size (e.g., `N=1000`) is not a multiple of 32, we can create more than 1000 threads and use an extra check within the kernel to handle the excess threads.

  1. We pass the dataset size `N` as an argument to the kernel.
  2. Inside the kernel, we check if the thread ID is greater than or equal to `N`. If it is, those threads should not perform any operations.

```
__global__ void kernelFunction(float* data, int N) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < N) {
        // Perform operations only if thread ID
        // is within the dataset size
        data[tid] = data[tid] * 2.0f;
    }
}

```

- **Insufficient threads**. In some cases, there may be fewer threads than data elements to process, either for performance reasons or because the dataset is very large.

  In this case, the solution is using a **Stride Factor**. We can use a stride factor **corresponding to the total number of threads**. The stride factor is calculated as `gridDim.x * blockDim.x`, representing the **total number of threads in the grid**. Each thread processes multiple elements by iterating with a step size equal to the stride.

```
__global__ void kernelFunctionWithStride(float* data, int N) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = gridDim.x * blockDim.x;
    for (int i = tid; i < N; i += stride) {
        data[i] = data[i] * 2.0f;
    }
}

```

References

[1] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities, reprinted from the afips conference proceedings, vol. 30 (atlantic city, nj, apr. 18–20), afips press, reston, va., 1967, pp. 483–485, when dr. amdahl was at international business machines corporation, sunnyvale, california. *IEEE Solid-State Circuits Society Newsletter*, 12(3):19–20, 2007.

[2] Guy E. Blelloch, Laxman Dhulipala and Yihan Sun. Introduction to parallel algorithms. https://www.cs.cmu.edu/~guyb/paralg/paralg/parallel.pdf, 2024. [Accessed 22-10-2024].

[3] Ferrandi Fabrizio. Parallel computing. Slides from the HPC-E master's degree course on Politecnico di Milano, 2024.

[4] John L Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[5] Johnston Hans. OpenMP by Example. https://people.math.umass.edu/~johnston/PHI_WG_2014/OpenMPSlides_tamu_sc.pdf, 2024. [Accessed 23-10-2024].

[6] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing*, volume 110. Benjamin/Cummings Redwood City, CA, 1994.

[7] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE micro*, 28(2):39–55, 2008.

[8] M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. ITPro collection. Elsevier Science, 2012.

[9] Microsoft. The critical directive. https://learn.microsoft.com/en-us/cpp/parallel/openmp/a-examples?view=msvc-170#a5-the-critical-directive, 2024. [Accessed 29-10-2024].

[10] M. Nemirovsky and D. Tullsen. *Multithreading Architecture*. Synthesis Lectures on Computer Architecture. Springer International Publishing, 2022.

[11] University of Michigan, EECS Department, Prof. Ronald Dreslinski. Lecture 5, Synchronization I - EECS 570, 2024. [Accessed 24-11-2024].

[12] Wikipedia. Cache (computing) - Wikipedia. https://en.wikipedia.org/wiki/Cache_(computing). [Accessed 20-10-2024].

Index