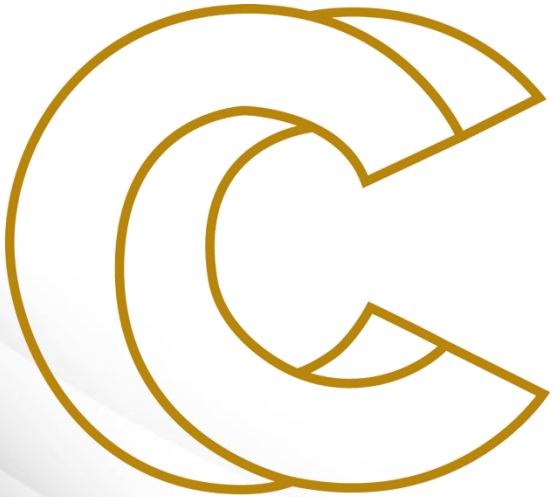


EURO 4SEE

Introduction to HPC using GPUs,

Denizhan Tutar, Istanbul Technical University

ncc@ulakbim.gov.tr

Lesson 2: Remarks on Parallelism

What is parallelism

- When we want to speed up a calculation, there's only so much things we can do with only one calculator
- In real world, we are trying to reduce total time needed for a work by increasing number of workers assigned to it
- However, some of the work might be non-parallelizable: we should do it sequentially anyway
- Also, organizing workers is not an easy task and would have some overhead: you're really lucky if using 10x workers means 10x less time for parallelizable part too

Lesson 2: Remarks on Parallelism

Scaling

- We want to know how fast a work will be done for how many workers added.
- **Speedup**: the ratio of time to complete the problem on **one PE** (processing element—which can be a node, processor, or core), **versus** time to complete the problem on **N PEs**
- Scaling refers to a program's expected speedup with increasing PEs.

- We evaluate scaling for either **fixed problem size** (strong scaling) or **fixed problem size processor(PE)** (weak scaling)

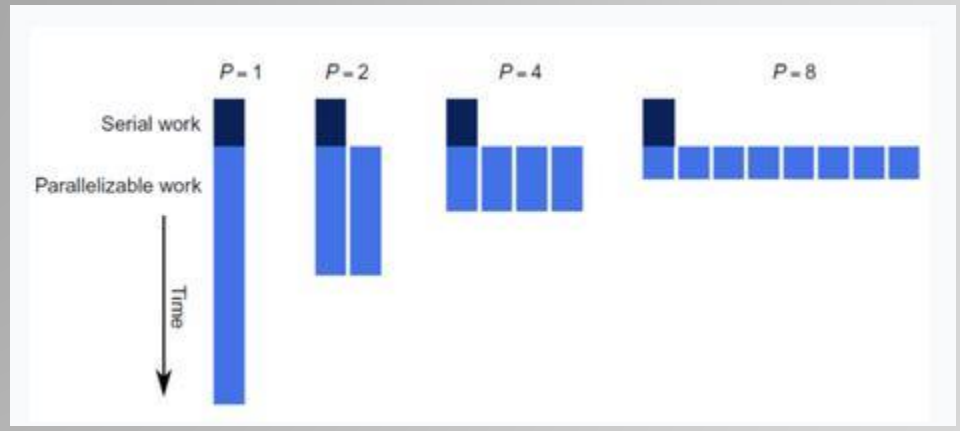
Lesson 2: Remarks on Parallelism

Scaling with Fixed Problem Size (strong scaling, Amdahl's law)

- We will assume our code consists of 2 parts: a serial part, which will not be affected by number of workers, and a parallelizable part, whose needed time will be inversely proportional to the number of workers at limit (best) case.
- Measures **how fast** I'd do **the same work** with more workers
- P: parallelizable part (between 0 and 1)
- N: number of workers (processes, threads, …)

$$\text{Speedup} = \frac{1}{(1-P) + \frac{P}{N}}$$

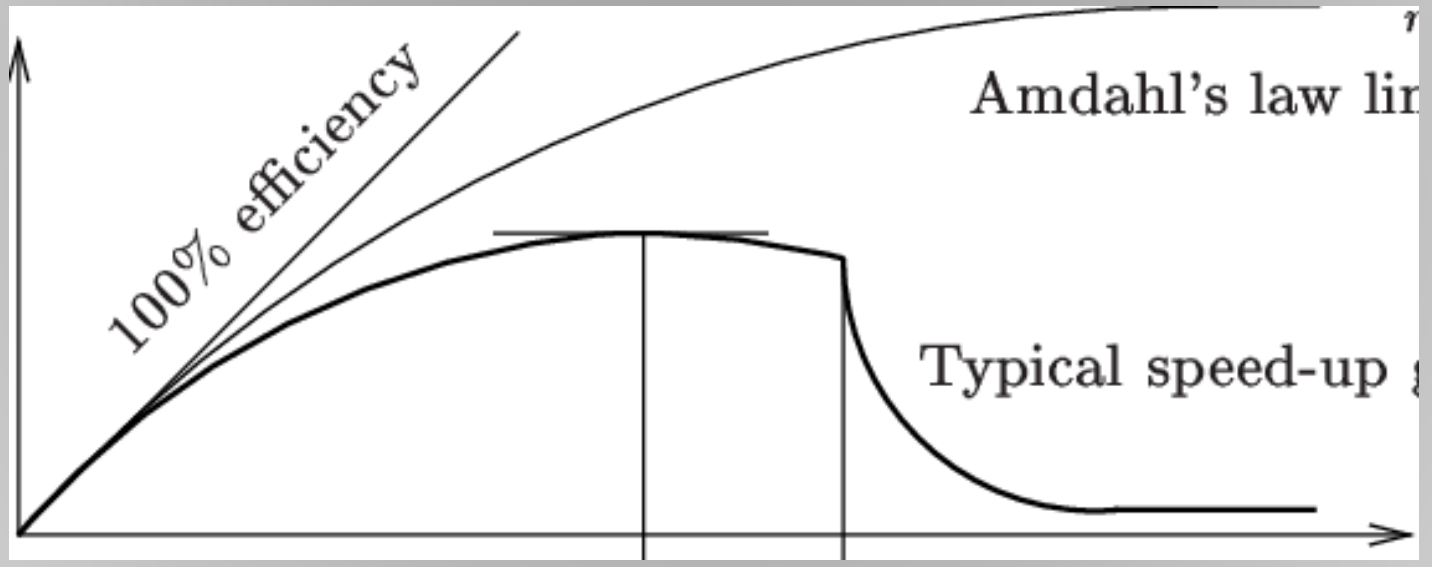
- Reflects the law of diminishing returns



Lesson 2: Remarks on Parallelism

Scaling with Fixed Problem Size (strong scaling, Amdahl's law)

- **Limit** at infinite number of workers: parallel part vanishes, so **1/(1-P)**
- Reflects the law of diminishing returns

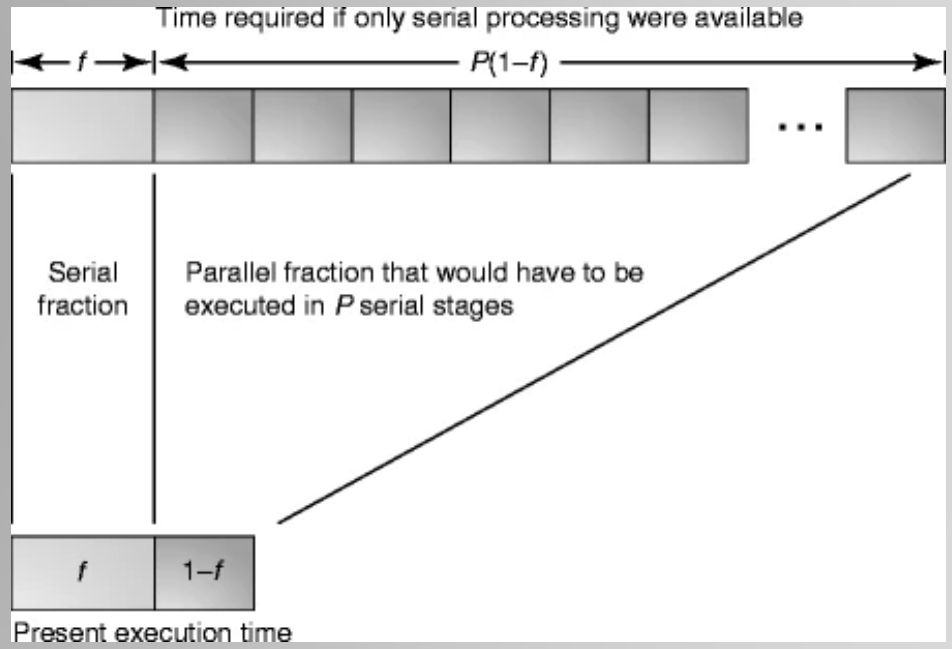


Lesson 2: Remarks on Parallelism

Scaling with Fixed Problem Size <u>per worker</u> (weak scaling, Gustafson's law)

- Measures **how much work** I'd do **in the same time** with more workers
- Speedup: the ratio of time to complete the problem on one PE (processing element—which can be a node, processor, or core), versus time to complete the problem on N PEs **(but problem size = N x reference problem size)**
- Serial time for new problem size: serial part + N*parallelizable part = (1-P) + N*(P)
- Speedup: ((1-P) + N*(P))/ ((1-P) + P = 1) =

  (1-P) + N*(P) = N - (1-P)*(N-1)

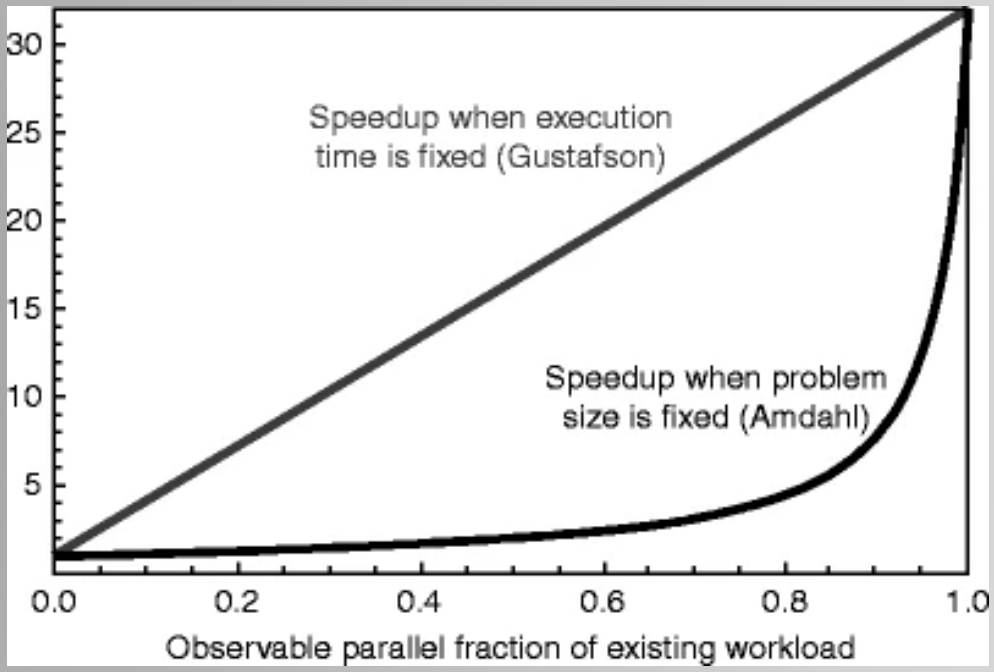


Lesson 2: Remarks on Parallelism

Scaling with Fixed Problem Size <u>per worker</u> (weak scaling, Gustafson's law)

- **Limit** at infinite number of workers: **infinity**

- Comparing two might be meaningless, they measure different kind of returns; but both shows the importance of increasing parallel fraction of the code



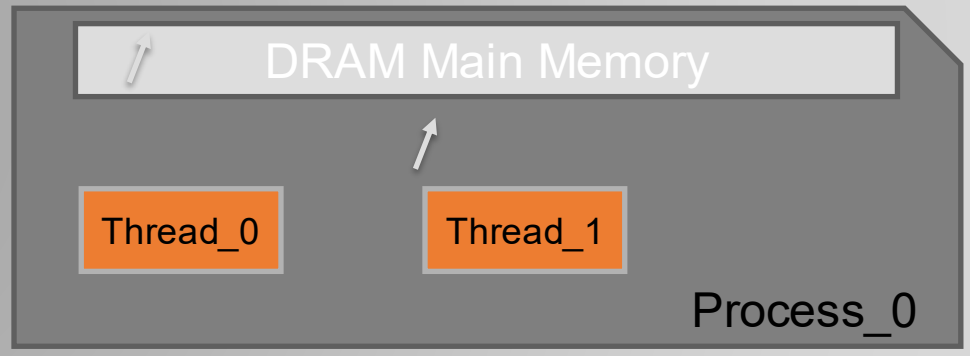
For 32 threads

Lesson 2: Remarks on Parallelism

Types of Parallelism

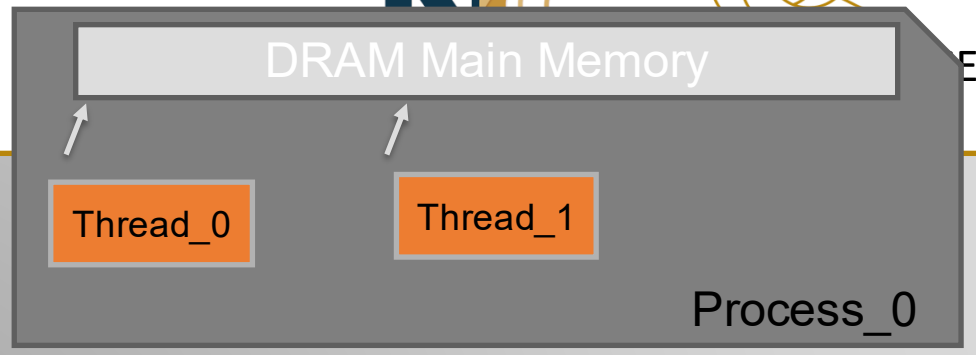
- Shared Memory: OpenMP
  - Better suited for Uniform Memory Access (UMA) cases
  - Compiler & CPU handles copy which data & when & where

```
#include <omp.h>
// Function to perform y = ax + b
void y_ax_plus_b(const std::vector<double>&x,
 std::vector<double>& y, double a, double b) {
   int n = x.size();
   omp_set_num_threads(thread_count); //optional
      #pragma omp parallel for
   for (int i = 0; i < n; i++) { y[i] = a * x[i] + b;
} }
```



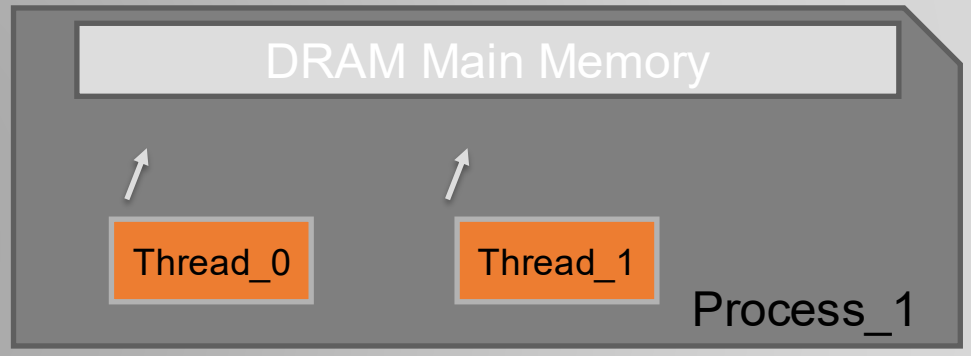

```
g++ -fopenmp -o myprogram
mycode.cpp ./myprogram
```

Lesson 2: Remarks on Parallelism



Types of Parallelism

- Distributed Memory: MPI
- More suitable to NUMA (Non-Uniform Memory Access)



- Process doesn't share
  - Memory addresses as return of malloc
    Except of memory Windows in MPI-2 and 3
  - Accesible memory addresses
  - Most variable values (global variables are possible but a headache)
  - Usually CPU core they run into (often 1 or 2 process per core)

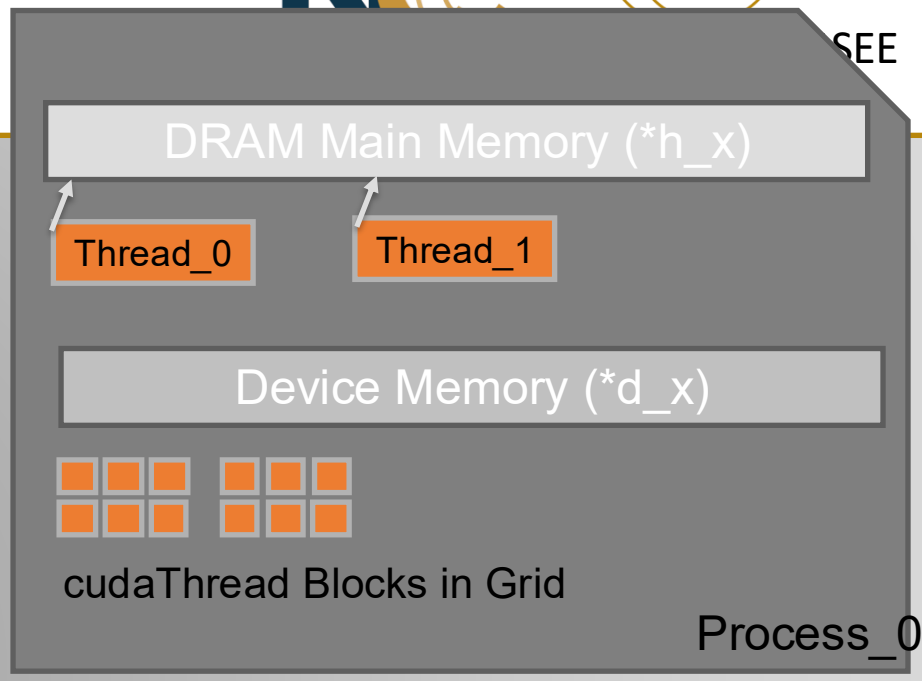
Lesson 2: Remarks on Parallelism

Types of Parallelism

- Using a device/accelerator
  - Resources are seperate bw device&host

  (dereferencing a device pointer on host would cause a crash) (but unified memory is implemented)
  - Resources are shared by threads

  (we use built-in variables that differs by thread to organize reaching different places, but pointers hold same addresses) -> fetching, synchronization etc. more similar to OpenMP



Lesson 2: Remarks on Parallelism

Types of Parallelism

Feature	CUDA	OpenMP	MPI	(C++17+)
Programming Model	GPU-based parallelism (SIMT)	Shared-memory CPU parallelism	Distributed-memory parallelism	Shared-memory CPU parallelism (with STL)
Ease of Use	Complex (requires GPU knowledge)	Simple (directives)	Moderate (requires communication handling)	Simple (via `std::for_each` with execution policies)
Scalability	Excellent on GPUs	Limited to CPU cores (shared memory)	Excellent on clusters and large systems	Limited to CPU cores (shared memory)
Performance	High for GPU-accelerated tasks	Good for CPU multi-core	Good for large-scale distributed systems	Good for CPU multi-core with optimizations
Use Cases	Deep learning, simulations, HPC	Data parallelism, scientific computing	Large-scale scientific computing, supercomputing	Data parallelism, general-purpose CPU tasks
Complexity	High (GPU-specific optimizations)	Low (easy to add directives)	Moderate (requires managing inter-process communication)	Low (just add `std::execution::par` to STL algorithms)
Key Features	Explicit memory management, thread/block management, GPU-specific optimizations	Compiler directives (`#pragma`) for parallel loops	Message-passing (e.g., `MPI_Send`, `MPI_Recv`)	Parallel algorithms (e.g., `std::for_each`, `std::transform`) with execution policies
Hardware Requirements	Requires NVIDIA GPUs	Multi-core CPUs	Multi-node distributed systems	Multi-core CPUs (with C++17 and later)

Lesson 2: Remarks on Parallelism

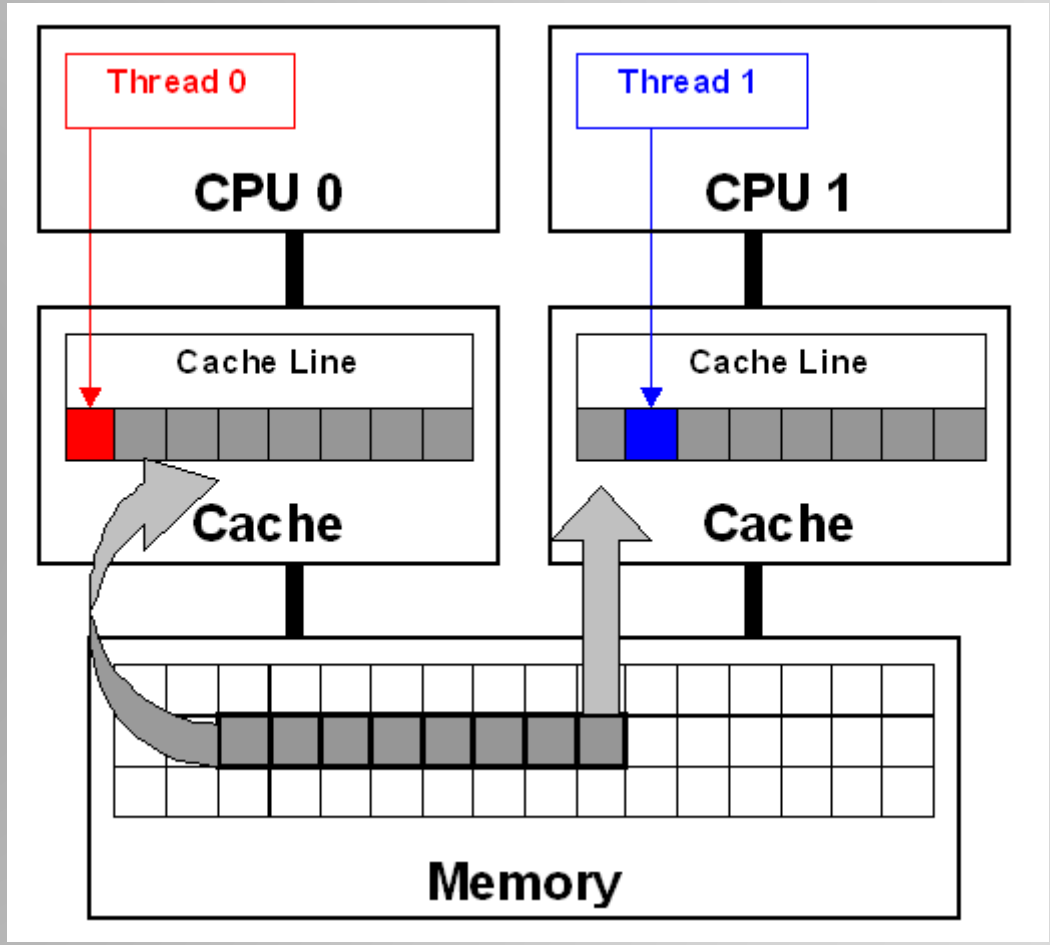
OpenMP examples

- Ex-1: for loop that initializes array
- Ex-2: private variables & **race condition**
- Ex-3: sum of an array: preventing race condition by serializing the work

Lesson 2: Remarks on Parallelism

Race Condition & False Sharing

- Race Conditions may happen when more than one thread access the same data (at least one writes)
- It make the program wrong, so must be dealt with
- In many cases, that'll be done by a locking mechanism that ensures that only one thread is modifying the data, serializing the workflow for that operation (keyword: **cache coherence**)
- Threads will still take nearby elements into cache, so modifying nearby elements would be serialized too (**false sharing**): it's closely related to bank conflicts on GPU



Next lecture

Introduction to GPUs

References

- https://link.springer.com/rwe/10.1007/978-0-387-09766-4_78
- https://akarshbolar.wordpress.com/2017/03/12/false-sharing-and-race-condition/

Copyright

EURO 4SEE

Thanks!

Co-funded by
the European Union

EuroHPC
Joint Undertaking

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 101191697. The JU receives support from the Digital Europe Programme and Germany, Türkiye, Republic of North Macedonia, Montenegro, Serbia, Bosnia and Herzegovina.