

Module – 02
GPU programming

GPU programming

GPU programming is fundamentally heterogeneous programming, involving the coordination of two distinct types of processors: the CPU “host” system and the GPU itself.

This is because GPUs are typically not standalone processors; they don’t run their own operating systems or directly access secondary storage.

Here’s a breakdown of the key aspects of GPU programming:

1. CPU Host’s Role:

The CPU host is responsible for the overall control and coordination.

It allocates and initializes memory on both the CPU (host memory) and the GPU (device memory), which are usually separate.

It initiates the execution of programs on the GPU.

It handles the output and retrieval of results from the GPU program.

2. GPU Architecture:

A GPU contains one or more processors.

Each of these processors is designed to run hundreds or thousands of threads concurrently.

Processors typically share a large block of global memory.

Critically, each individual processor also has a small block of much faster memory, often referred to as a

programmer-managed cache. This faster memory is only accessible by threads running on that specific processor.

3. Thread Organization and Execution (SIMD Groups):

Threads running on a GPU processor are typically organized into groups, often called SIMD (Single Instruction, Multiple Data) groups.

Within a SIMD group: Threads generally operate under the SIMD model. While they may not execute in strict “lockstep” (i.e., not all execute the exact same instruction at the precise same time), no thread in the group will move to the next instruction until all threads in the group have completed the current instruction.

Across different SIMD groups: Threads in different groups can run largely independently.

4. Branching and Idling Threads:

A significant challenge in SIMD execution is branching (e.g., if-else statements) where threads within the same group take different execution paths.

Example:

```
C
// Thread private variables
int rank_in_gp, my_x;

my_x += 1;

else
my_x -= 1;
```

If there are 32 threads in a SIMD group, threads with `rank_in_gp < 16` will execute the `my_x += 1` branch, while threads with `rank_in_gp >= 16` will be idled. Once the first set of threads finishes, the roles are reversed: the `rank_in_gp < 16` threads are idled, and the `rank_in_gp >= 16` threads execute `my_x -= 1`.

Efficiency Impact: Idling half of the threads for portions of execution is inefficient. Therefore, GPU programmers must strive to minimize branching within SIMD groups.

5. Hardware Scheduling and Maximizing Resource Use:

- GPUs utilize a hardware scheduler (unlike CPUs, which typically use software schedulers). This hardware scheduler operates with very low overhead.
- The scheduler's goal is to execute an instruction when all threads in a SIMD group are ready.
- To maximize hardware utilization, it's common practice to create a large number of SIMD groups.
 - This provides the scheduler with more options: if some SIMD groups are not ready (e.g., waiting for data from memory or a previous instruction), they can be idled, and the scheduler can pick another ready group to execute, thereby keeping the GPU busy.

Programming hybrid systems

When programming systems composed of clusters of multicore processors, a common approach for achieving high performance is to combine different programming models:

- **Shared-memory API:** Used for communication and data sharing within a single node (i.e., among the cores of a single multicore processor).
- **Distributed-memory API:** Used for communication and data exchange between different nodes in the cluster.

This combination is often referred to as a "hybrid" API programming model.

Why use a hybrid API?

The primary motivation for employing a hybrid API is to achieve the highest possible levels of performance. By utilizing the shared memory model within a node, programs can exploit faster communication paths (e.g. directly accessing shared memory) compared to the overheads of inter-node communication.

Challenges of Hybrid API Programming:

Despite the performance benefits, hybrid API programming introduces significant complexity, making program development much more difficult. This increased complexity arises from:

- Managing two distinct programming models: Developers must understand and effectively integrate both

shared-memory and distributed-memory paradigms.

- Data placement and movement: Carefully orchestrating data movement between shared memory within a node and distributed memory across nodes becomes crucial for performance.
- Synchronization: Ensuring correct synchronization both within and across nodes adds layers of complexity.

Alternative Approach:

- Given the challenges, such systems are often programmed using a single, unified distributed-memory API for both inter-node and intra-node communication.
- While this might not always yield the absolute peak performance achievable with a finely-tuned hybrid approach, it significantly reduces program development complexity by providing a consistent communication model across the entire cluster.
- In essence, the choice between a hybrid API and a purely distributed-memory API for clusters of multicore processors is a trade-off between maximizing performance and managing development complexity.
- Hybrid APIs are typically reserved for applications where squeezing out every last bit of performance is paramount.

MIMD systems

MIMD Systems: Navigating Input/Output Challenges

When discussing MIMD (Multiple Instruction, Multiple Data) systems, the topic of input/output (I/O) often presents unique challenges compared to serial programming.

While parallel I/O is a complex field in itself, the vast majority of parallel programs for these systems perform relatively little I/O, often relying on standard C functions like `printf`, `fprintf`, `scanf`, and `fscanf`.

However, even this limited use can lead to significant issues due to the inherent nature of parallel execution.

I/O Challenges in MIMD Systems:

1. Standard C's Ambiguity for Processes: Standard C is a serial language, and its specifications do not define the behavior of I/O functions when called by multiple independent processes. This means their behavior in a multiprocess environment can vary significantly across different systems.
2. Nondeterministic Outcomes for Threads: While threads forked by a single process do share `stdin`, `stdout`, and `stderr`, simultaneous access by multiple threads to these shared resources leads to nondeterministic outcomes. It's impossible to predict the exact order interleaving of operations.

3. Output to Console (`printf`, `fprintf`):

- Developer Expectation: Developers usually want all output to appear on the console of the single system where the program is initiated.
- System Behavior: Most systems indeed coalesce output to a single console. However, there's no guarantee. Some systems might restrict stdout or stderr access to only one process, or even none.
- Nondeterminism: When multiple processes/threads can access stdout, the sequence of output is usually nondeterministic. Data from different processes/threads might interleave unexpectedly, making it hard to read or debug.

4. Input from Console (scanf, fscanf):

- Ambiguity: It's less obvious how scanf should behave with multiple processes/threads. Should input be divided? Should only one be allowed to read?
- System Behaviour: The majority of systems allow at least one process (commonly process 0) to call scanf. Some, and most allow multiple threads to call scanf. However, some systems might not allow any processes to call scanf.
- Non-determinism: When multiple processes/threads read from stdin, the data read by such an I/O on different runs, even with identical input, due to race conditions.

Assumptions and Rules for I/O in Parallel Programs:

- To manage these potential issues and ensure predictable (or at least understandable) I/O behavior in parallel programs, the following conventions are often adopted:

Standard Input (stdin) Access:

- Distributed-Memory Programs: Only process 0 will access stdin.
- Shared-Memory Programs: Only the master thread or thread 0 will access stdin.
- Rationale: This constrains input, avoiding nondeterminism and ensuring a single, clear source for input data.

Standard Output (stdout) and Standard Error (stderr) Access:

- Both Distributed and Shared-Memory Programs: All processes/threads can access stdout and stderr.
- However, for most cases: Only a single process/thread (e.g. process 0 or the master thread) will be used for all intended output to stdout.
- Exception: Debugging prints to stdout or stderr can be done by all processes/threads. This is acceptable because such output is for debugging/observation means, not for interpreted output.
- Rationale: This completely avoids race conditions and interleaving when dealing with file I/O.

File I/O (other than stdin/stdout/stderr):

- Only a single process/thread will attempt to access any specific file.
- This means each process/thread opens its own private file for reading or for writing.
- Rationale: This completely avoids race conditions and interleaving when dealing with file I/O.

Debug Output Best Practice:

- Debug output should always include the rank or ID of the process/thread generating the output. This is crucial for distinguishing messages and understanding the flow of execution from different parallel entities, especially when output is interleaved.

GPUs

GPUs and I/O: Host-Centric Approach with Debugging Exceptions

In GPU programming, the CPU host code is generally responsible for all input/output (I/O) operations. This simplifies I/O management significantly for a few key reasons:

- Typically, only one process or thread runs on the CPU host to manage the GPU computations. This means that the standard C I/O functions (like printf, scanf, fprintf, fscanf) behave exactly as they would in a regular serial C program, without the complexities of nondeterministic behavior found in multi-process/multi-thread CPU environments

GPU Limitations:

- GPUs are specialized for parallel computation and generally do not have direct access to standard I/O streams or secondary storage (like hard drives). This offloads I/O responsibilities entirely to the host.

Exception for Debugging:

- stdout for GPU Threads: In the systems commonly used for GPU programming, individual GPU threads can write to stdout.
- Nondeterministic Output: Similar to MIMD programs running on CPUs, when multiple GPU threads write to stdout concurrently, the order of the output is nondeterministic. This means output can vary from one run to the next.
- No Access to Other Streams/Storage: Crucially, GPU threads typically do not have access to stderr, media, or

secondary storage. This underscores the host's role for all but very specific debugging output.

Performance - Speedup and efficiency in MIMD systems

The primary goal of writing parallel programs is to achieve increased performance. When evaluating the performance of homogeneous MIMD systems (where all cores have the same architecture, unlike GPUs), we use metrics like speedup and efficiency.

Debug Output Best Practice:

- Debug output should **always include the rank or ID** of the process/thread generating the output. This is crucial for distinguishing messages and understanding the flow of execution from different parallel entities, especially when output is interleaved.

GPUs

GPUs and I/O: Host-Centric Approach with Debugging Exceptions

In GPU programming, the **CPU host code is generally responsible for all input/output (I/O) operations**. This simplifies I/O management significantly for a few key reasons:

Single Host Process/Thread:

Typically, only one process or thread runs on the CPU host to manage the GPU computations. This means that the standard C I/O functions (like `printf`, `scanf`, `fprintf`, `fscanf`) behave exactly as they would in a regular serial C program, without the complexities of nondeterministic behavior found in multi-process/multi-thread CPU environments.

GPU Limitations:

GPUs are specialized for parallel computation and generally **do not have direct access to standard I/O streams or secondary storage** (like hard drives). This offloads I/O responsibilities entirely to the host.

Exception for Debugging:

While the rule is to use the host for all I/O, there's a crucial exception for **debugging GPU code**:

- **stdout for GPU Threads:** In the systems commonly used for GPU programming, individual GPU threads *can* write to stdout.
- **Nondeterministic Output:** Similar to MIMD programs running on CPUs, when multiple GPU threads write to stdout concurrently, the **order of the output is nondeterministic**. This means the interleaved output can vary from one run to the next.
- **No Access to Other Streams/Storage:** Crucially, GPU threads typically **do not have access to stderr, stdin, or secondary storage**. This reinforces the host's role for all but very specific debugging output.

Performance – Speedup and efficiency in MIMD systems

The primary goal of writing parallel programs is to achieve increased performance. When evaluating the performance of **homogeneous MIMD systems** (where all cores have the same architecture, unlike GPUs), we use metrics like speedup and efficiency.

Speedup and Efficiency in MIMD Systems

Ideally, a parallel program divides work equally among cores with no added overhead.

If a program runs on p cores (one process or thread per core), the best possible runtime, T_{parallel} , would be T_{serial}/p , where T_{serial} is the runtime of the equivalent serial program on a single core of the same design.

When this ideal is achieved, we say the program has **linear speedup**.

Speedup (S) is formally defined as:

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

Linear Speedup: Occurs when $S = p$. This is the ideal scenario where the parallel program runs p times faster than the serial program.

Practical Considerations: In practice, achieving linear speedup is rare due to inherent **parallel overheads**:

- **Shared-memory programs:** Often involve critical sections that require mutual exclusion mechanisms (e.g., mutexes). Calls to mutex functions and serialization of critical sections introduce overhead not present in the serial version.
- **Distributed-memory programs:** Require data transmission across networks, which is significantly slower than local memory access.

These overheads are not present in serial programs and tend to **increase with the number of processes or threads (p)**.

For example, more threads mean more contention for critical sections, and more processes mean more network communication. Consequently, the speedup S usually becomes a smaller fraction of the ideal p as p increases.

Efficiency (E) is defined as:

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

Interpretation: Efficiency can be viewed as the **average utilization of the parallel cores** in solving the original problem. It represents the fraction of the parallel runtime that each core, on average, spends working on the actual problem, with the remainder being parallel overhead.

Calculation of Parallel Overhead: $E \cdot T_{parallel} = \frac{T_{serial}}{p}$. The total time spent in parallel overhead is $T_{parallel} - \frac{T_{serial}}{p}$.

Example:

If $T_{serial} = 24$ ms, $p = 8$, and $T_{parallel} = 4$ ms:

$$E = \frac{24}{8 \cdot 4} = \frac{24}{32} = \frac{3}{4}$$

On average, each process/thread spends $\frac{3}{4} \cdot 4$ ms = 3 ms on the original problem and 4 ms – 3 ms = 1 ms on parallel overhead.

If we express parallel runtime as $T_{parallel} = \frac{T_{serial}}{p} + T_{overhead}$, where $T_{overhead}$ is the parallel overhead, then efficiency is simply the fraction of parallel runtime spent on the original problem, as it explicitly separates work and overhead.

Dependence on Problem Size

$T_{parallel}$, S , and E are not only dependent on p but also significantly on the **problem size**.

Observation:

In many parallel programs, as the problem size increases (with p fixed), speedups and efficiencies generally **increase** (as illustrated in Tables 2.4 and 2.5 and Figures 2.18 and 2.19). Conversely, they decrease when the problem size is reduced.

Table 2.4 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

Table 2.5 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

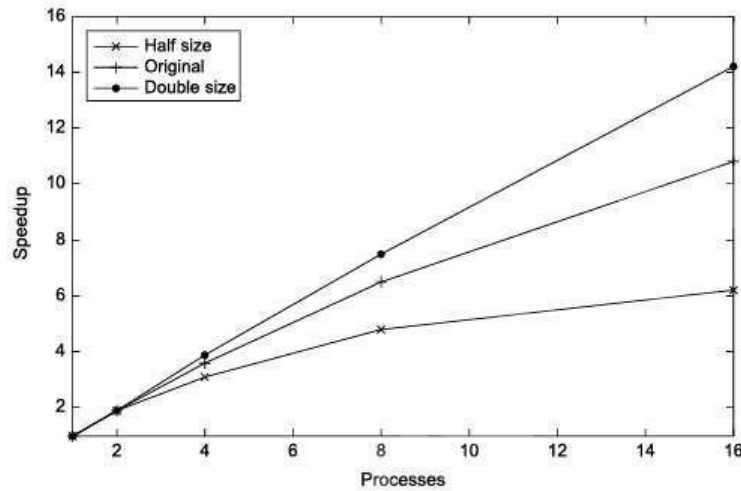


FIGURE 2.18

Speedups of parallel program on different problem sizes.

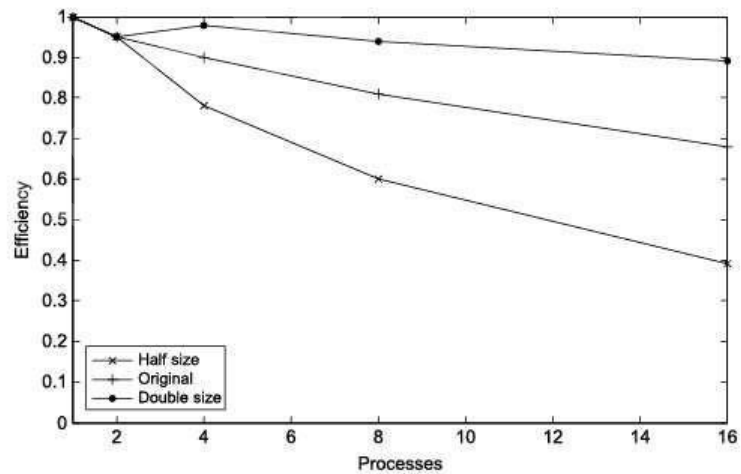


FIGURE 2.19

Efficiencies of parallel program on different problem sizes.

Reason:

This common behavior occurs because, for many parallel algorithms, the parallel overhead (T_{overhead}) grows much more slowly than the time spent solving the original problem (T_{serial}) as the problem size increases. That is, T_{serial} grows faster with problem size than T_{overhead} .

Choosing T_{serial} for Performance Measurement

When reporting speedups and efficiencies, it's crucial to define how T_{serial} is measured:

Approach 1 (Fastest possible serial program on fastest processor):

Some argue for using the runtime of the absolute fastest serial program (even if it uses a different algorithm) on the fastest available processor. This provides a "best-case" baseline for problem solving.

Approach 2 (Serial version of the parallel program on a single core of the parallel system):

In practice, most authors, including the text's authors, use the serial version of the *same program* that was parallelized, run on a single core of the parallel system itself.

- **Rationale:** This approach is more practical and provides a better measure of the *utilization of the parallel cores* for the specific parallelized algorithm, allowing efficiency to directly reflect how well the parallel system is being used for that problem.

The second approach is generally preferred because it provides a more direct comparison of the parallel implementation's effectiveness relative to its serial counterpart on the same hardware.

Amdahl's law

Amdahl's Law: The Limits of Parallel Speedup

Amdahl's Law, an observation made by Gene Amdahl in the 1960s, states that the potential speedup of a program due to parallelization is limited by the fraction of the program that remains inherently serial (unparallelized), regardless of how many processing cores are available.

Illustrative Example:

Let's assume:

- T_{serial} = Total serial runtime of the program (e.g., 20 seconds).
- f = Fraction of the program that can be parallelized (e.g., 90% or 0.9).
- r = Fraction of the program that remains unparallelized (inherently serial) ($r = 1 - f$, e.g., 10% or 0.1).
- p = Number of cores available.

If the parallelized part achieves perfect linear speedup (p times faster), then:

- **Runtime of the parallelized part:** $f \times \frac{T_{serial}}{p}$ (e.g., $0.9 \times \frac{20}{p} = \frac{18}{p}$)
- **Runtime of the unparallelized part:** $r \times T_{serial}$ (e.g., $0.1 \times 20 = 2$)

The **overall parallel runtime** ($T_{parallel}$) will be the sum of these two parts:

$T_{parallel} = \left(f \times \frac{T_{serial}}{p}\right) + (r \times T_{serial})$ Using our example:

$$T_{parallel} = \frac{18}{p} + 2$$

The **speedup** (S) is then:

$S = \frac{T_{serial}}{T_{parallel}} = \frac{T_{serial}}{\left(f \times \frac{T_{serial}}{p}\right) + (r \times T_{serial})} = \frac{1}{\left(\frac{f}{p}\right) + r}$ Using our example:

$$S = \frac{20}{\frac{18}{p} + 2}$$

The Limiting Factor:

As the number of cores (p) becomes very large ($p \rightarrow \infty$), the term $\frac{f}{p}$ (the parallelizable part's runtime divided by p) approaches 0. Therefore, the denominator in the speedup formula approaches $r \times T_{serial}$. This leads to the maximum possible speedup:

$$S_{max} = \frac{T_{serial}}{r \times T_{serial}} = \frac{1}{r}$$

In our example, where $r = 0.1$:

$$S_{max} = \frac{1}{0.1} = 10$$

This means that even with an infinite number of cores and perfect parallelization of 90% of the code, the speedup will never exceed 10. If r is as small as 1/100 (i.e., 99% parallelizable), the maximum speedup is still limited to 100.

Implications:

Amdahl's Law highlights a critical bottleneck: the **inherently serial portion of a program severely limits the achievable speedup**, no matter how much parallel processing power is thrown at the parallelizable parts.

Even a small serial fraction can cap the potential performance gains.

Why Not Be Too Worried by Amdahl's Law:

Despite its seemingly daunting implications, there are several reasons why Amdahl's Law isn't always a cause for despair in parallel programming:

Problem Size (Gustafson's Law):

- Amdahl's Law assumes a fixed problem size. However, for many problems, as the **problem size increases, the "inherently serial" fraction (r) often decreases proportionally** to the total work.
- This means that larger problems can exhibit much better speedups on many cores, a concept formalized by **Gustafson's Law**.
- In essence, as you scale up the problem, the parallelizable work often grows much faster than the serial work.

Empirical Evidence:

- Scientists and engineers routinely achieve **huge speedups** on large distributed-memory systems for real-world problems.
- This practical success demonstrates that for many relevant applications, the serial bottleneck is either sufficiently small or grows slowly enough relative to the parallelizable work.

Adequate Speedup:

- A "small" speedup (e.g., 5x or 10x) might still be more than sufficient for many applications.
- If the effort involved in developing the parallel program is reasonable, such speedups can translate into significant practical benefits, making the parallelization worthwhile even if theoretical linear speedup isn't achieved.

Scalability in MIMD systems

The term "scalable" is used informally in many contexts to mean that a system or program can handle increasing demands by growing its resources.

In the context of **MIMD (Multiple Instruction, Multiple Data) parallel program performance**, however, "scalability" has a more formal and precise definition.

Formal Definition of Scalability:

A parallel program is considered **scalable** if, when you increase the number of processes/threads (p) used by the program, you can find a corresponding rate of increase in the **problem size** such that the program maintains a **constant efficiency** (E).

In simpler terms: Can we keep the processors busy (maintain good efficiency) as we add more processors, provided we also make the problem bigger?

Example:

Let's illustrate with an example:

- Serial runtime: $T_{serial} = n$ (where n is the problem size in microseconds)
- Parallel runtime: $T_{parallel} = n/p + 1$

The efficiency (E) for this program is:

$$E = \frac{T_{serial}}{p \cdot T_{parallel}} = \frac{n}{p(n/p + 1)} = \frac{n}{n + p}$$

Now, to check for scalability, suppose we increase the number of processes/threads by a factor of k , so the new number of processes is kp . We want to find a factor x by which to increase the problem size (so the new problem size is xn) such that the efficiency remains the same:

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

Let's solve for x :

If we set $x = k$, then the right side becomes:

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

Since the expression simplifies back to the original efficiency E , this demonstrates that if we increase the problem size at the **same rate** (factor k) as we increase the number of processes/threads (factor k), the efficiency remains unchanged. Therefore, this program is **scalable**.

Types of Scalability:

Two specific types of scalability are defined based on how the problem size changes:

1. Strongly Scalable:

- A program is **strongly scalable** if, when we increase the number of processes/threads, we can keep the efficiency fixed **without increasing the problem size**.
- This is generally harder to achieve because, for a fixed problem, increasing processors inevitably increases communication/synchronization overhead relative to computation, thus often decreasing efficiency.

2. Weakly Scalable:

- A program is **weakly scalable** if we can keep the efficiency fixed by **increasing the problem size at the same rate as we increase the number of processes/threads**.
- The example program discussed above ($T_{serial} = n$, $T_{parallel} = n/p + 1$) is an example of a **weakly scalable** program. This is a more common and often more practical form of scalability in parallel computing, as it acknowledges that larger problems can effectively utilize more resources.

Taking timings of MIMD programs

Accurately measuring the performance of MIMD (Multiple Instruction, Multiple Data) parallel programs requires careful consideration beyond simple start-to-finish timers.

The methodology depends on the purpose of the timing and the nature of parallel execution.

Reasons for Taking Timings

There are generally two main reasons for timing parallel programs:

During Program Development (Detailed Analysis):

To understand if the program is behaving as intended. This often requires very detailed information, such as time spent in specific code sections, communication overheads, or waiting times (e.g., processes waiting for messages in distributed-memory programs). Large waiting times can indicate design or implementation flaws.

After Development (Overall Performance Evaluation):

To determine the program's overall performance. For this, a single, concise runtime value is usually reported. We will focus on this second type of timing here.

Key Considerations for Performance Timings**Timing Specific Code Sections, Not Entire Program:**

- We are usually interested in the time taken by the *core computational part* of the program, not the total execution time from start to finish (which includes setup, I/O, etc.).
- Therefore, standard shell commands like `time` (Unix) are often insufficient as they report total program execution time.

Wall Clock Time vs. CPU Time:

- We are generally **not interested in "CPU time"** (time spent actively executing instructions or system calls initiated by the program). The standard C `clock()` function reports CPU time.
- **Problem with CPU time:** In parallel programs, processes/threads might spend significant time *idle* (e.g., waiting for other processes/threads to send data, or waiting for a mutex to be released). This idle time is not counted as CPU time because no function called by the process is active. However, this idle time is a *real cost* to the overall parallel execution and must be included in performance evaluation to provide an accurate picture of the program's true runtime.
- **Solution: Wall Clock Time:** The reported time for parallel programs is almost always **"wall clock time"** (also known as elapsed time or real time). This measures the total time that elapses from the beginning to the end of the timed section, as if measured by a stopwatch.

A typical structure for measuring wall clock time is:

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

- **Get_current_time() examples:** Specific APIs provide such functions:
 - **MPI:** `MPI_Wtime()`
 - **OpenMP:** `omp_get_wtime()` Both return wall clock time.

Timer Resolution:

- **Definition:** Timer resolution is the smallest unit of measurement the timer can report (e.g., milliseconds, microseconds, nanoseconds).
- **Issue:** If the resolution is too coarse (e.g., milliseconds) and the timed code section executes very quickly (e.g., less than a millisecond), the timer might report zero or an inaccurate value.
- **Verification:** Many APIs provide functions to query the timer's resolution, or they specify a minimum required resolution. Programmers must check these values.

Timing Parallel Sections Carefully:

- When the code being timed is executed by multiple processes or threads, each will record its own `my_start` and `my_finish` times, resulting in `p` individual elapsed times.
- However, what's usually desired is a **single overall time**: the duration from when the *first* process/thread began the timed section to when the *last* process/thread finished it.
- **Challenge:** Due to potentially unsynchronized clocks across different nodes (in distributed memory), obtaining this exact "global" start and end time is often difficult.
- **Compromise Approach:**

C

```

shared double global_elapsed;
private double my_start, my_finish, my_elapsed;
// ...
/* Synchronize all processes/threads */
Barrier(); // e.g., MPI_Barrier(), #pragma omp 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); // e.g., MPI_Reduce(..., MPI_MAX, ...)
if (my_rank == 0) { // Or master thread
    printf("The elapsed time = %e seconds\n", global_elapsed);
}

```

Barrier:

A Barrier() function ensures that all processes/threads reach a certain point before any can proceed past it. While it doesn't guarantee simultaneous return, it provides a crucial synchronization point.

Global Maximum:

Each process/thread calculates its `my_elapsed` time. A global reduction operation (like `Global_max` or `MPI_Reduce` with `MPI_MAX`) is then used to find the maximum of all these individual elapsed times.

This maximum represents the time it took for the *slowest* process/thread to complete its portion of the work, effectively capturing the total time the parallel section was active. Process/thread 0 (or the master) typically prints this value.

Variability in Timings and Reporting Minimum Time:

- Running a program multiple times will almost certainly yield slightly different elapsed times, even with identical inputs and systems.
- While reporting the mean or median might seem intuitive, it's generally unlikely for external factors to make a program run *faster* than its best possible time. External interference (OS activity, network jitter, etc.) almost always makes it run *slower*.
- Therefore, it's common practice to report the **minimum time** observed across multiple runs. This minimum time is considered the most representative of the program's optimal performance under the given conditions.

Threads per Core:

- Running more than one thread per physical core can significantly increase timing variability and introduce extra overhead due to system scheduling and descheduling.
- For this reason, it's generally recommended to run **no more than one thread per physical core** when aiming for optimal performance and consistent timings.

Excluding I/O from Reported Timings:

- Since most parallel programs (unless specifically designed for high-performance I/O) do not optimize for I/O operations, it's common practice to **exclude I/O time** (reading inputs, printing final results) from the reported performance timings. The focus remains on the core computation time.

GPU performance

GPU Performance Evaluation

While the general goal of parallel programming is increased performance, evaluating **GPU program performance** differs from homogeneous MIMD systems due to the distinct architecture of GPUs.

Comparing GPU Performance

It's common to report **speedups of GPU programs over serial CPU programs or parallel MIMD programs**. This gives a straightforward indication of the performance gain achieved by using the GPU.

However, certain metrics and concepts, typically applied to MIMD systems, are **less meaningful or directly applicable** to GPUs:

Efficiency:

In MIMD systems, efficiency is often calculated by comparing parallel runtime to a serial runtime on a single core of the *same type* used by the parallel system.

Since GPU cores are fundamentally different (inherently parallel, highly specialized) from conventional CPUs, this type of direct comparison to a "single GPU core serial execution" **doesn't make sense**.

Therefore, **efficiency is ordinarily not used** in discussions of GPU performance in the same formal way.

Linear Speedup:

Similarly, talking about "linear speedup" of a GPU program relative to a serial CPU program is not directly applicable.

A CPU and a GPU operate on vastly different architectural principles, so a direct scalar comparison of core counts for "linear" scaling is inappropriate.

Scalability in GPUs

- The **formal definition of scalability** (maintaining constant efficiency by increasing problem size proportionally to processor count) from MIMD programs **cannot be directly applied** to GPUs because the concept of "efficiency" (as previously defined) doesn't directly translate.

- However, the **informal usage of scalability is routinely applied to GPUs**: a GPU program is considered scalable if increasing the size or power of the GPU (e.g., more streaming multiprocessors, more memory bandwidth) results in a proportional speedup for the program.
- This implies that the algorithm can effectively utilize more GPU resources when available.

Amdahl's Law and GPUs

Applicability: If the inherently serial part of a GPU program (the portion that cannot be parallelized and runs on the CPU) is considered, then **Amdahl's Law can indeed be applied to GPU programs**.

Speedup Upper Bound:

The resulting upper bound on possible speedup will be the same as for MIMD programs: if a fraction r of the original serial program remains unparallelized and runs on a conventional serial processor, the best possible speedup for the combined CPU-GPU program will be **less than $1/r$** .

Caveats: The same caveats that apply to Amdahl's Law in MIMD systems also hold true for GPUs:

- The "inherently serial" fraction (r) is likely **dependent on the problem size**. For larger problems, r often decreases, leading to higher potential speedups.
- Despite the theoretical limits, many GPU programs achieve **huge speedups** in practice, demonstrating their effectiveness for highly parallelizable workloads.
- Even a relatively small speedup (e.g., 5x or 10x) can be perfectly adequate for many applications, making GPU acceleration worthwhile.

Taking Timings of GPU Programs

The basic principles of timing discussed for MIMD programs also apply to GPU programs: measuring wall clock time for the relevant sections.

Overall GPU Program Performance:

Since a GPU program is typically initiated and concluded on a conventional CPU, for overall performance measurement of the GPU-accelerated parts, you can usually just use the **CPU's timer**.

- Start the timer before initiating the GPU computation(s).
- Stop the timer after the GPU computation(s) have completed and results are returned to the host.

For more intricate setups (e.g., programs running on multiple CPU-GPU pairs), more sophisticated timing mechanisms (potentially involving synchronization across CPU-GPU pairs) would be required.

However, for simpler cases, the CPU host timer is sufficient.

Timing Subsets of GPU Code: If you only want to time a specific portion of the code *running directly on the GPU*, you will need to use **timer functions provided by the GPU's programming API** (e.g., CUDA events for NVIDIA GPUs).

These allow for precise measurement of kernels and data transfers on the GPU device itself.