Module 2

GPU programming, Programming hybrid systems, MIMD systems, GPUs, Performance – Speedup and efficiency in MIMD systems, Amdahl's law, Scalability in MIMD systems, Taking timings of MIMD programs, GPU performance.

1.1 GPU Programming

GPUs are not standalone processors and typically don't run their own operating system or access system services like storage. Therefore, programming a GPU also requires writing code for the CPU host, which manages tasks like memory allocation and data initialization. The CPU and GPU usually have separate memory spaces, so the host code must handle memory transfers and execution control. After launching the GPU program, the host is also responsible for collecting and displaying results. This makes GPU programming a form of heterogeneous programming, as it involves coordinating two different types of processors.

A GPU consists of one or more processors, each capable of running hundreds or thousands of threads. While all processors share a large global memory, each processor also has a smaller, faster memory accessible only to its own threads, functioning like a programmer-managed cache.

- Threads on a GPU processor are grouped into sets, where each group follows the SIMD (Single Instruction, Multiple Data) model.
- Threads in different groups can operate independently, executing different instructions concurrently.
- Within a SIMD group, threads may not execute the same instruction at the exact same time, but all must complete the current instruction before any can proceed to the next.
- When executing branch instructions, some threads in a group may need to idle, depending on conditional execution paths.
- For instance, if 32 threads are in a SIMD group and each has a variable rank_in_gp from 0 to 31, their execution may vary based on this variable, potentially causing divergence and idle threads.

Suppose also that the threads are executing the following code:

```
// Thread private variables
int rank_in_gp, my_x;
...
if (rank_in_gp < 16)
   my_x += 1;
else
   my_x -= 1;
```

When the condition is checked, threads with rank < 16 will run the first assignment while the others stay idle. Once that finishes, the roles switch and threads with rank ≥ 16 run the second assignment, while the rest wait. This back-and-forth leads to poor efficiency, so programmers should aim to reduce branching within SIMD groups to make better use of resources.

In GPU programming, thread scheduling is handled by hardware rather than software, unlike CPUs. The GPU's hardware scheduler introduces minimal overhead and executes an instruction only when all threads in a SIMD group are ready.

For example, each thread needs to have the variable *rank_in_gp* loaded into a register before the conditional test is executed. To efficiently utilize the hardware, many SIMD groups are created. This allows the scheduler to idle groups that are waiting (for data or previous instructions) and switch to another SIMD group that is ready to run.

1.2 Programming hybrid systems

Clusters of multicore processors can be programmed using a combination of shared-memory APIs within nodes and distributed-memory APIs between nodes. However, this hybrid approach is typically used only for performance-critical applications due to its complexity. In most cases, a single distributed-memory API is used for both intra- and inter-node communication to simplify development.

1.3 Input and Output

1.3.1 MIMD systems

We've mostly avoided discussing input and output because parallel I/O is a complex topic that could fill an entire book. Most programs we write do minimal I/O, which can be handled with standard C functions like printf, fprintf, scanf, and fscanf. However, these functions are part of the serial C standard and don't define behavior when used by multiple processes. Threads within a single process share stdin, stdout, and stderr, but concurrent access by threads can lead to unpredictable, nondeterministic results.

When *printf* is called from multiple processes or threads, we usually expect the output to appear on the console of the system that started the program. While most systems behave this way, it's not guaranteed—some may allow only one process to access *stdout* or *stderr*, or *none at all*. With *scanf*, it's unclear whether input should be shared or limited; most systems allow at least one process, typically process 0, to call *scanf*, and many allow multiple threads to use it. However, some systems block scanf entirely for processes. When multiple processes or threads access *stdin*, *stdout*, or *stderr*, the results can be nondeterministic. Output may appear in a different order or get interleaved across processes/threads. Similarly, input might vary each time the program runs, even with the same input data.

To simplify I/O in parallel programs, we'll follow these rules:

- Only process 0 (or thread 0) will read from stdin.
- All processes/threads can write to stdout and stderr, but usually only one will write to avoid jumbled output—except during debugging.
- Each process/thread will use its own file for input/output; no file (except stdin, stdout, or stderr) will be shared.
- Debug output should always include the process/thread rank or ID.

1.3.2 GPUs

In most GPU programs, the host code handles all I/O using standard C functions, as only one host thread runs. An exception occurs during debugging, when GPU threads may write to stdout, though the output order is nondeterministic. However, GPU threads cannot access stderr, stdin, or secondary storage.

1.4 Performance

The main goal of parallel programming is typically improved performance, so it's important to understand what to expect and how to evaluate it. This section focuses on performance in homogeneous MIMD systems, where all cores share the same architecture; GPU performance will be discussed separately.

1.4.1 Speedup and efficiency in MIMD systems

The ideal case for a parallel program is when the work is evenly divided among p cores without adding extra overhead. In this case, the program can run p times faster than the serial version on

a single core. If T_{serial} is the serial run-time and $T_{parallel}$ is the parallel run-time, the best possible outcome is $T_{parallel} = T_{serial} / p$. This is known as **linear speedup**.

Table 1: Speedups and efficiencies of a parallel program.

p	1	2	4	8	16
S	1.0	1.9	3.6	6.5	10.8
$E = S/p$	1.0	0.95	0.90	0.81	0.68

- In reality, parallel programs rarely achieve perfect linear speedup due to overhead.
- Shared-memory programs often have critical sections that require mutexes, adding extra function calls and forcing some parts to run serially.
- Distributed-memory programs typically need to send data over a network, which is much slower than accessing local memory.
- These overheads don't exist in serial programs, so parallel programs usually fall short of linear speedup.
- As the number of threads or processes increases, the overhead generally grows—for example, more threads may compete for a critical section, and more processes may need to exchange data.

Speedup (S) is defined as the ratio of serial execution time to parallel execution time:

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}},$$

Efficiency (E) is the speedup divided by the number of processors (**p**):

$$E = \frac{S}{p} = \frac{T_{\text{serial}}}{p \cdot T_{\text{parallel}}}$$

If a program has **linear speedup** $T_{parallel} = T_{serial} / p$, then S = p, and hence efficiency E = 1.

As the number of processors **p** increases, **parallel overhead** increases (e.g., communication, synchronization).

This causes **S** to become less than **p**, and thus E=S/p decreases with higher **p**.

The term S/p or **efficiency** reflects how well the processors are being utilized in the parallel execution.

Table 1 illustrates how both **S** and **E = S/p** decline as more processors are added, indicating diminishing returns.

If the serial run-time is measured on the same type of core as the parallel system, efficiency represents how well the parallel cores are used. It shows the average fraction of time each core spends doing useful work on the problem. The rest of the time accounts for parallel overhead. This relationship is clear when we multiply efficiency by the parallel run-time.

$$E \cdot T_{\text{parallel}} = \frac{T_{\text{serial}}}{p \cdot T_{\text{parallel}}} \cdot T_{\text{parallel}} = \frac{T_{\text{serial}}}{p}.$$

For example, suppose we have $T_{serial} = 24$ ms, $p = 8$, and $T_{parallel} = 4$ ms. Then

$$E = \frac{24}{8 \cdot 4} = \frac{3}{4},$$

and, on average, each process/thread spends $3/4 \cdot 4 = 3$ ms on solving the original problem, and $4 - 3 = 1$ ms in parallel overhead. Many parallel programs are developed by explicitly dividing the work of the serial program among the processes/threads and adding in the necessary "parallel overhead," such as mutual exclusion or communication. Therefore if $T_{overhead}$ denotes this parallel overhead, it's often the case that

$$T_{\text{parallel}} = T_{\text{serial}}/p + T_{\text{overhead}}.$$

When this formula applies, parallel efficiency is the fraction of time spent on the actual problem: $\mathbf{T_{serial}/p}$ versus the total parallel time, $\mathbf{T_{parallel}}$, which includes overhead $\mathbf{T_{overhead}}$. We've seen that **$T_{parallel}$, speedup S, and efficiency E depend on the number of threads or processes p**. They also vary with problem size. For instance, halving or doubling the problem size changes the values of S and E, as shown in Tables 1 and 2 and Figures 2.1 and 2.2.

Table 2: Speedups and efficiencies of parallel program on different problem sizes.

	p	1	2	4	8	16
Half	*S*	1.0	1.9	3.1	4.8	6.2
	E	1.0	0.95	0.78	0.60	0.39
Original	*S*	1.0	1.9	3.6	6.5	10.8
	E	1.0	0.95	0.90	0.81	0.68
Double	*S*	1.0	1.9	3.9	7.5	14.2
	E	1.0	0.95	0.98	0.94	0.89

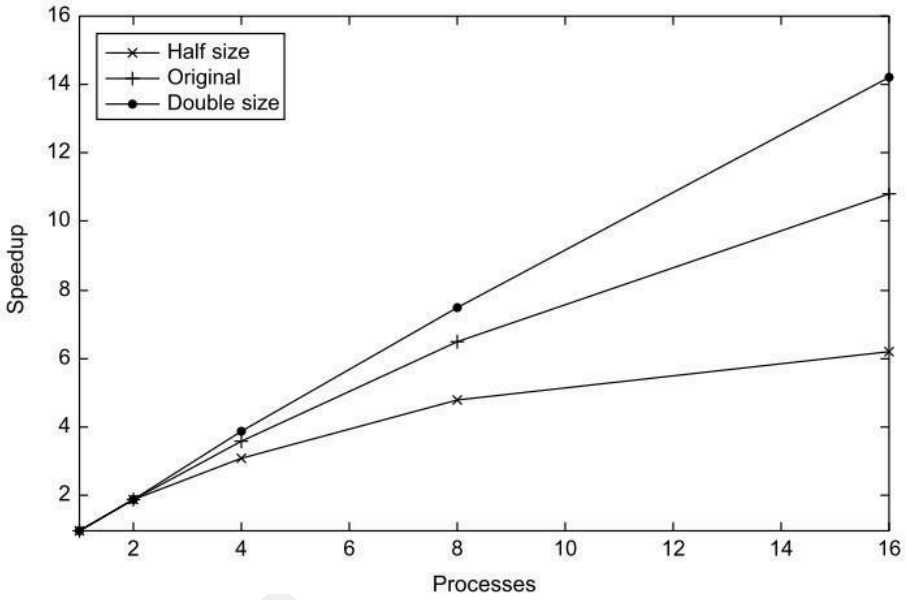

Fig 2.1: Speedups of parallel program on different problem sizes.

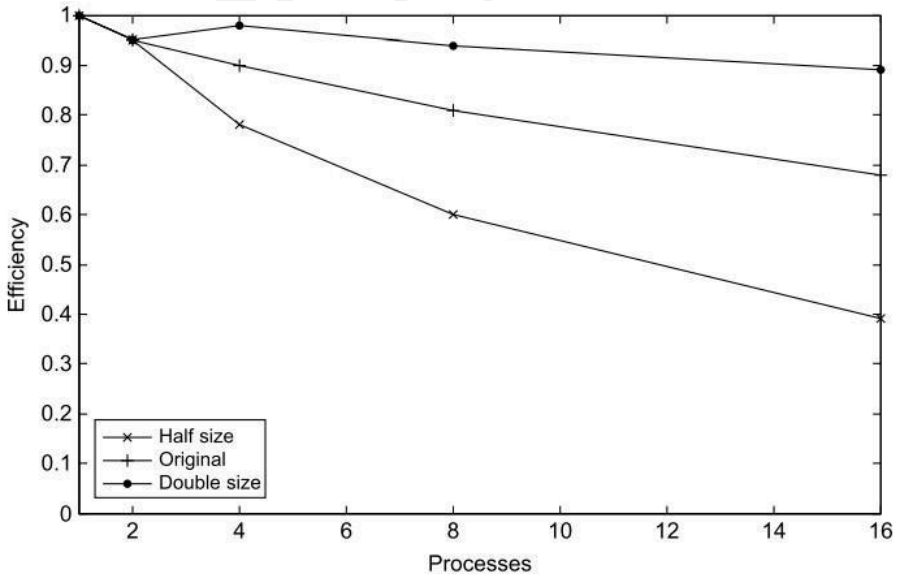

Fig 2.2: Efficiencies of parallel program on different problem sizes

In many parallel programs, as the problem size increases and the number of threads p remains fixed, T_{serial} (time for solving the original problem) grows faster than the parallel overhead $T_{overhead}$. This results in better efficiency for larger problems.

When reporting speedup and efficiency, there's debate on which T_{serial} to use—some prefer the time of the fastest algorithm on the fastest system, while others use the serial version of the same algorithm run on a single processor of the parallel system. Since efficiency reflects core utilization on the actual system, most use the second approach. For consistency, we'll also use the serial version of the same program on a single core of the parallel system.

1.4.2 Amdahl's law

Amdahl's law states that if a portion of a program remains serial, the overall speedup is limited, no matter how many cores are used. For example, if 90% of a program is perfectly parallelized,

$$S = \frac{T_{\text{serial}}}{0.9 \times T_{\text{serial}}/p + 0.1 \times T_{\text{serial}}} = \frac{20}{18/p + 2}.$$

then that part runs in 18/p seconds when T_{serial}= 20 seconds. The remaining 10% is serial and takes $0.1 \times T_{serial}$= 2 seconds. So, the total parallel time is 18/p + 2, showing that the unoptimized part limits the speedup.

$$T_{\text{parallel}} = 0.9 \times T_{\text{serial}}/p + 0.1 \times T_{\text{serial}} = 18/p + 2,$$

and speedup will be

Now as p gets larger and larger, 0.9 × Tserial/p = 18/p gets closer and closer to 0, so the total parallel run-time can't be smaller than 0.1 × Tserial = 2. That is, the denominator in S can't be smaller than 0.1 × Tserial = 2. The fraction S must therefore satisfy the inequality

$$S \leq \frac{T_{\text{serial}}}{0.1 \times T_{\text{serial}}} = \frac{20}{2} = 10.$$

That is, S ≤ 10. This is saying that even though we've done a perfect job in parallelizing 90% of the program, and even if we have, say, 1000 cores, we'll never get a speedup better than 10.

Amdahl's Law explains that if a part of a program, say a fraction **r**, cannot be parallelized, then the maximum speedup we can achieve is limited to **1/r**. For example, if 10% of the program is serial (**r = 0.1**), the best speedup we can get is **10**, no matter how many cores we use. Even if

only 1% is serial (**r = 0.01**), the maximum possible speedup is **100**. So, unless nearly the entire program is parallelized, adding more cores won't lead to huge speedups.

There are several reasons not to be too worried by Amdahl's law

- Amdahl's Law doesn't consider problem size; as problem size increases, the serial portion may shrink — this idea is formalized in Gustafson's Law.
- Many scientific and engineering applications still achieve very large speedups on distributed-memory systems.
- Small speedups (like 5 or 10) can still be valuable, especially when the effort to parallelize is minimal.

In practice, even modest improvements can justify using parallel computing for better performance.

1.4.3 Scalability in MIMD systems

The term "scalable" is often used informally, generally meaning that a program can benefit from increased computing power, such as more cores. In the context of MIMD parallel program performance, scalability has a more precise meaning. If a program maintains the same efficiency E when both the number of processes or threads and the problem size are increased proportionally, it is considered scalable. This means the program continues to perform efficiently as system resources grow. Scalability, in this sense, measures how well a program adapts to larger systems and workloads.

As an example, suppose that Tserial = n, where the units of Tserial are in microseconds, and n is also the problem size. Also suppose that Tparallel = n/p + 1. Then

$$E = \frac{n}{p(n/p+1)} = \frac{n}{n+p}.$$

To test a program's scalability, we increase the number of processes or threads by a factor of k and determine the factor x by which the problem size must grow to maintain the same efficiency E. With kp processes and a problem size of xn, we aim to find x such that efficiency remains unchanged.

$$E = \frac{n}{n+p} = \frac{xn}{xn+kp}.$$

Well, if x = k, there will be a common factor of k in the denominator xn + kp = kn + kp = k(n + p), and we can reduce the fraction to get

$$\frac{xn}{xn+kp} = \frac{kn}{k(n+p)} = \frac{n}{n+p}.$$

In other words, if we scale the problem size proportionally with the number of processes or threads, the efficiency stays the same, indicating that the program is scalable.

There are two special types of scalability: **strong** and **weak** scalability.

- A program is **strongly scalable** if efficiency remains constant as we increase the number of processes or threads, without changing the problem size.
- A program is **weakly scalable** if efficiency stays the same when both the problem size and the number of processes or threads are increased at the same rate.
- In our example, the program is considered **weakly scalable**.

1.4.4 Taking timings of MIMD programs

To find **Tserial** and **Tparallel**, we typically measure how long the program takes to run in serial and parallel modes. While the exact method can vary depending on the parallel API used, some general practices can help simplify the process.

Firstly,

1. Timing in parallel programming is done for two main purposes: debugging during development and measuring overall performance after development.
2. During development, timings help identify issues such as excessive message-waiting time in distributed-memory programs, which could indicate design or implementation problems.
3. This stage requires detailed timings, breaking down how much time is spent in specific sections of the code.
4. After development, the focus shifts to evaluating the program's performance, typically using a single overall timing value.
5. The method and level of detail in timing differ significantly between development and performance evaluation phases.

Secondly,

1. In performance analysis, we typically focus on the time taken by a specific part of the program, like sorting, rather than the entire execution time.
2. Therefore, tools like the Unix time command, which measure total program duration, are not suitable for such targeted measurements.

Third,

1. We generally don't rely on "CPU time" reported by functions like clock in C, as it only measures the active execution time of the program.
2. It includes time spent in user-written code, standard library functions (e.g., pow, sin), and OS-level calls (e.g., printf, scanf).
3. However, it excludes idle time, which can be significant in parallel programs, making CPU time less reliable for performance analysis.

In distributed-memory programs, if a process waits for a matching send after calling a receive, the operating system may put it to sleep, and this idle time won't be counted as CPU time. However, this waiting period should be included in the total run-time, as ignoring it would give a misleading view of the program's actual performance.

When articles report the run-time of a parallel program, they typically refer to the "wall clock" time—meaning the actual elapsed time from the start to the end of execution. Although users can't watch the execution directly, they can measure this time by adding timing code at the beginning and end of the relevant sections.

```
double start, finish;
. . .
start = Get_current_time();
/* Code that we want to time */
. . .
finish = Get_current_time();
printf("The elapsed time = %e seconds\n", finish-start);
```

The function Get_current_time() is a placeholder meant to return the number of seconds elapsed since a fixed reference point. The actual function used depends on the API—MPI provides MPI_Wtime(), and OpenMP offers omp_get_wtime(), both of which return wall clock time. However, the resolution of these timer functions can vary; resolution refers to the smallest time

interval the timer can measure. Some timers operate in milliseconds, which may be too coarse for measuring very short events, especially when modern instructions execute in nanoseconds. Therefore, programmers should check the resolution either through API-provided functions or documentation to ensure accurate timing.

When timing parallel programs, it's important to consider that multiple processes or threads are running the timed code. This often results in *p* different elapsed times, one for each process or thread.

```
private double start, finish;
. . .
start = Get_current_time();
/* Code that we want to time */
. . .
finish = Get_current_time();
printf("The elapsed time = %e seconds\n", finish-start);
```

What we usually want is the total time from when the first process/thread starts to when the last one finishes. Since clocks across nodes may not be perfectly synchronized, we typically use an approximate method to estimate this elapsed time.

```
shared double global_elapsed;
private double my_start, my_finish, my_elapsed;
. . .
/* Synchronize all processes/threads */
Barrier();
my_start = Get_current_time();

/* Code that we want to time */
. . .

my_finish = Get_current_time();
my_elapsed = my_finish - my_start;

/* Find the max across all processes/threads */
global_elapsed = Global_max(my_elapsed);
if (my_rank == 0)
   printf("The elapsed time = %e seconds\n",
          global_elapsed);
```

First, a barrier function is used to roughly synchronize all processes or threads. Each one then records its own elapsed time, and a global maximum function finds the longest time, which is printed by process/thread 0.

Timing results can vary across multiple runs, even with the same inputs and system. Since external factors are unlikely to make a program run faster than its best time, we usually report the minimum elapsed time instead of the mean or median.

Running multiple threads per core increases timing variability and adds overhead due to extra scheduling. As a result, we typically use only one thread per core and exclude I/O operations from reported run-times.

1.4.5 GPU Performance

When analyzing MIMD performance, we often compare how a parallel program performs relative to its serial counterpart. This same idea is sometimes applied to GPU programs, with reported speedups showing how much faster a GPU implementation is compared to a serial or even a MIMD version. However, there's a key difference: MIMD systems usually involve cores that are similar to those used in the serial baseline, whereas GPUs are built on inherently parallel cores that function differently. Therefore, directly comparing GPU and CPU performance may not provide meaningful insights.

As a result, metrics like efficiency—which are useful for MIMD programs—aren't commonly applied to GPU performance. Similarly, the concept of linear speedup doesn't quite work with GPU programs when compared against serial CPU versions. In the case of MIMD programs, scalability is formally defined based on efficiency and performance growth with added cores. But for GPUs, scalability is discussed more informally: if increasing the GPU's size leads to better performance, the program is considered scalable.

Amdahl's Law, which sets a theoretical upper bound on speedup based on the non-parallelizable fraction of a program, can still be applied to GPU programs—*if* the serial portion of the code runs on a regular CPU. For example, if 10% of the program cannot be parallelized and is handled by a CPU, then the speedup will be limited to less than 10x, regardless of how powerful the GPU is. However, like in MIMD systems, the size of the serial portion often depends on the problem size; as the problem grows, the impact of that portion may shrink, allowing for better speedups.

It's also worth noting that GPU programs often demonstrate very large speedups in practice. Still, even modest speedups can be valuable depending on the application. When it comes to

timing

GPU programs, many of the same principles used in MIMD timing apply. Because a GPU program typically starts and ends on a conventional CPU, a CPU-based timer can be used to measure the full duration of GPU execution—by starting it before the GPU kernel begins and stopping it after completion.

For more complex setups, such as when multiple CPU-GPU pairs are involved, careful timing strategies are necessary. However, those advanced configurations are outside the current scope. If you're only interested in measuring a specific part of GPU execution, it's best to use timing tools provided by the GPU's API rather than relying on the CPU timer.