Introduction to parallel and distributed computing

Marc Moreno Maza

Ontario Research Center for Computer Algebra
Departments of Computer Science and Mathematics
University of Western Ontario, Canada

CS4402 - CS9635, January 30, 2024

Introduction to parallel and distributed computing

Marc Moreno Maza

Ontario Research Center for Computer Algebra
Departments of Computer Science and Mathematics
University of Western Ontario, Canada

CS4402 - CS9635, January 30, 2024

Plan

1. Hardware architecture and concurrency

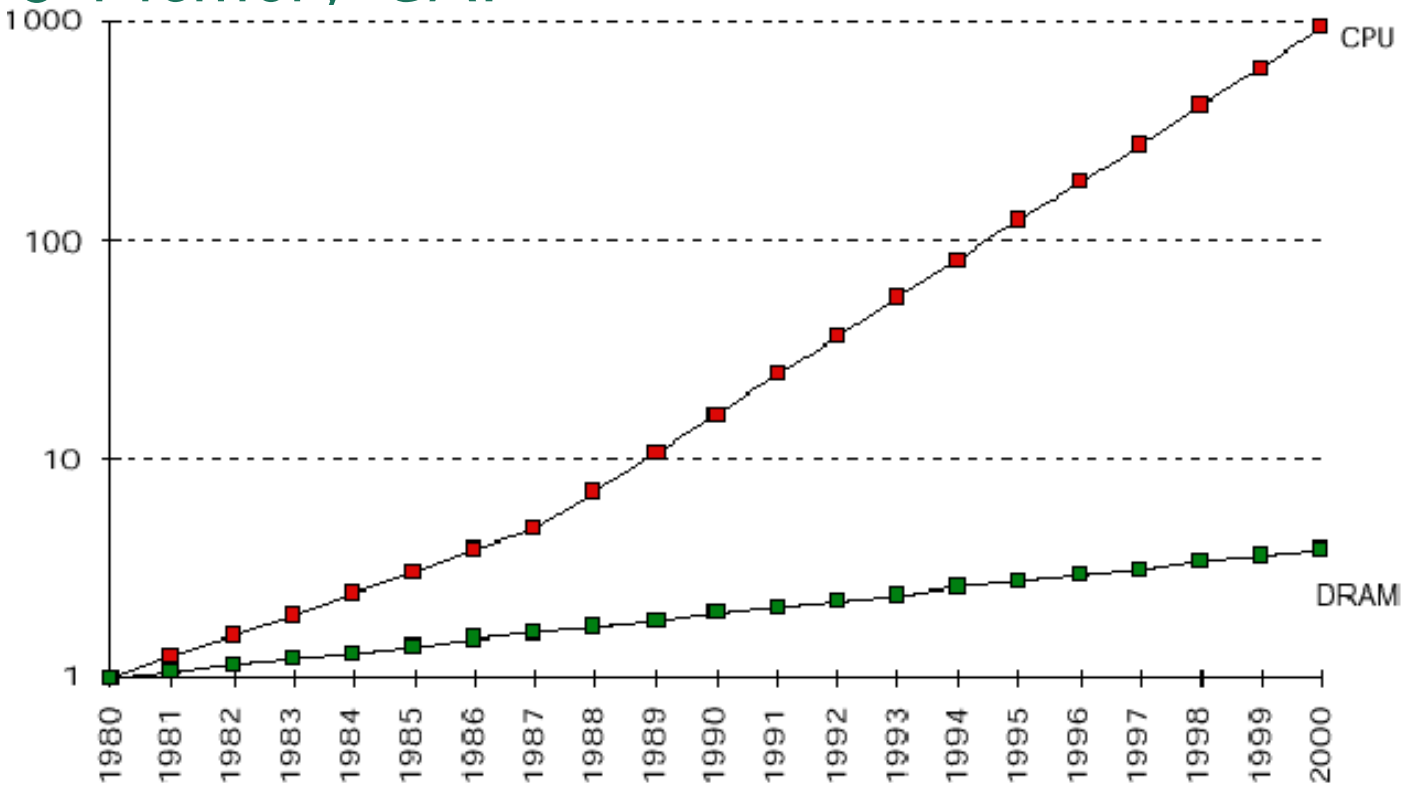
2. Parallel programming patterns

3. Concurrency platforms

Outline

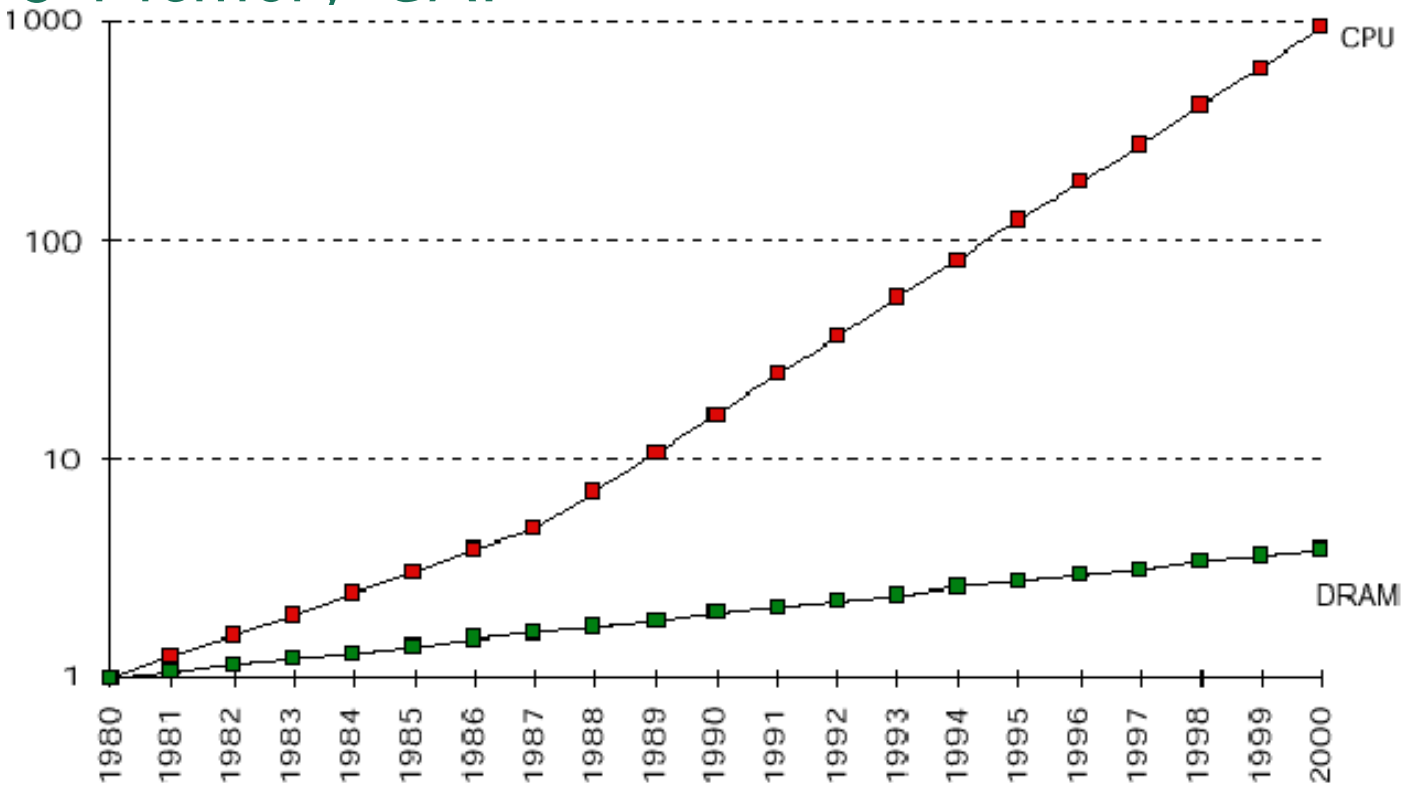
1. Hardware architecture and concurrency
2. Parallel programming patterns
3. Concurrency platforms

The CPU-Memory GAP



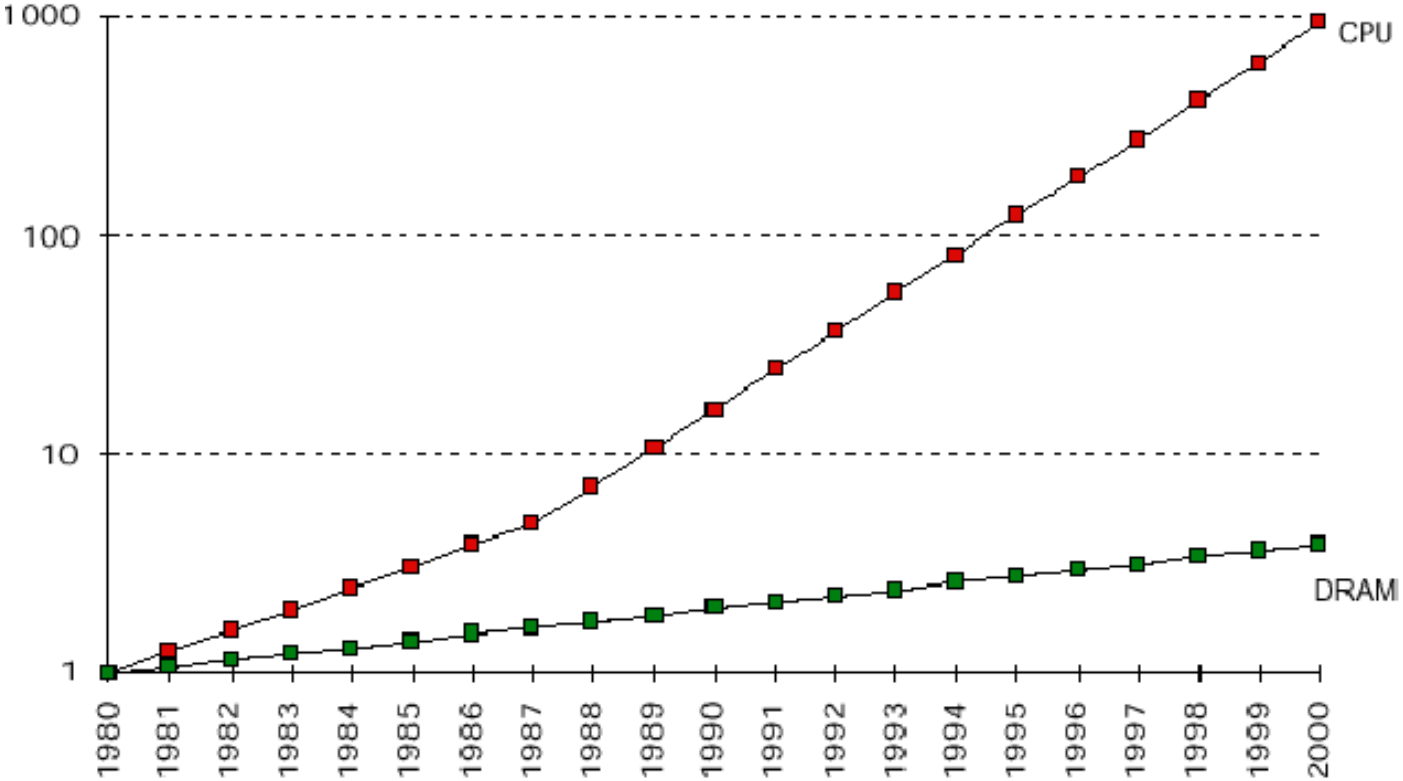
- In the 1980's, a memory access and a CPU operation were both as slow as the other

The CPU-Memory GAP



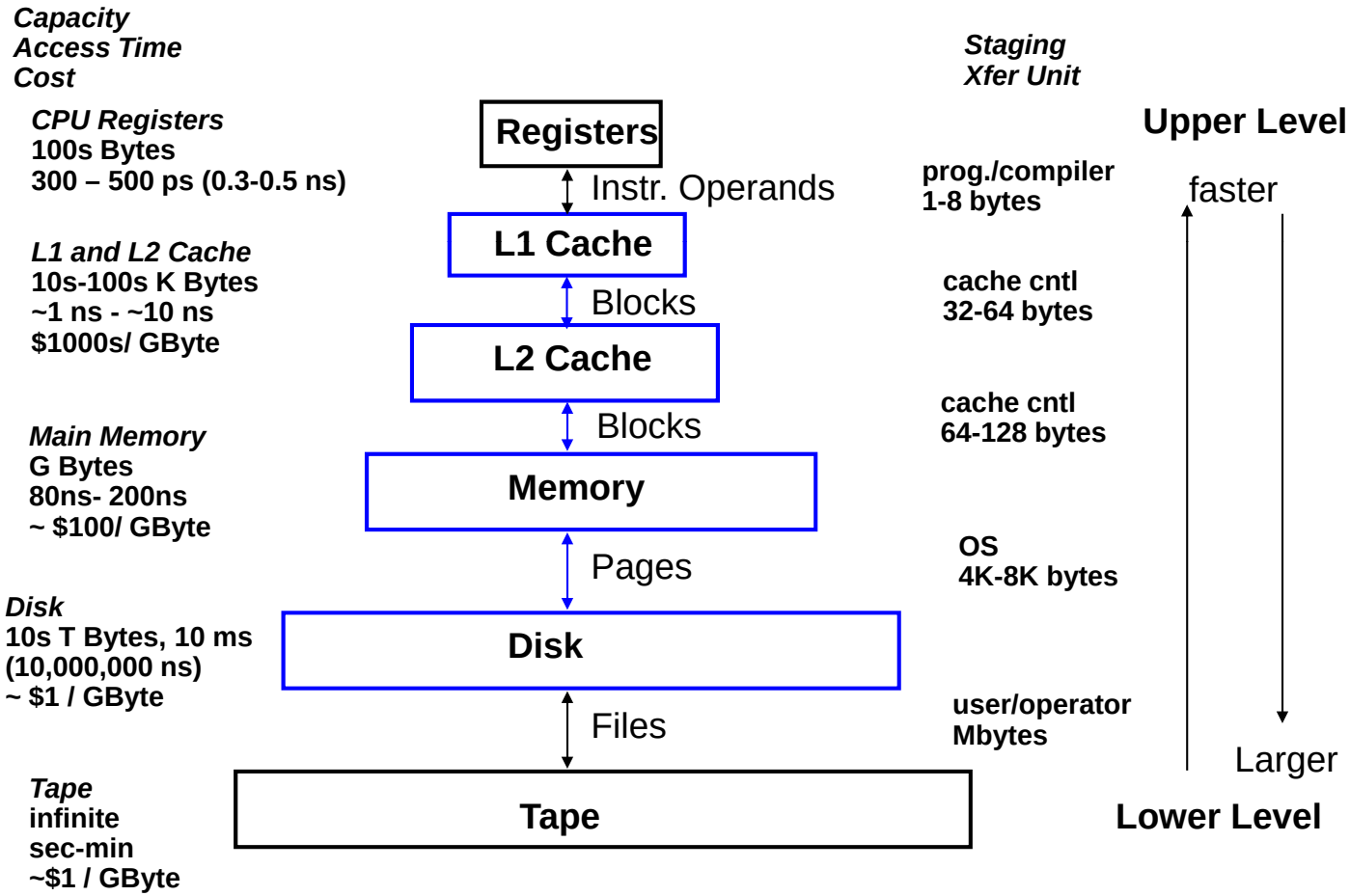
- In the 1980's, a memory access and a CPU operation were both as slow as the other
- CPU frequency increase, between 1985 and 2005, has reduced CPU op times much more than DRAM technology improvement could reduce memory access times.

The CPU-Memory GAP



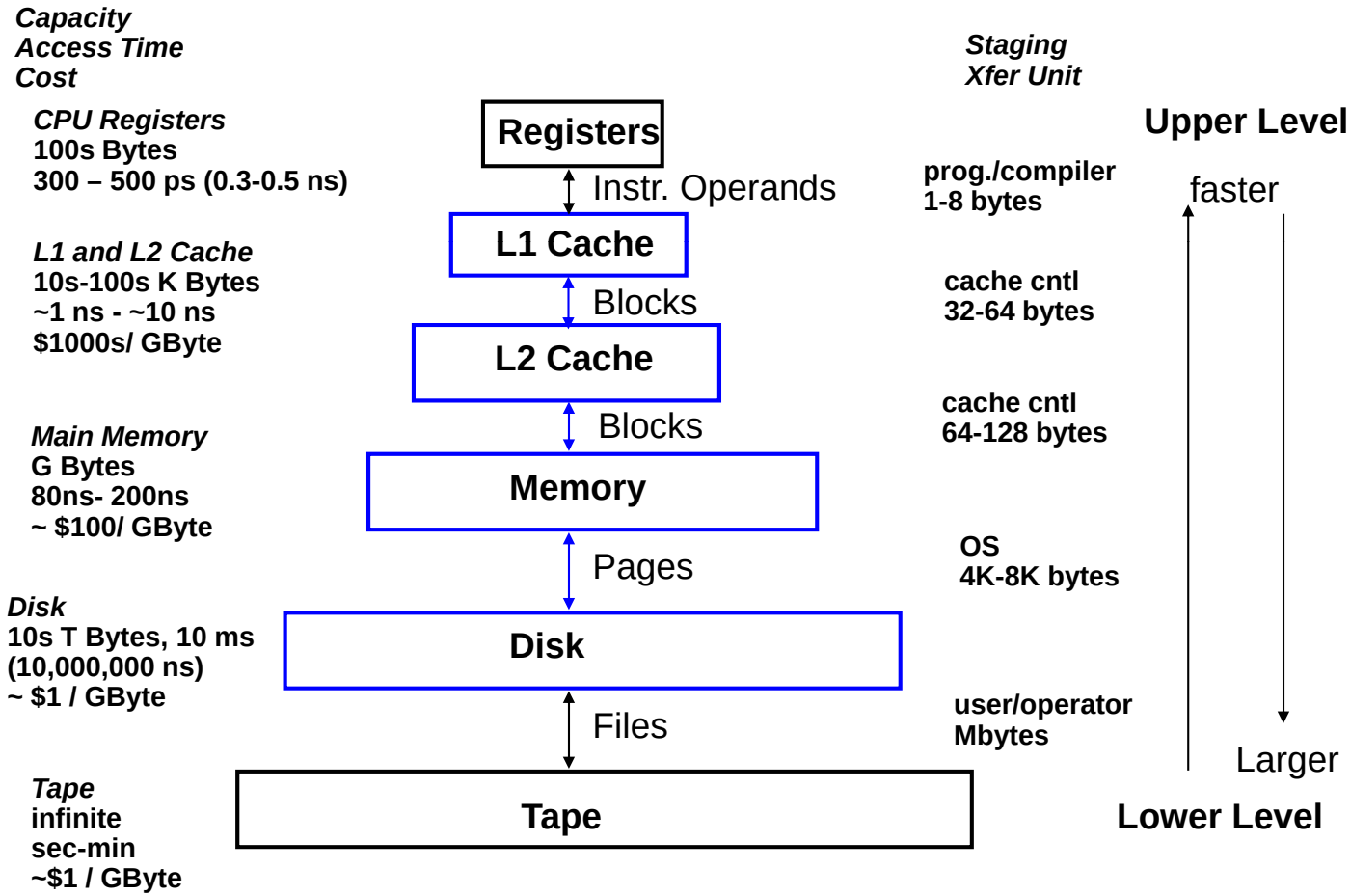
- In the 1980's, a memory access and a CPU operation were both as slow as the other
- CPU frequency increase, between 1985 and 2005, has reduced CPU op times much more than DRAM technology improvement could reduce memory access times.
- Even after the introduction of multicore processors, the gap is still huge.

Hierarchical memory



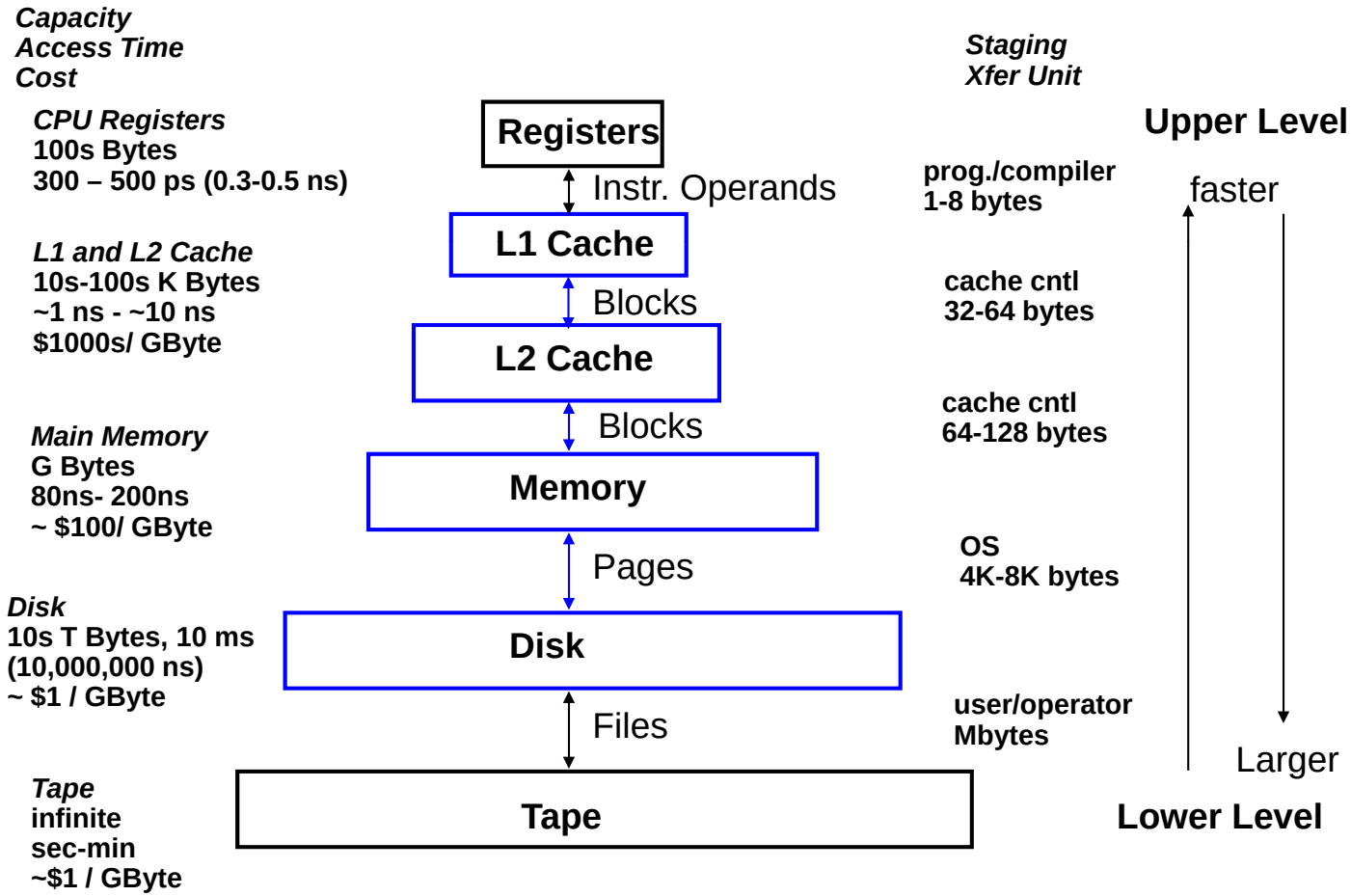
- Data moves in blocks (cache-lines, pages) between levels

Hierarchical memory



- Data moves in blocks (cache-lines, pages) between levels
- On the right, note the block sizes

Hierarchical memory



- Data moves in blocks (cache-lines, pages) between levels
- On the right, note the block sizes
- On the left, note the access times, sizes and prices.

Moore's law

The Pentium Family: Do not rewrite software, just buy a new machine!

https://en.wikipedia.org/wiki/Moore%27s_law

From Moore's law to multicore processors

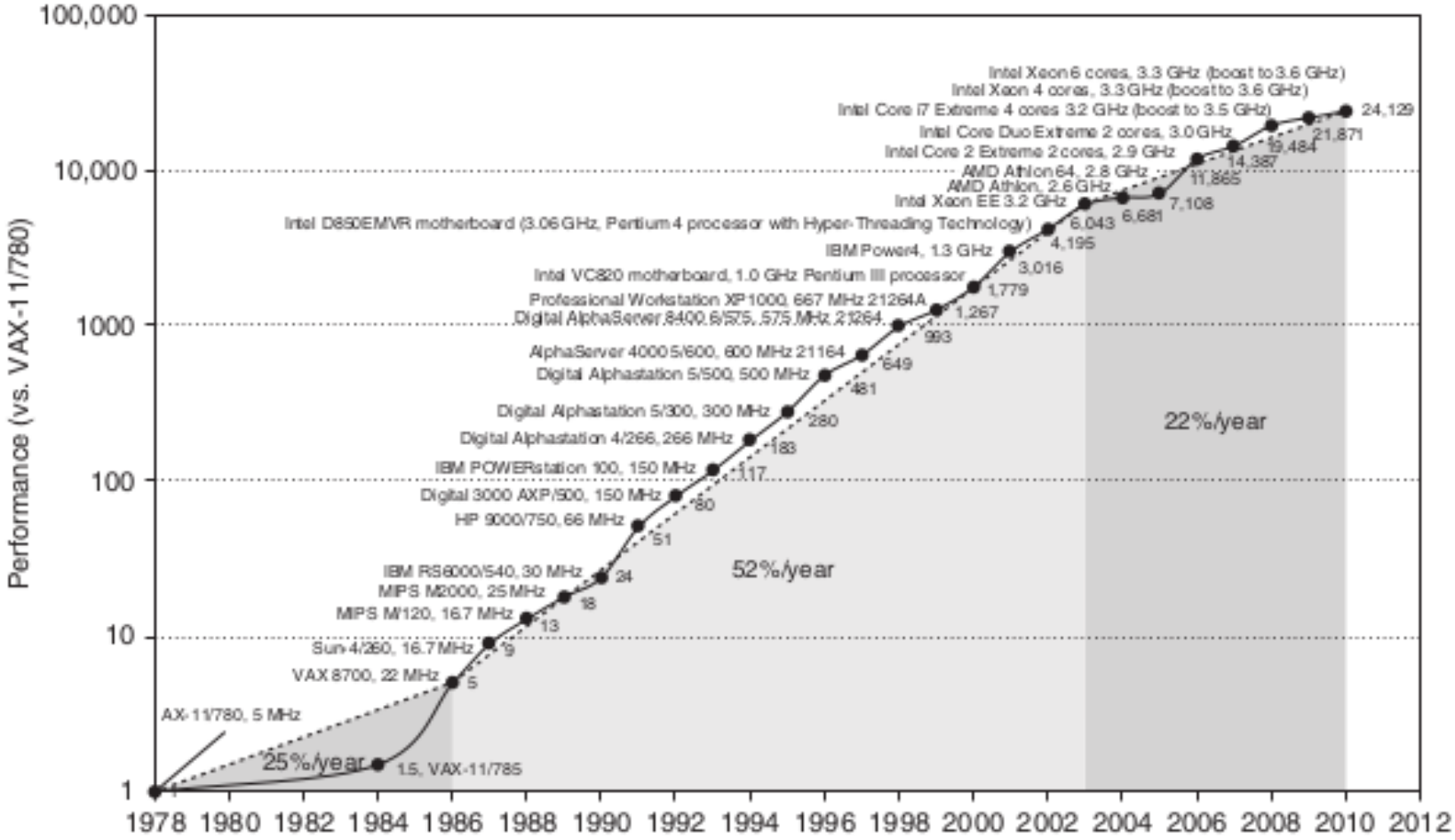
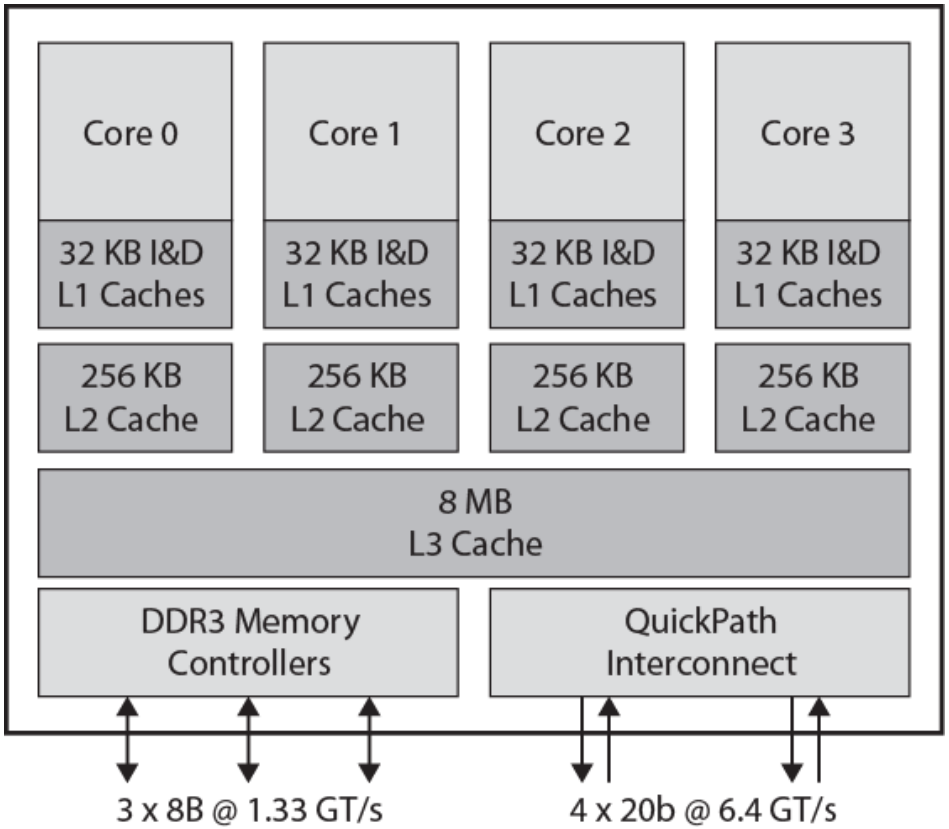


Image taken from Hennessy, Patterson. Computer Architecture, a quantitative approach. 5 th Ed. 2010.

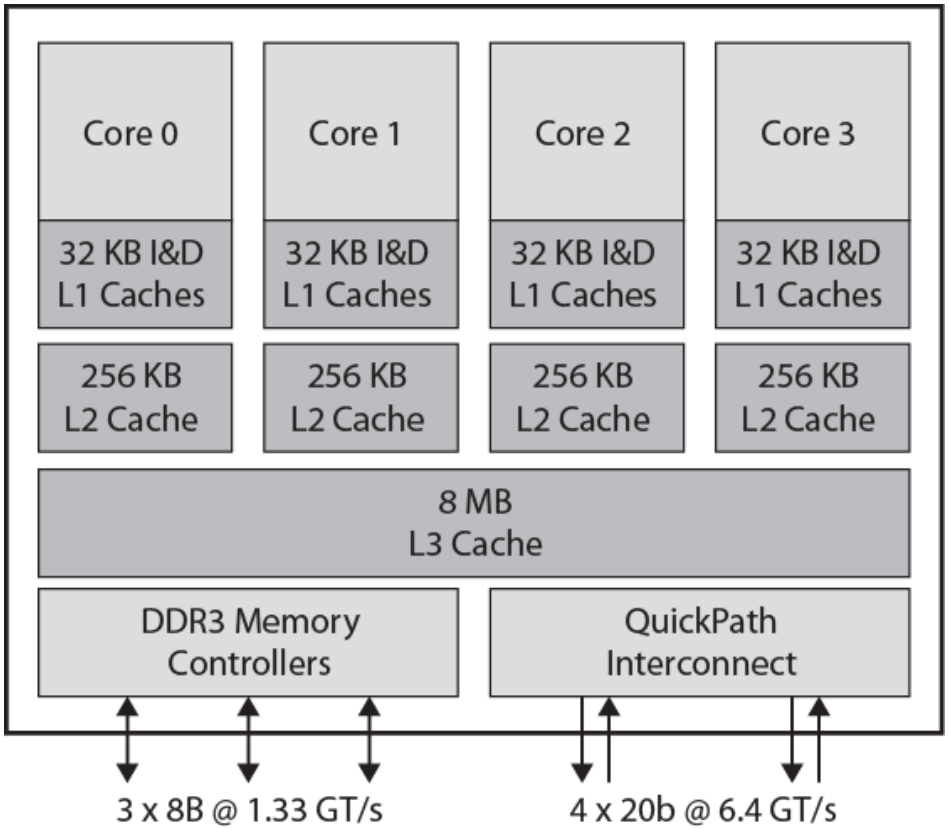
Multicore processors

Multicore processors



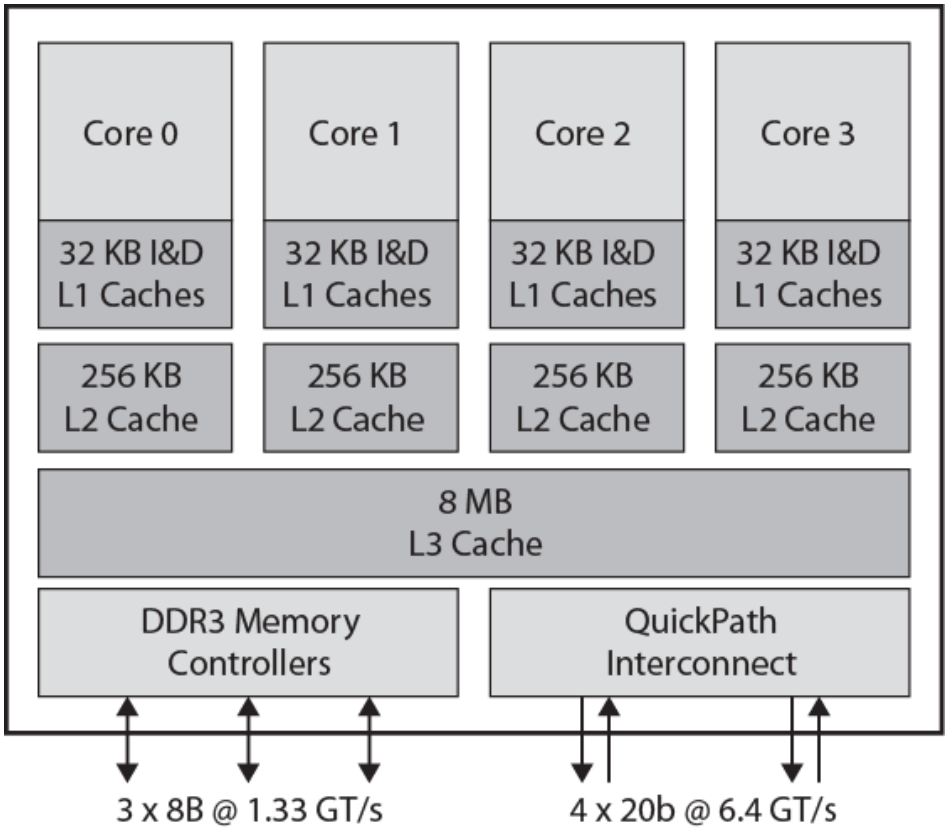
- In the 1st Gen. Intel Core i7, each core had an L1 data cache and an L1 instruction cache, together with a unified L2 cache

Multicore processors



- In the 1st Gen. Intel Core i7, each core had an L1 data cache and an L1 instruction cache, together with a unified L2 cache
- The cores share an L3 cache

Multicore processors



- In the 1st Gen. Intel Core i7, each core had an L1 data cache and an L1 instruction cache, together with a unified L2 cache
- The cores share an L3 cache
- Note the sizes of the successive caches

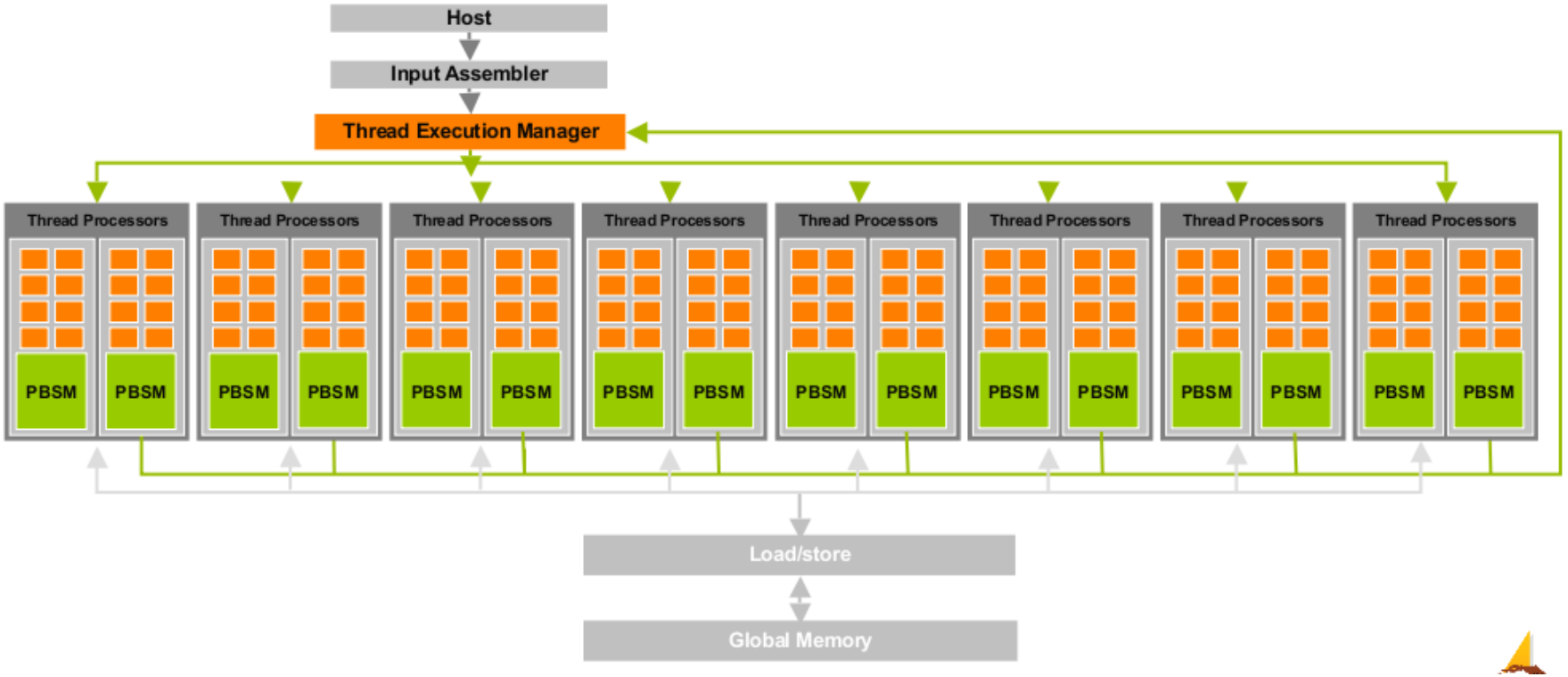
Graphics processing units (GPUs)

- A GPU consists of a scheduler, a large shared memory and several streaming multiprocessors (SMs)

Graphics processing units (GPUs)

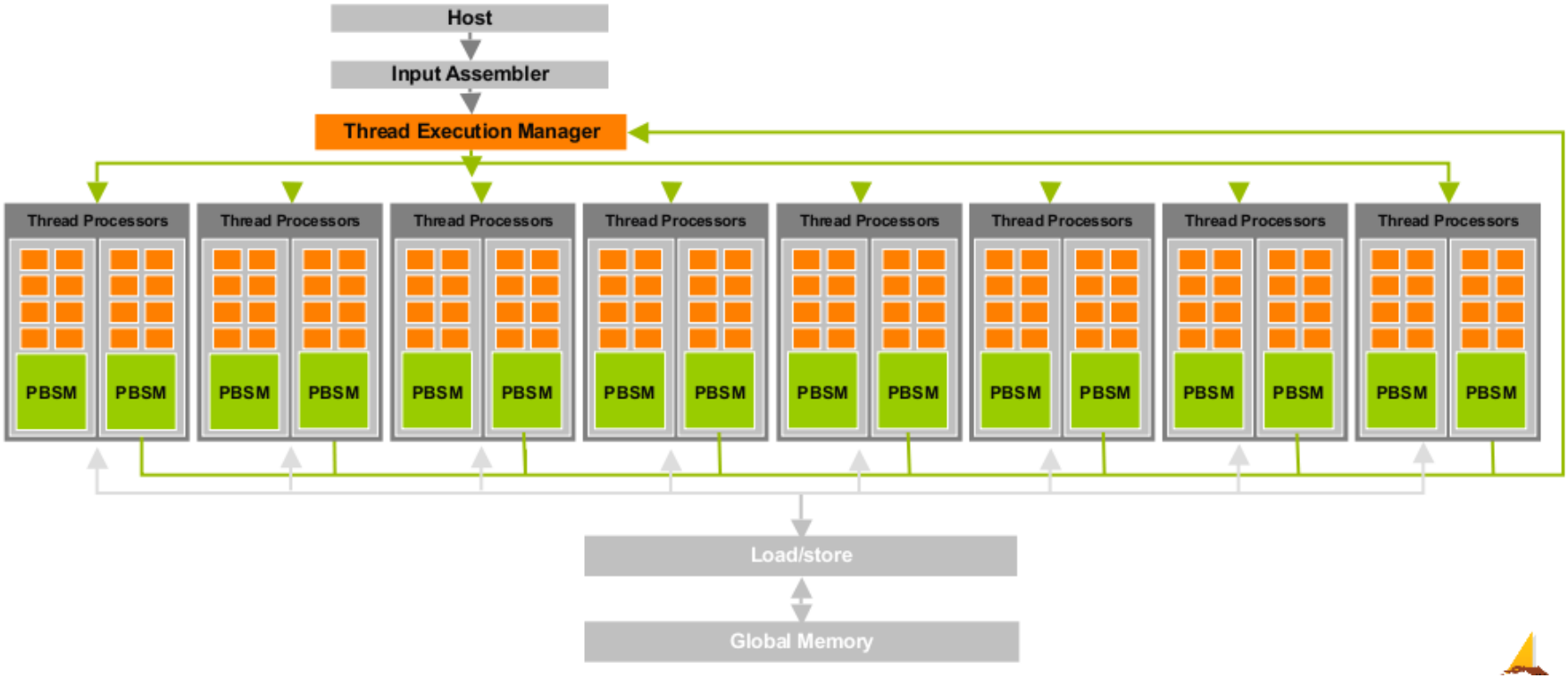
- A GPU consists of a scheduler, a large shared memory and several streaming multiprocessors (SMs)
- In addition, each SM has a local (private) small memory.

Graphics processing units (GPUs)



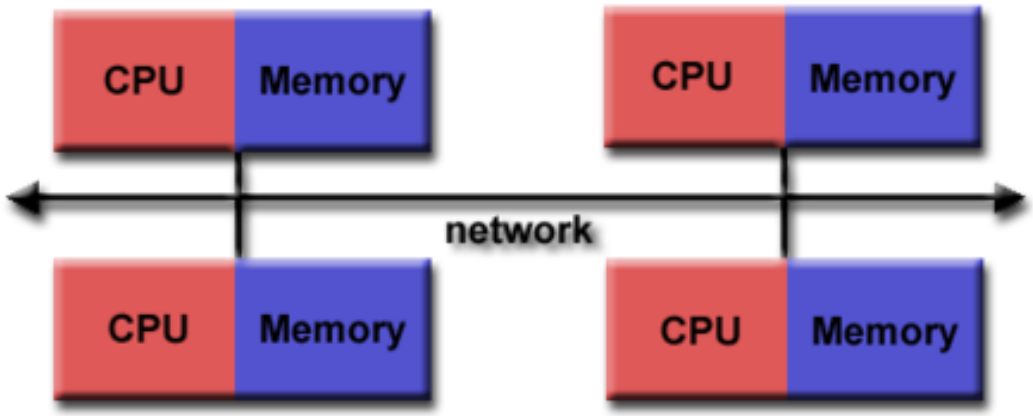
- In a GPU, the small local memories have much smaller access time than the large shared memory.

Graphics processing units (GPUs)



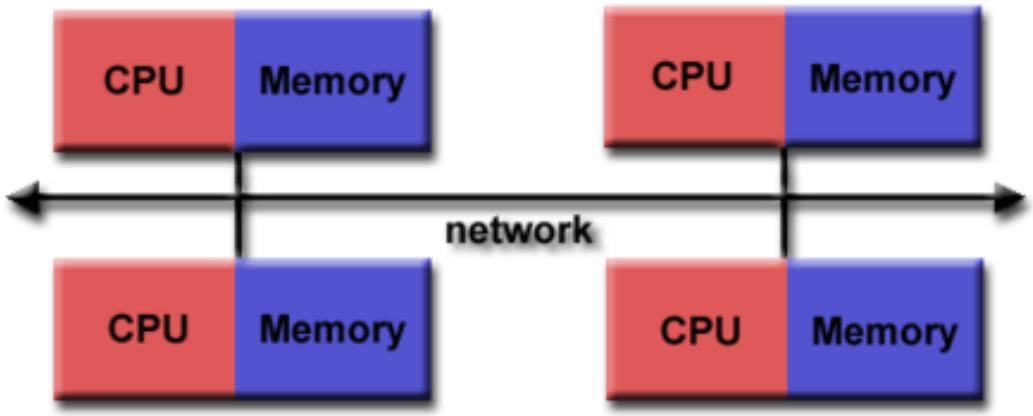
- In a GPU, the small local memories have much smaller access time than the large shared memory.
- Thus, as much as possible, cores access data in the local memories while the shared memory should essentially be used for data exchange between SMs.

Distributed Memory



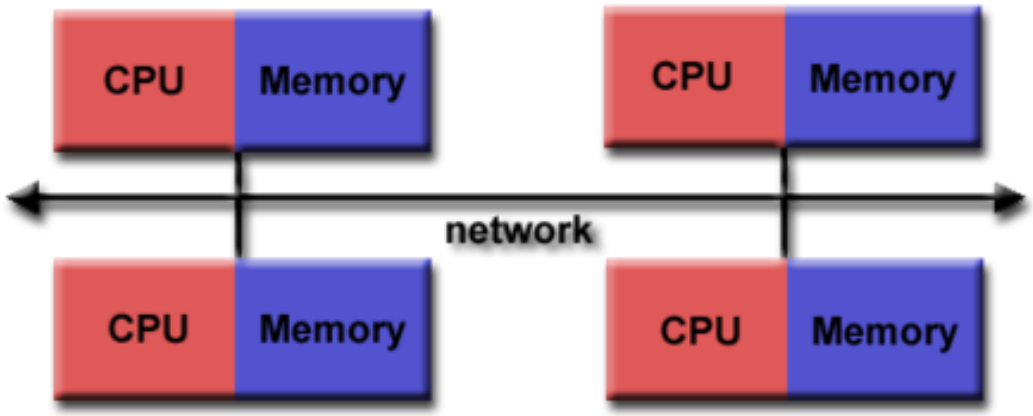
- Distributed memory systems require a communication network to connect inter-processor memory.

Distributed Memory



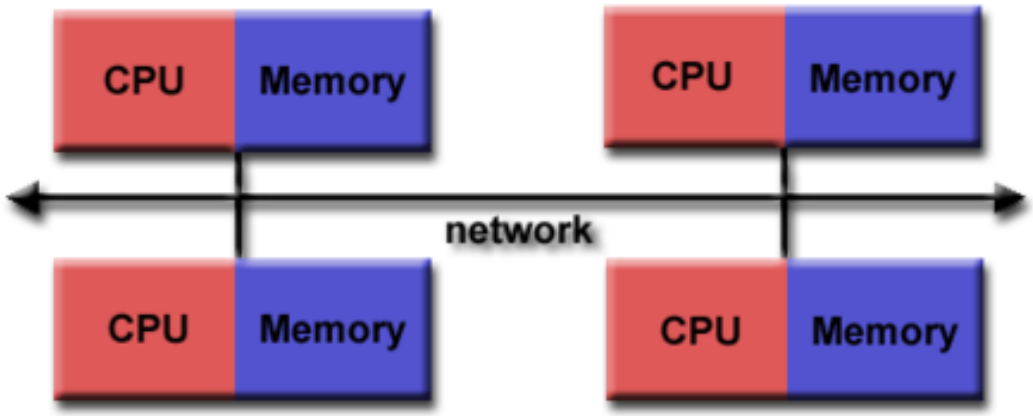
- Distributed memory systems require a communication network to connect inter-processor memory.
- Processors have their own local memory and operate independently.

Distributed Memory



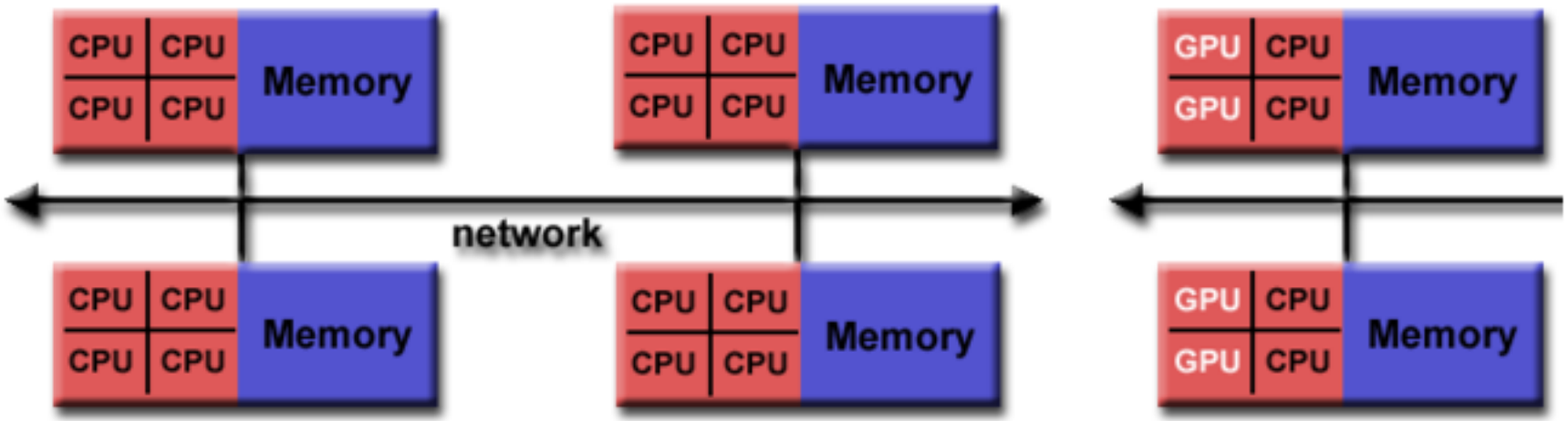
- Distributed memory systems require a communication network to connect inter-processor memory.
- Processors have their own local memory and operate independently.
- Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors.

Distributed Memory



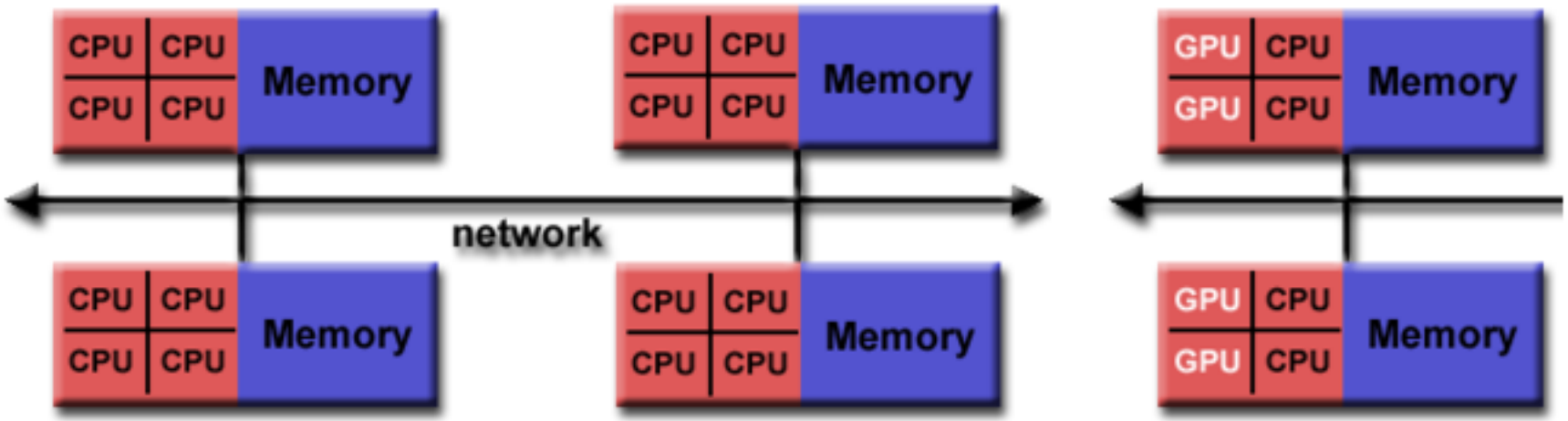
- Distributed memory systems require a communication network to connect inter-processor memory.
- Processors have their own local memory and operate independently.
- Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors.
- Data exchange between processors is managed by the programmer , not by the hardware.

Hybrid Distributed-Shared Memory



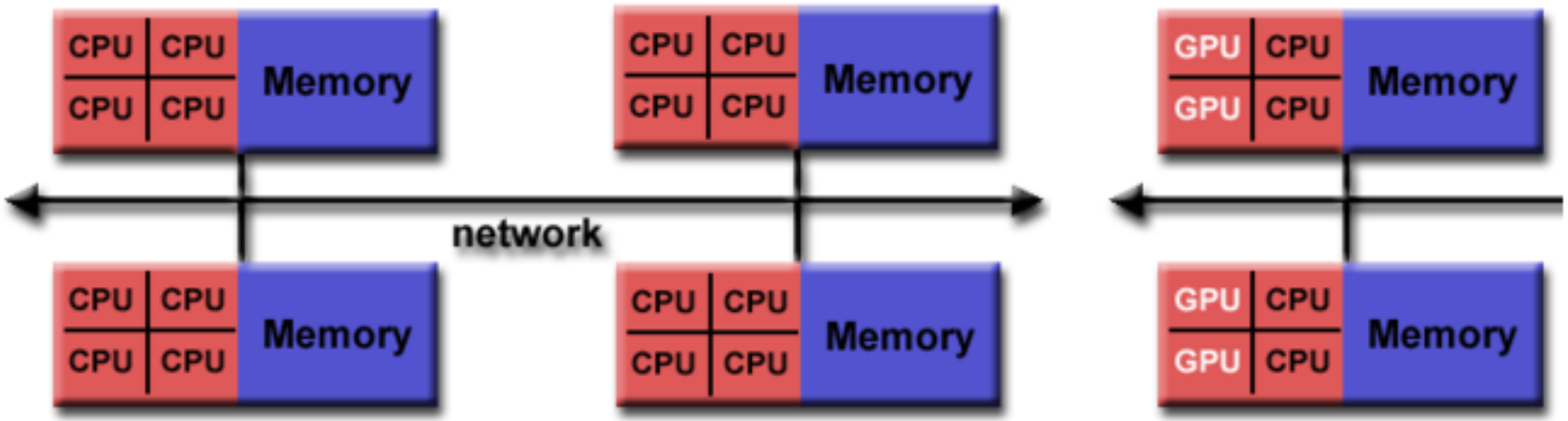
- The largest and fastest computers in the world today employ both shared and distributed memory architectures.

Hybrid Distributed-Shared Memory



- The largest and fastest computers in the world today employ both shared and distributed memory architectures.
- Current trends seem to indicate that this type of memory architecture will continue to prevail.

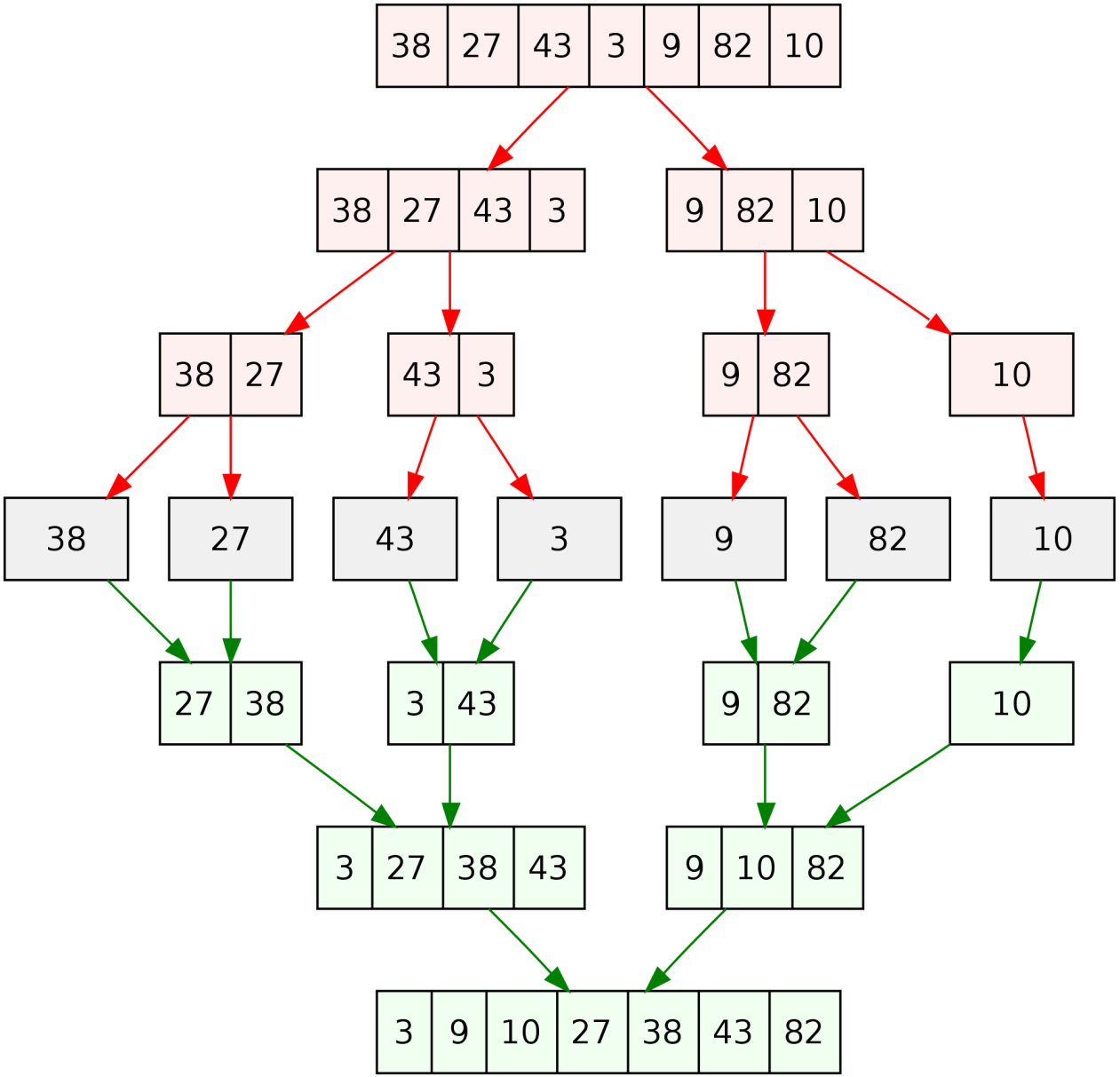
Hybrid Distributed-Shared Memory



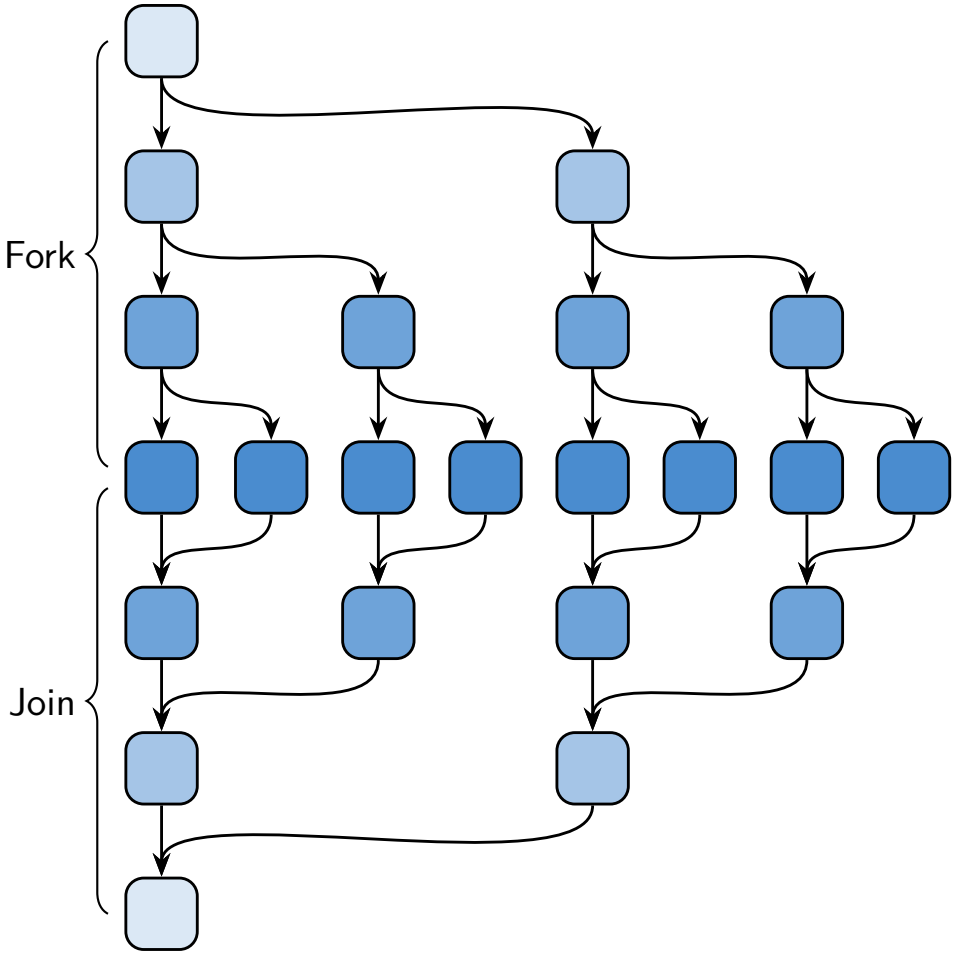
- The largest and fastest computers in the world today employ both shared and distributed memory architectures.
- Current trends seem to indicate that this type of memory architecture will continue to prevail.
- While this model allows for applications to scale, it increases the complexity of writing computer programs.

Outline

Divide-and-Conquer

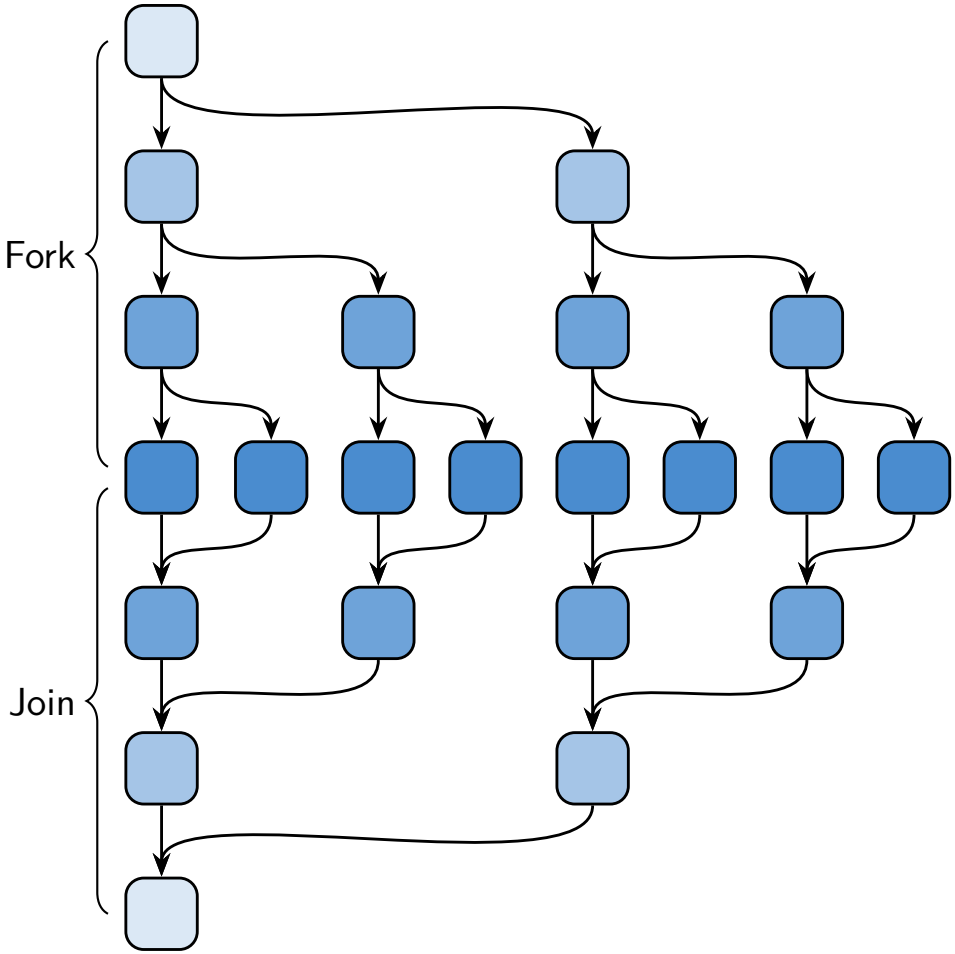


Divide-and-Conquer and Fork-Join



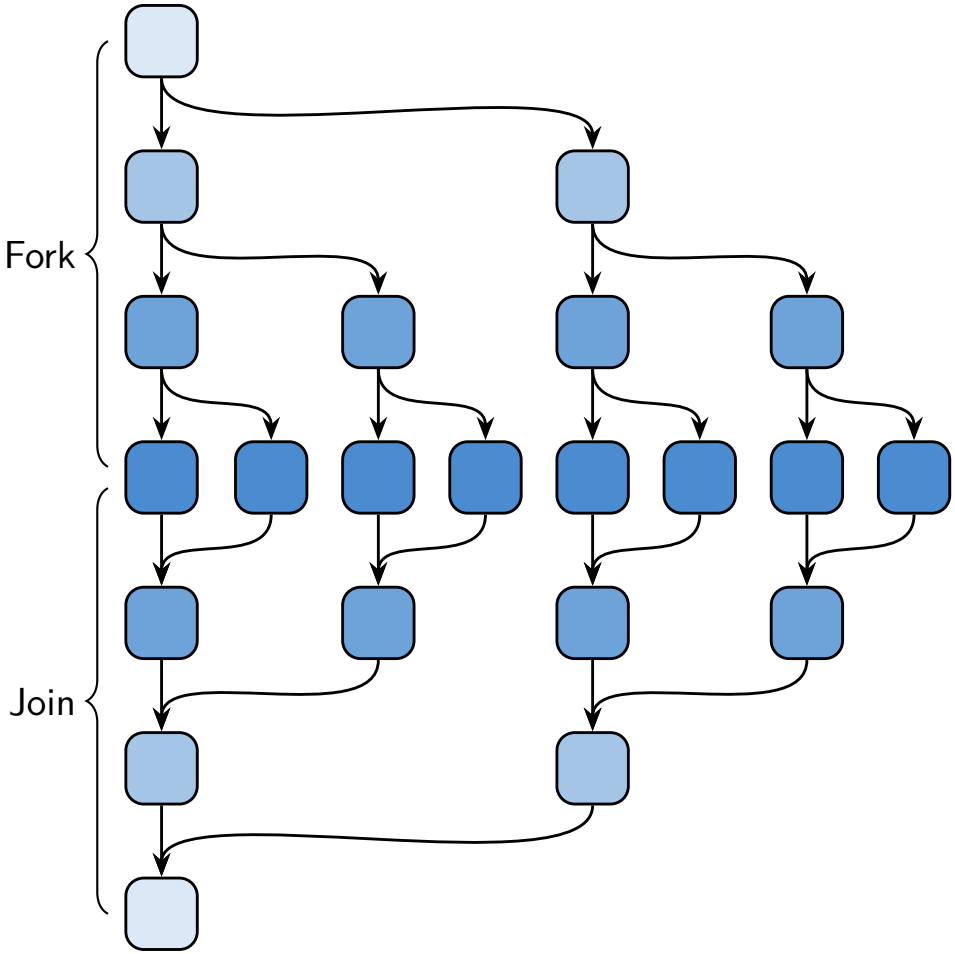
- **Fork**: divide problem and execute separate calls in parallel

Divide-and-Conquer and Fork-Join



- **Fork**: divide problem and execute separate calls in parallel
- **Join**: merge parallel execution back into serial

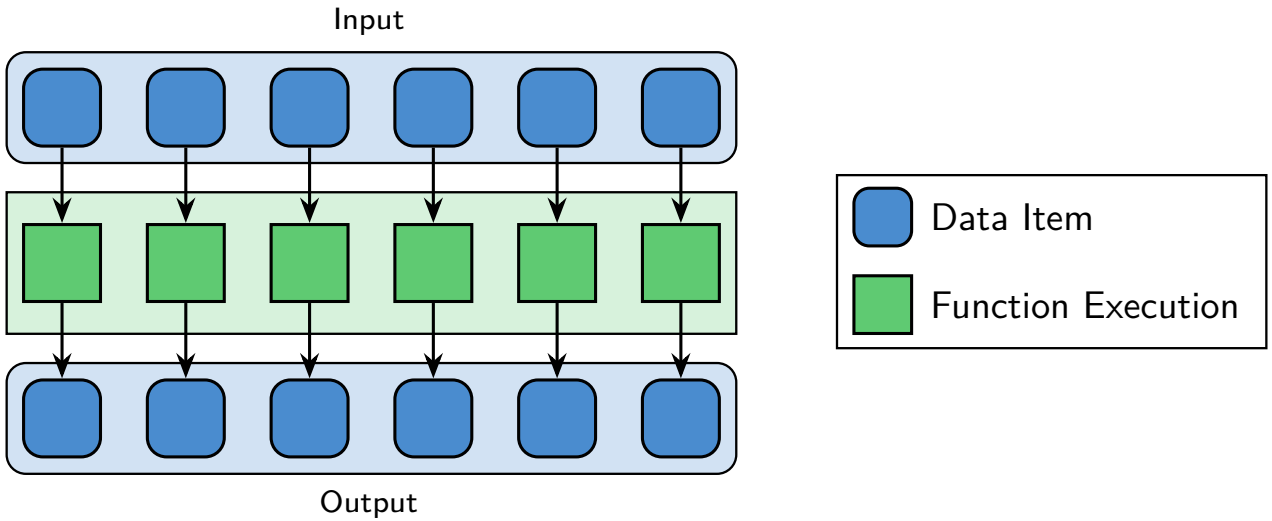
Divide-and-Conquer and Fork-Join



- **Fork**: divide problem and execute separate calls in parallel
- **Join**: merge parallel execution back into serial
- Recursively applying fork-join can "easily" parallelize a divide-and-conquer algorithm

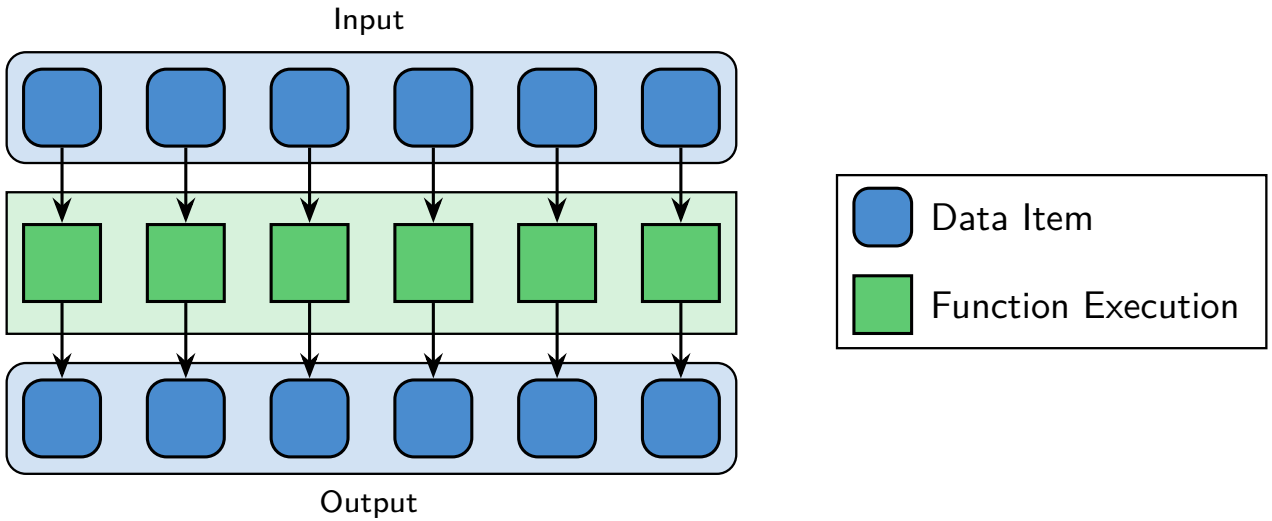
Map

- Simultaneously execute a function on each data item in a collection



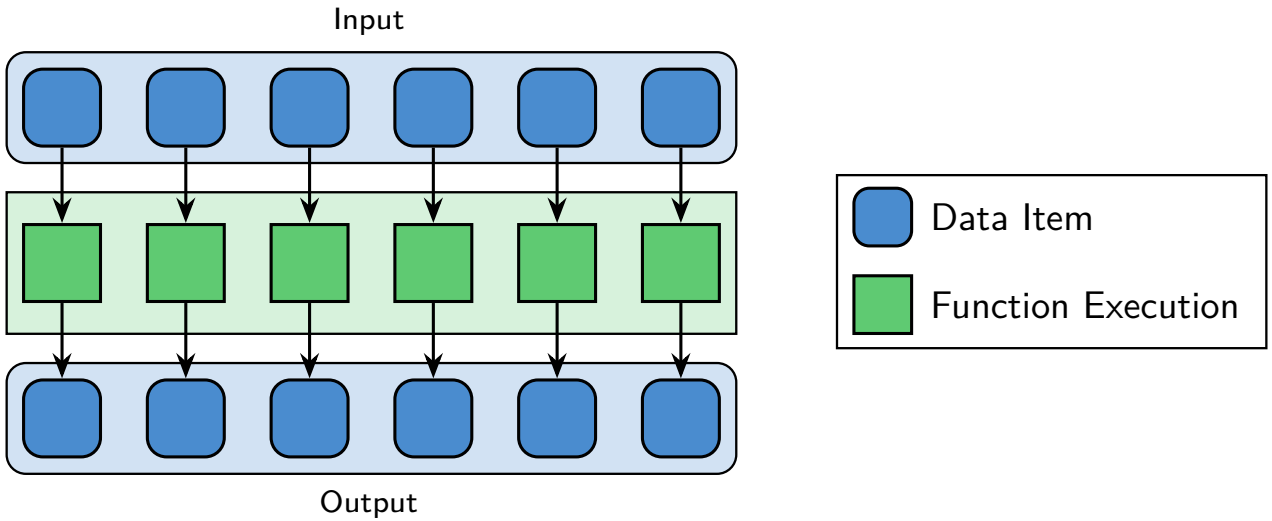
Map

- Simultaneously execute a function on each data item in a collection
- If more data items than threads, apply the pattern block-wise: (1) partition the collection, and (2) apply one thread to each part



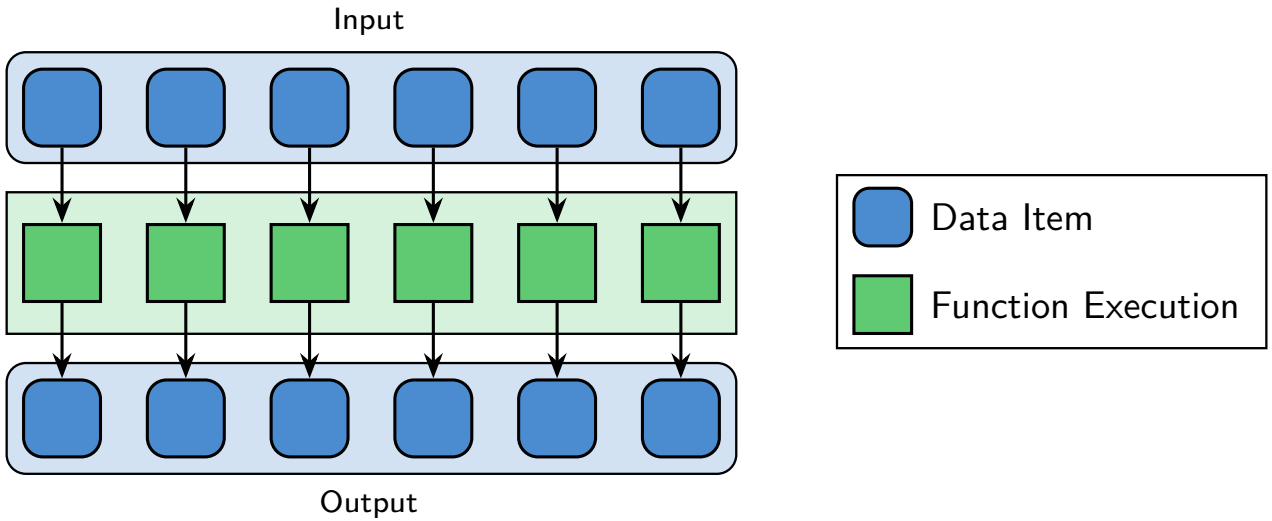
Map

- Simultaneously execute a function on each data item in a collection
- If more data items than threads, apply the pattern block-wise: (1) partition the collection, and (2) apply one thread to each part
- This pattern is often simplified as just a `parallel_for` loop



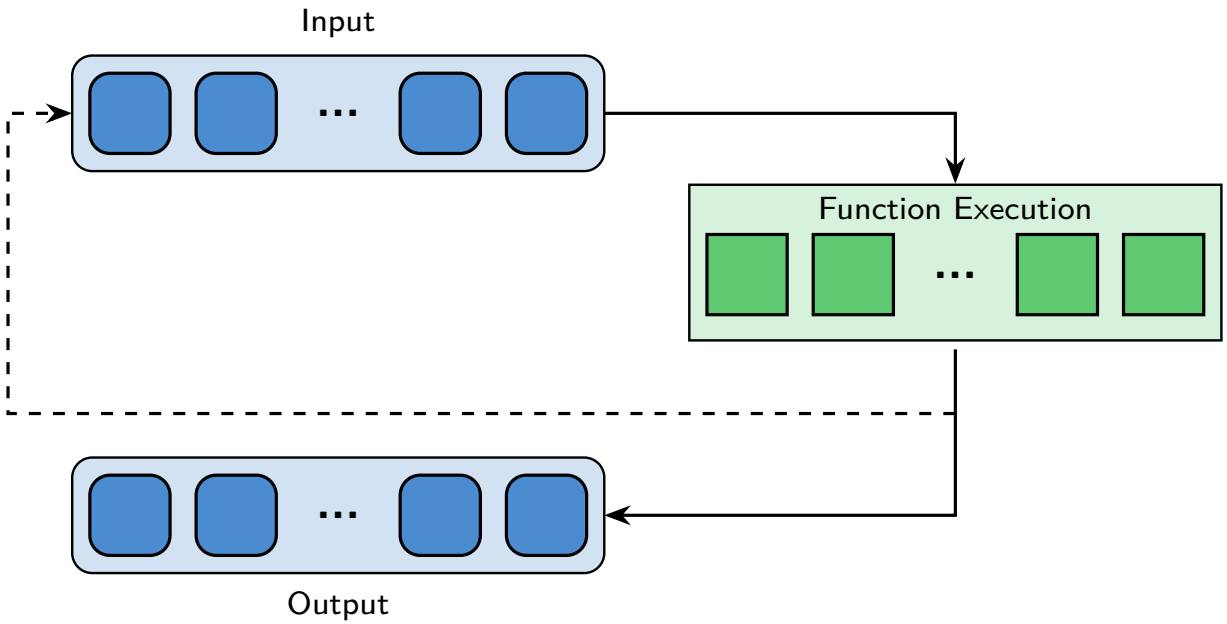
Map

- Simultaneously execute a function on each data item in a collection
- If more data items than threads, apply the pattern block-wise:
  (1) partition the collection, and (2) apply one thread to each part
- This pattern is often simplified as just a `parallel_for` loop
- Where multiple map steps are performed in a row,
  they may operate in lockstep



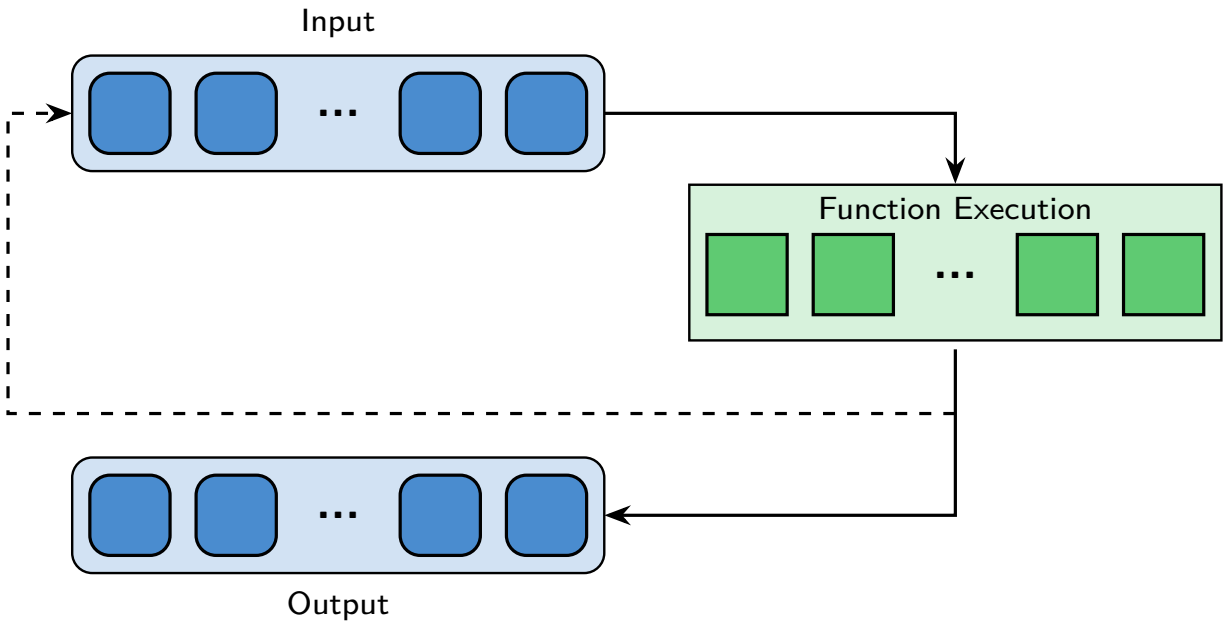
Workpile

- Workpile generalizes map pattern to a *queue* of tasks



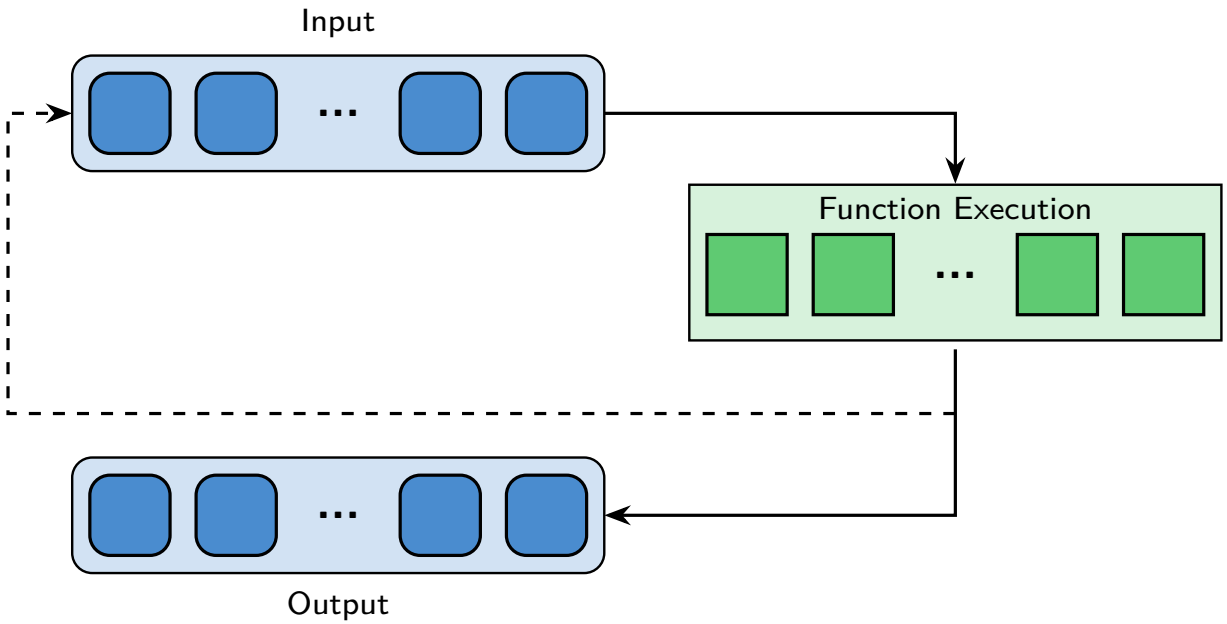
Workpile

- Workpile generalizes map pattern to a *queue* of tasks
- Tasks in-flight can add new tasks to input queue



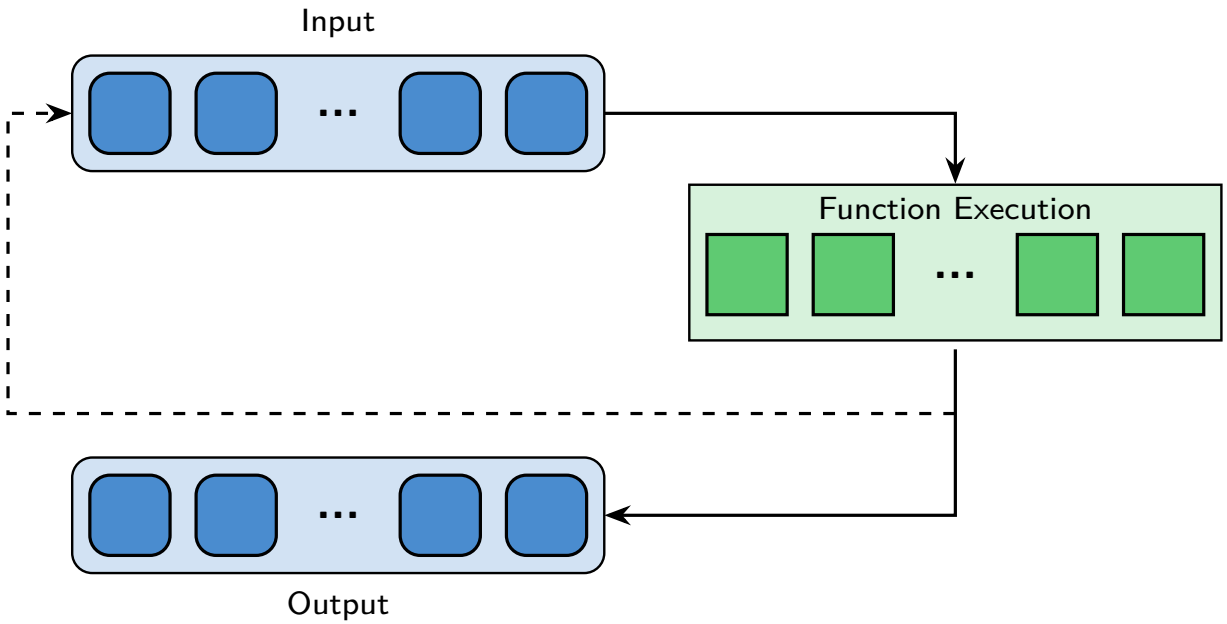
Workpile

- Workpile generalizes map pattern to a *queue* of tasks
- Tasks in-flight can add new tasks to input queue
- Threads take tasks from queue until it is empty

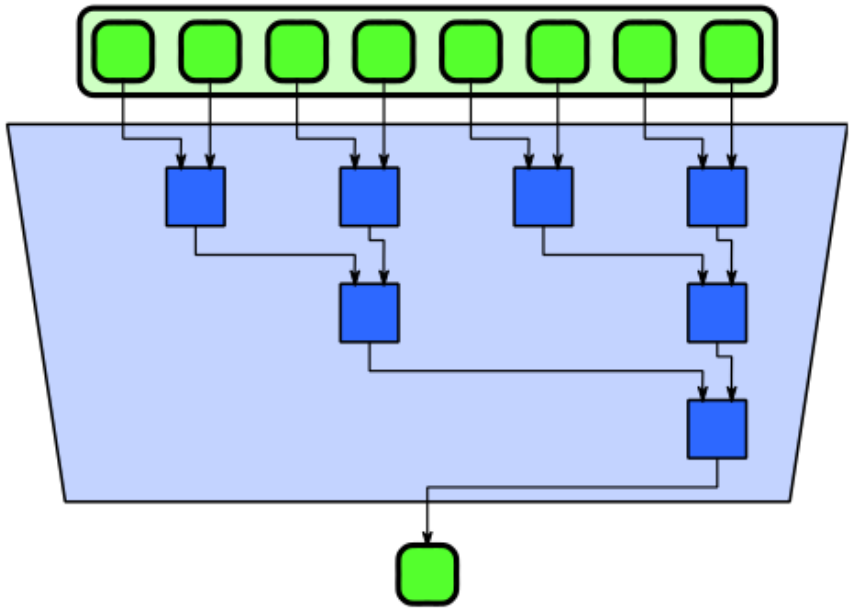


Workpile

- Workpile generalizes map pattern to a *queue* of tasks
- Tasks in-flight can add new tasks to input queue
- Threads take tasks from queue until it is empty
- Can be seen as a **`parallel_while`** loop

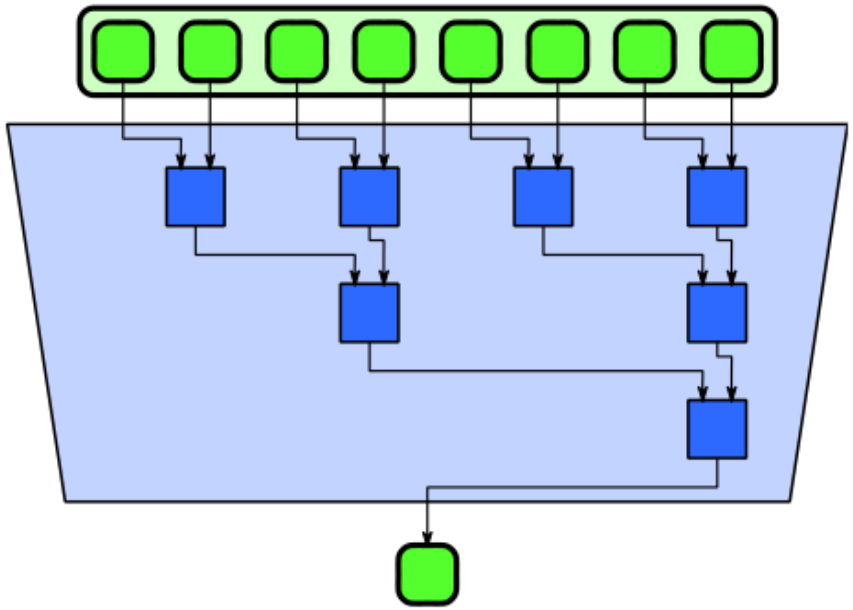


Reduction



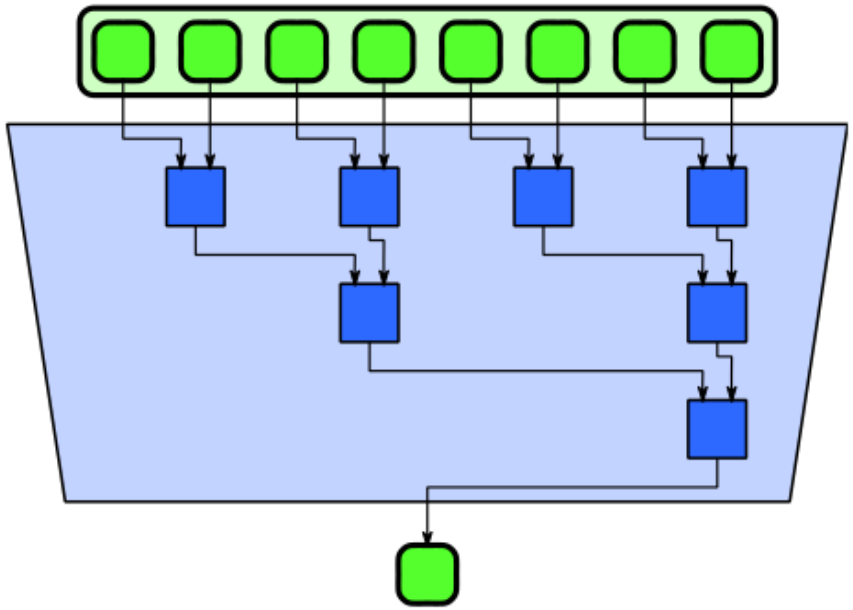
- A reduction combines every element in a collection into one element, using an associative operator.

Reduction



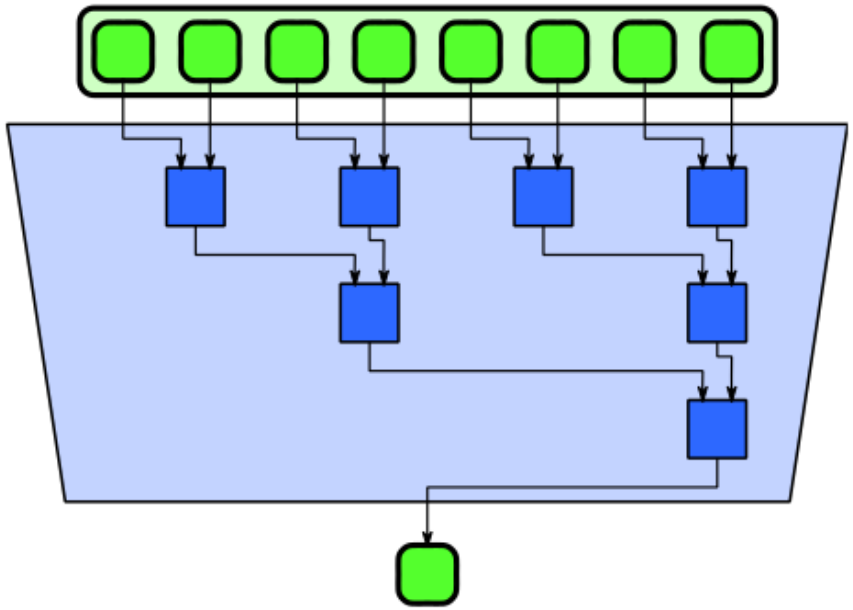
- A reduction combines every element in a collection into one element, using an associative operator.
- Example: computing the sum (or product) of n matrices.

Reduction



- A reduction combines every element in a collection into one element, using an associative operator.
- Example: computing the sum (or product) of n matrices.
- Grouping the operations is often needed to allow for parallelism.

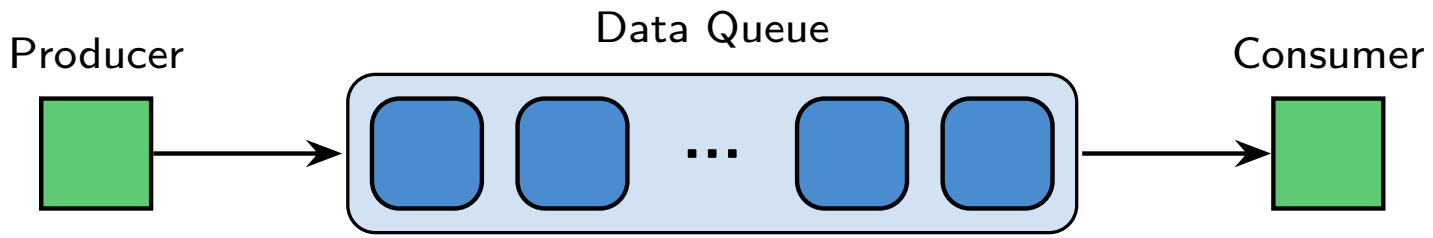
Reduction



- A reduction combines every element in a collection into one element, using an associative operator.
- Example: computing the sum (or product) of n matrices.
- Grouping the operations is often needed to allow for parallelism.
- This grouping requires associativity, but not commutativity.

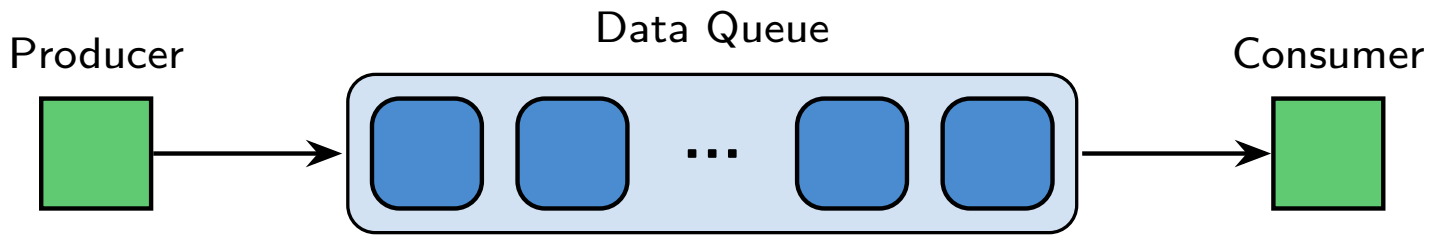
Producer-Consumer

- Two functions connected by a queue



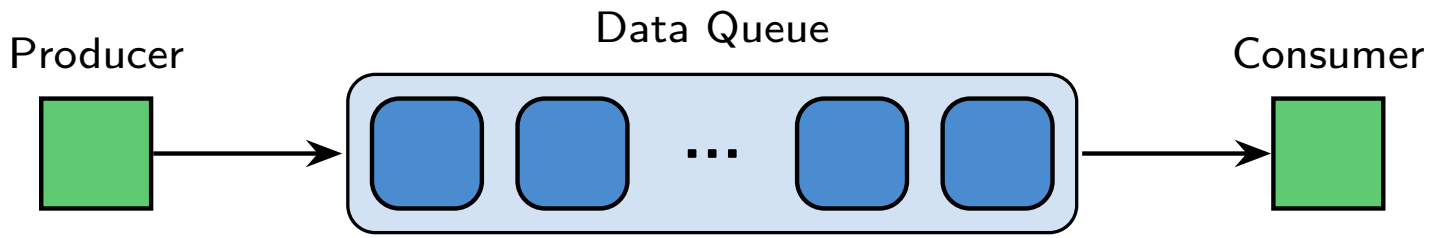
Producer-Consumer

- Two functions connected by a queue
- The producer produces data items, pushing them to the queue



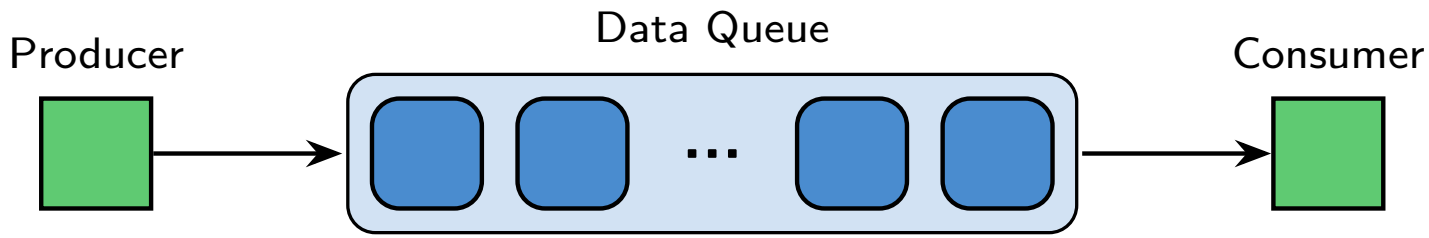
Producer-Consumer

- Two functions connected by a queue
- The producer produces data items, pushing them to the queue
- The consumer processes data items, pulling them from the queue



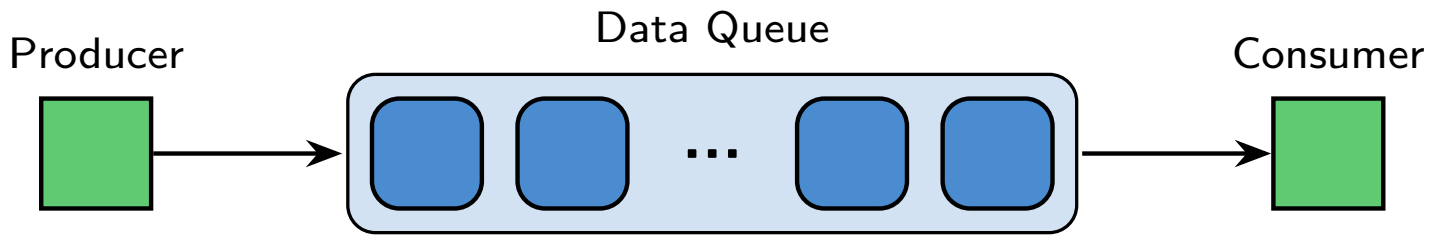
Producer-Consumer

- Two functions connected by a queue
- The producer produces data items, pushing them to the queue
- The consumer processes data items, pulling them from the queue
- Producer and consumer execute simultaneously; at least one must be active at all times $\Longrightarrow$ no **deadlock**



Producer-Consumer

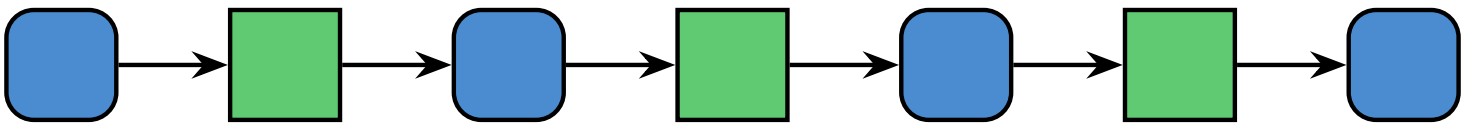
- Two functions connected by a queue
- The producer produces data items, pushing them to the queue
- The consumer processes data items, pulling them from the queue
- Producer and consumer execute simultaneously; at least one must be active at all times $\Longrightarrow$ no **deadlock**



- In some circumstances, the producer may be considered as an **iterator** or **generator**

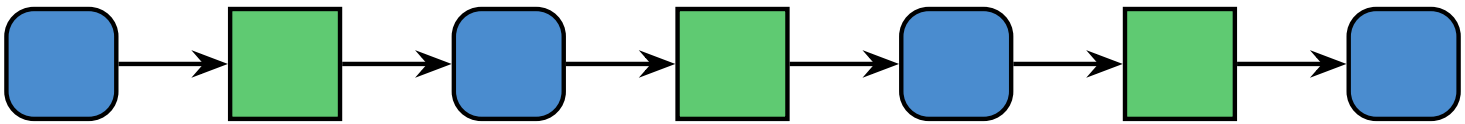
Pipeline

- A sequence of stages where the output of one stage is used as the input to another



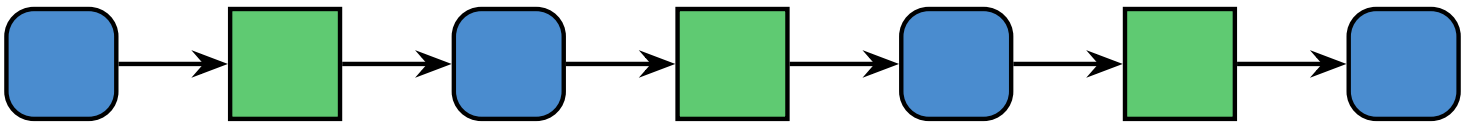
Pipeline

- A sequence of stages where the output of one stage is used as the input to another
- Example: in a pipelined processor, instructions flow through the central processing unit (CPU) in stages (Instruction Fetch, Decode, Execute, etc.)



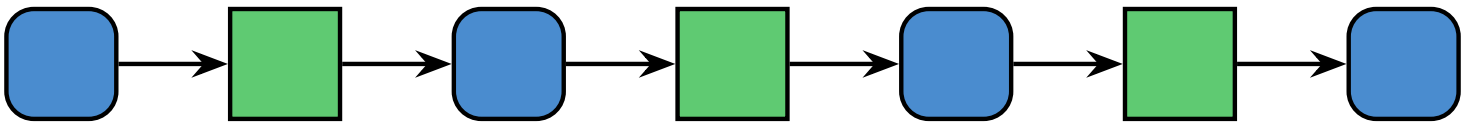
Pipeline

- A sequence of stages where the output of one stage is used as the input to another
- Example: in a pipelined processor, instructions flow through the central processing unit (CPU) in stages (Instruction Fetch, Decode, Execute, etc.)
- Two consecutive stages form a producer-consumer pair



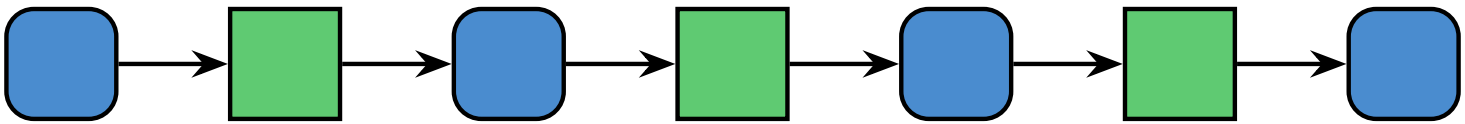
Pipeline

- A sequence of stages where the output of one stage is used as the input to another
- Example: in a pipelined processor, instructions flow through the central processing unit (CPU) in stages (Instruction Fetch, Decode, Execute, etc.)
- Two consecutive stages form a producer-consumer pair
- Internal stages are both producer and consumer



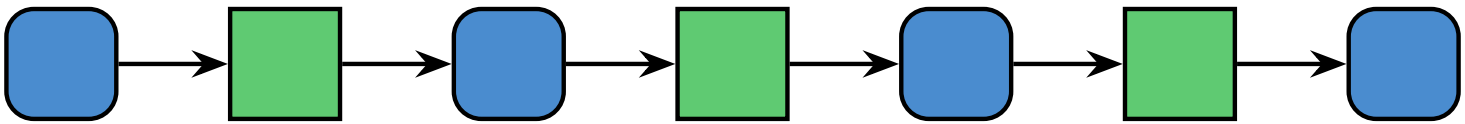
Pipeline

- A sequence of stages where the output of one stage is used as the input to another
- Example: in a pipelined processor, instructions flow through the central processing unit (CPU) in stages (Instruction Fetch, Decode, Execute, etc.)
- Two consecutive stages form a producer-consumer pair
- Internal stages are both producer and consumer
- Typically, a pipeline is constructed statically through code organization

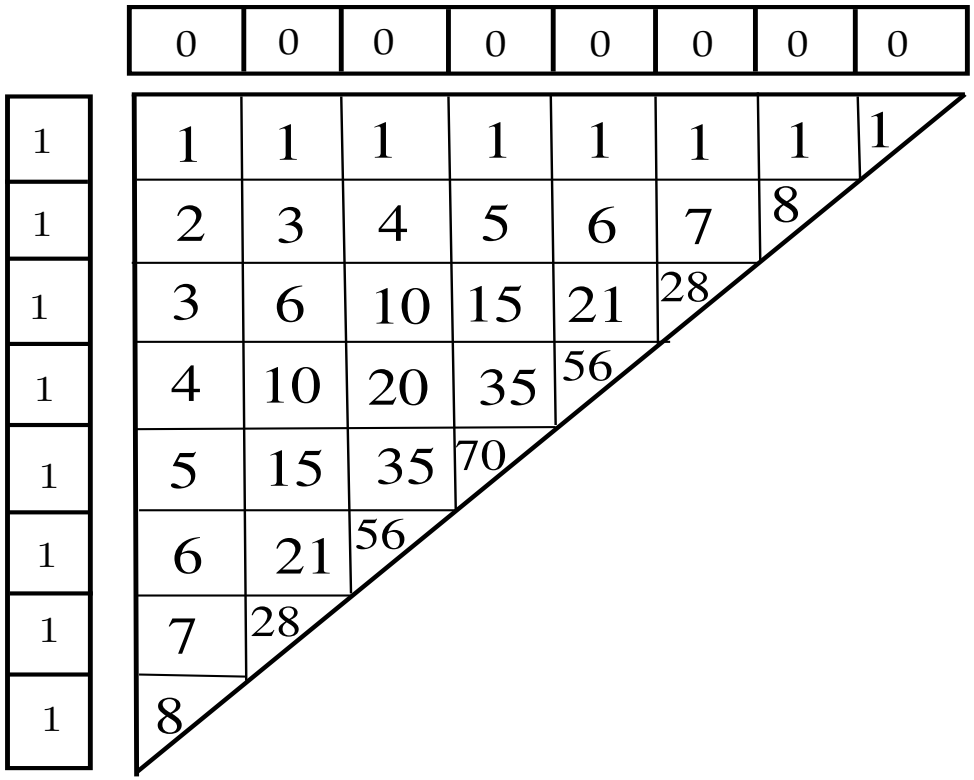


Pipeline

- A sequence of stages where the output of one stage is used as the input to another
- Example: in a pipelined processor, instructions flow through the central processing unit (CPU) in stages (Instruction Fetch, Decode, Execute, etc.)
- Two consecutive stages form a producer-consumer pair
- Internal stages are both producer and consumer
- Typically, a pipeline is constructed statically through code organization
- Pipelines can be created dynamically and implicitly with `AsyncGenerators` and the call-stack

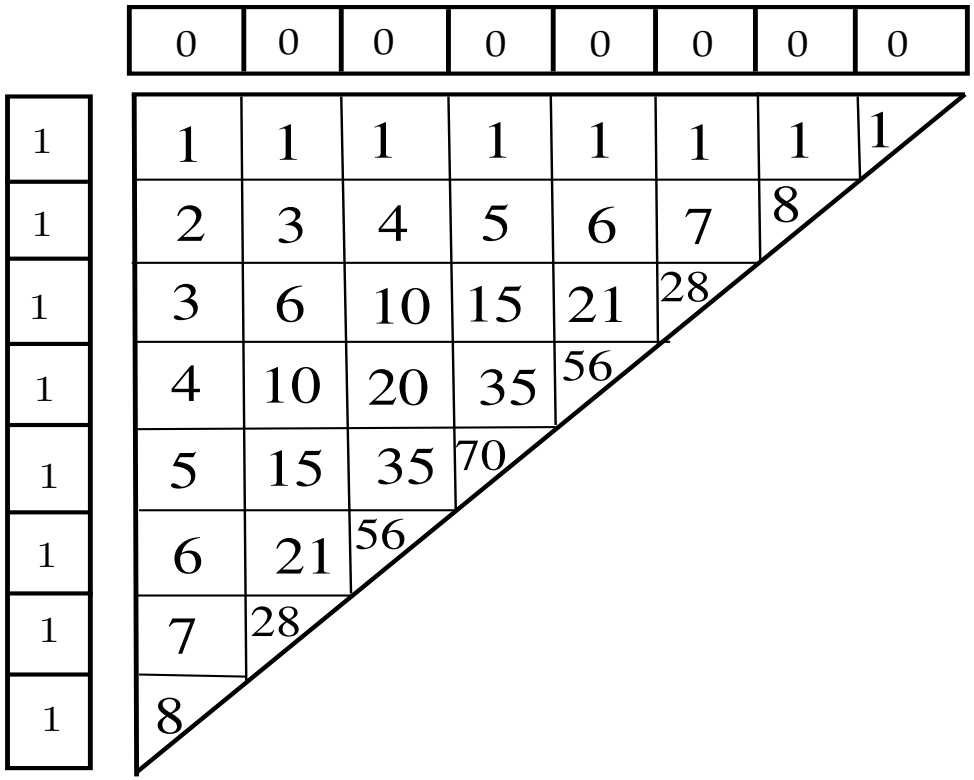


Pascal triangle construction: a stencil computation



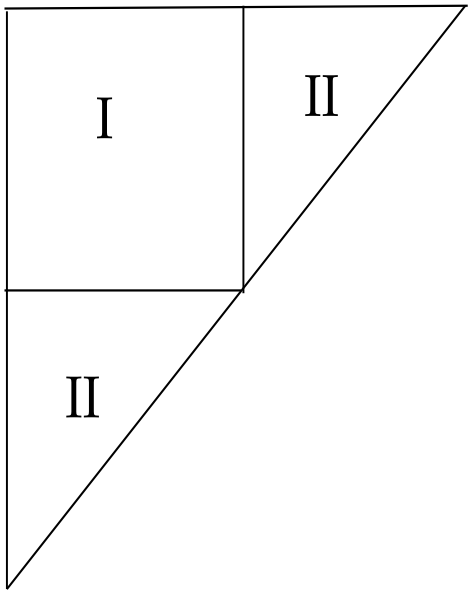
- Stencil computations are a class of data processing techniques which update array elements according to a pattern

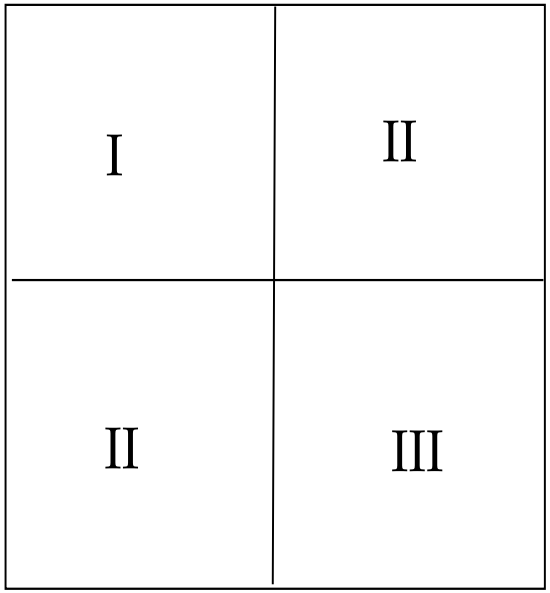
Pascal triangle construction: a stencil computation



- Stencil computations are a class of data processing techniques which update array elements according to a pattern
- Construction of the Pascal Triangle: nearly the simplest stencil computation!

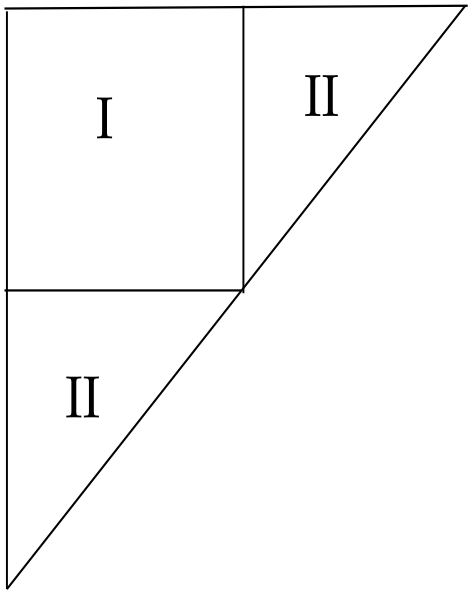
Divide and conquer: principle

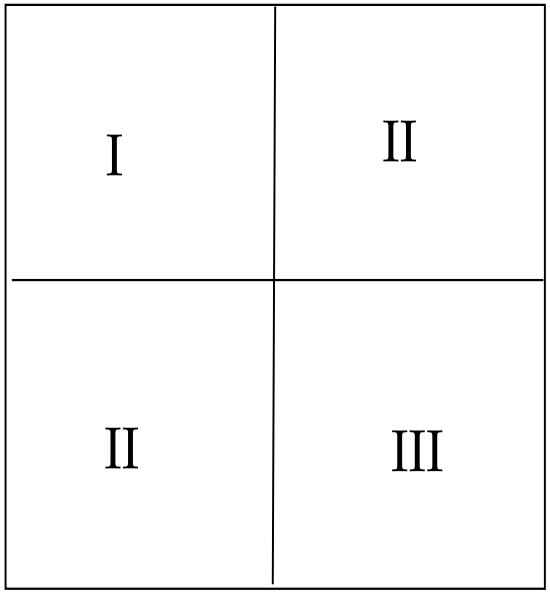




- Each triangle region can be computed as a square region followed by two (concurrent) triangle regions.

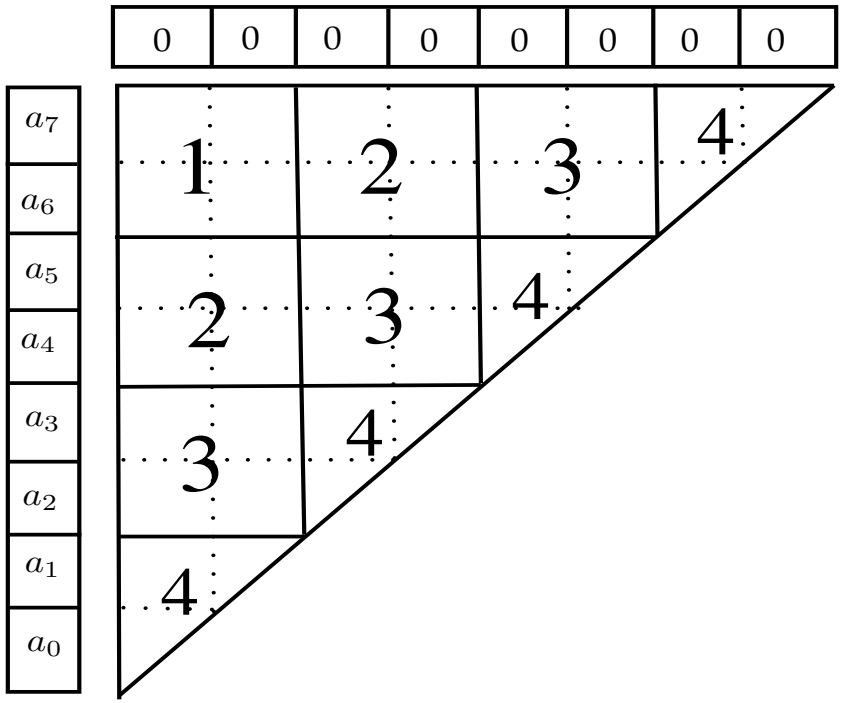
Divide and conquer: principle





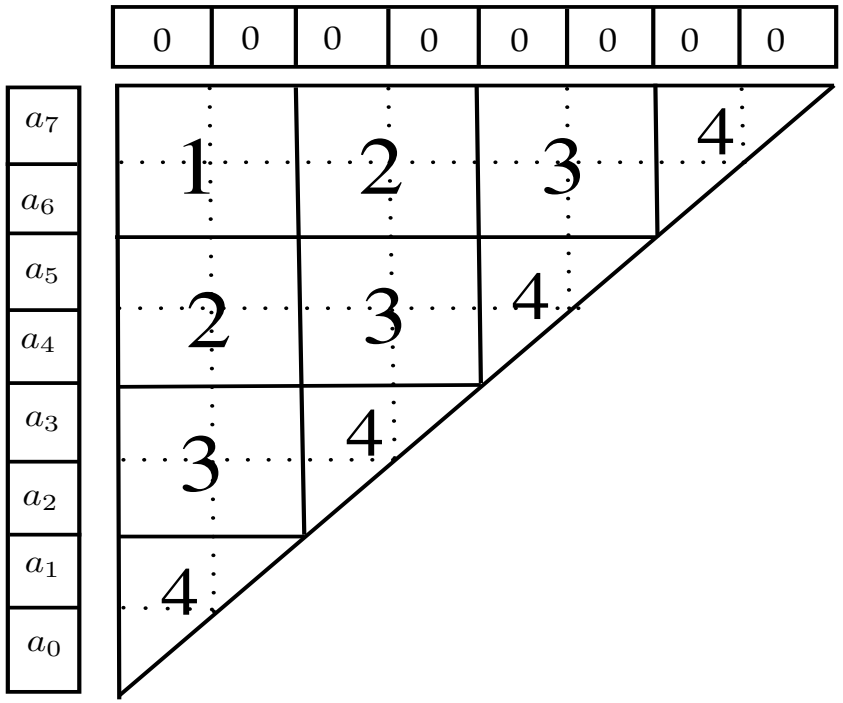
- Each triangle region can be computed as a square region followed by two (concurrent) triangle regions.
- Each square region can also be computed in a divide and conquer manner.

Blocking strategy: principle



- Let B be the order of a block and n be the number of elements.

Blocking strategy: principle



- Let B be the order of a block and n be the number of elements.
- Each block is processed serially (as a task) and the set of all blocks is computed concurrently.

Outline

1. Hardware architecture and concurrency

2. Parallel programming patterns

3. Concurrency platforms

Programming patterms in Julia

```
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.7.1 (2021-12-22)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia> map(x -> x * 2, [1, 2, 3])
3-element Vector{Int64}:
 2
 4
 6

julia> mapreduce(x->x^2, +, [1:3;])
14
```

Julia

```
function pmap(f, lst)
    np = nprocs()  # the number of processes available
    n = length(lst)
    results = Vector{Any}(n)
    i = 1
    # function to produce the next work item from the queue.
    nextidx() = (idx=i; i+=1; idx)
    @sync begin
        for p=1:np
            if p != myid() || np == 1
                @async begin
                    while true
                        idx = nextidx()
                        if idx > n
                            break
                        end
                        results[idx]=remotecall_fetch(f,p,lst[idx])
                    end
                end
            end
        end
    end
    results
end
```

Fork-Join with Cilk

```
int fib(int n)
{
   if (n < 2) return n;
   int x, y;
   x = cilk_spawn fib(n-1);
   y = fib(n-2);
   cilk_sync;
   return x+y;
}
```

- The named child function `cilk_spawn fib(n-1)` may execute in parallel with its parent
- `Cilk` keywords `cilk_spawn` and `cilk_sync` grant permissions for parallel execution. They do not command parallel execution.
- Visit `https://www.opencilk.org/`

Cilk

- `Cilk` has been developed since 1994 at the MIT Laboratory for Computer Science by Prof. Charles E. Leiserson and his group, in particular by Matteo Frigo and Tao B. Schardl

Cilk

- `Cilk` has been developed since 1994 at the MIT Laboratory for Computer Science by Prof. Charles E. Leiserson and his group, in particular by Matteo Frigo and Tao B. Schardl
- `Cilk` is a multithreaded language for parallel programming that generalizes the semantics of `C` (resp. `C++`) by introducing linguistic constructs for parallel control.

Cilk

- `Cilk` has been developed since 1994 at the MIT Laboratory for Computer Science by Prof. Charles E. Leiserson and his group, in particular by Matteo Frigo and Tao B. Schardl
- `Cilk` is a multithreaded language for parallel programming that generalizes the semantics of `C` (resp. `C++`) by introducing linguistic constructs for parallel control.
- `Cilk` is a faithful extension of `C` (resp. `C++`). That is, the `C` (resp. `C++`) elision of a `Cilk` program is a correct implementation of the semantics of that program.

Cilk

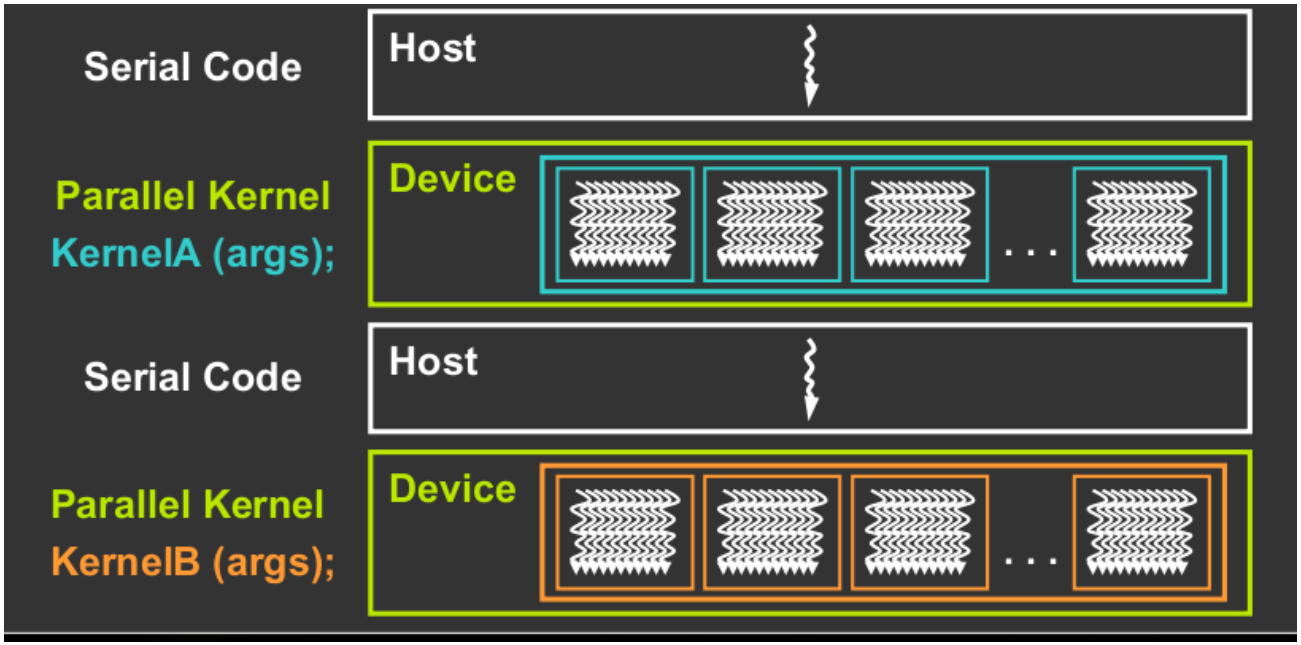
- `Cilk` has been developed since 1994 at the MIT Laboratory for Computer Science by Prof. Charles E. Leiserson and his group, in particular by Matteo Frigo and Tao B. Schardl
- `Cilk` is a multithreaded language for parallel programming that generalizes the semantics of `C` (resp. `C++`) by introducing linguistic constructs for parallel control.
- `Cilk` is a faithful extension of `C` (resp. `C++`). That is, the `C` (resp. `C++`) elision of a `Cilk` program is a correct implementation of the semantics of that program.
- `Cilk`'s scheduler maps strands onto processors dynamically at runtime, using the *work-stealing principle*. Under reasonable assumptions, this provives a guarantee of performance.

Cilk

- `Cilk` has been developed since 1994 at the MIT Laboratory for Computer Science by Prof. Charles E. Leiserson and his group, in particular by Matteo Frigo and Tao B. Schardl
- `Cilk` is a multithreaded language for parallel programming that generalizes the semantics of `C` (resp. `C++`) by introducing linguistic constructs for parallel control.
- `Cilk` is a faithful extension of `C` (resp. `C++`). That is, the `C` (resp. `C++`) elision of a `Cilk` program is a correct implementation of the semantics of that program.
- `Cilk`'s scheduler maps strands onto processors dynamically at runtime, using the *work-stealing principle*. Under reasonable assumptions, this provives a guarantee of performance.
- `Cilk` has supporting tools for data race (thus non-deterministic behaviour) detection and performance analysis.

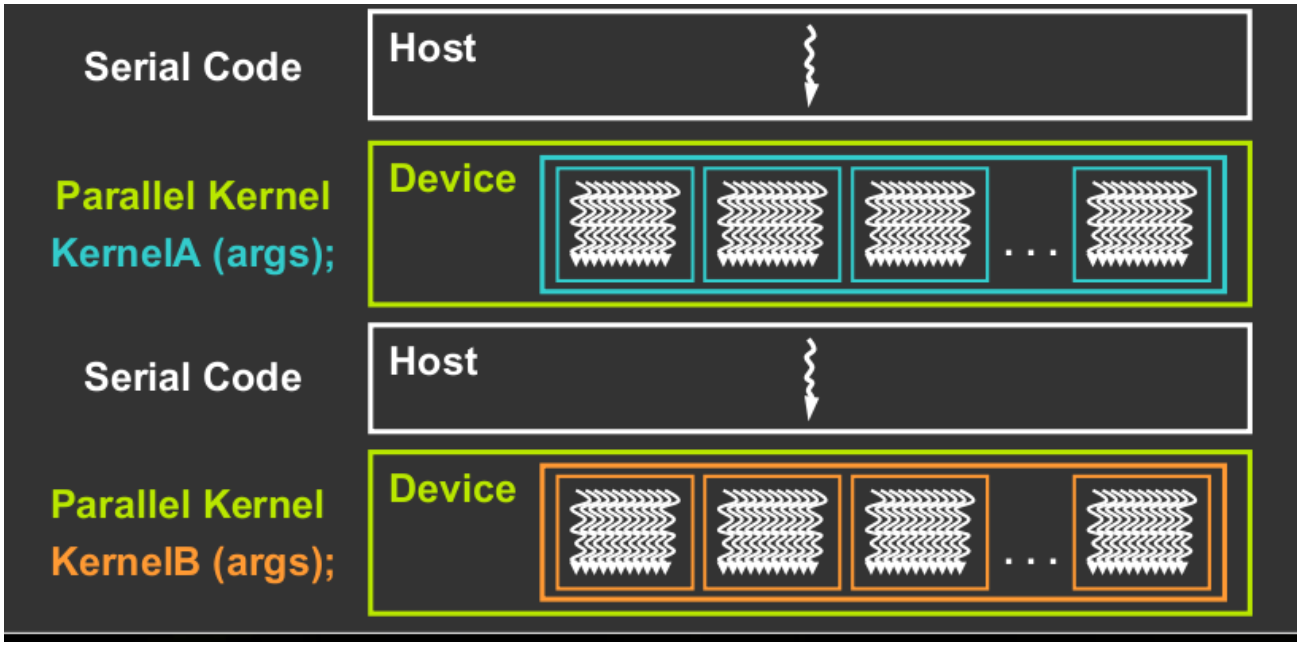
Heterogeneous programming with CUDA

- The parallel code is written for a thread



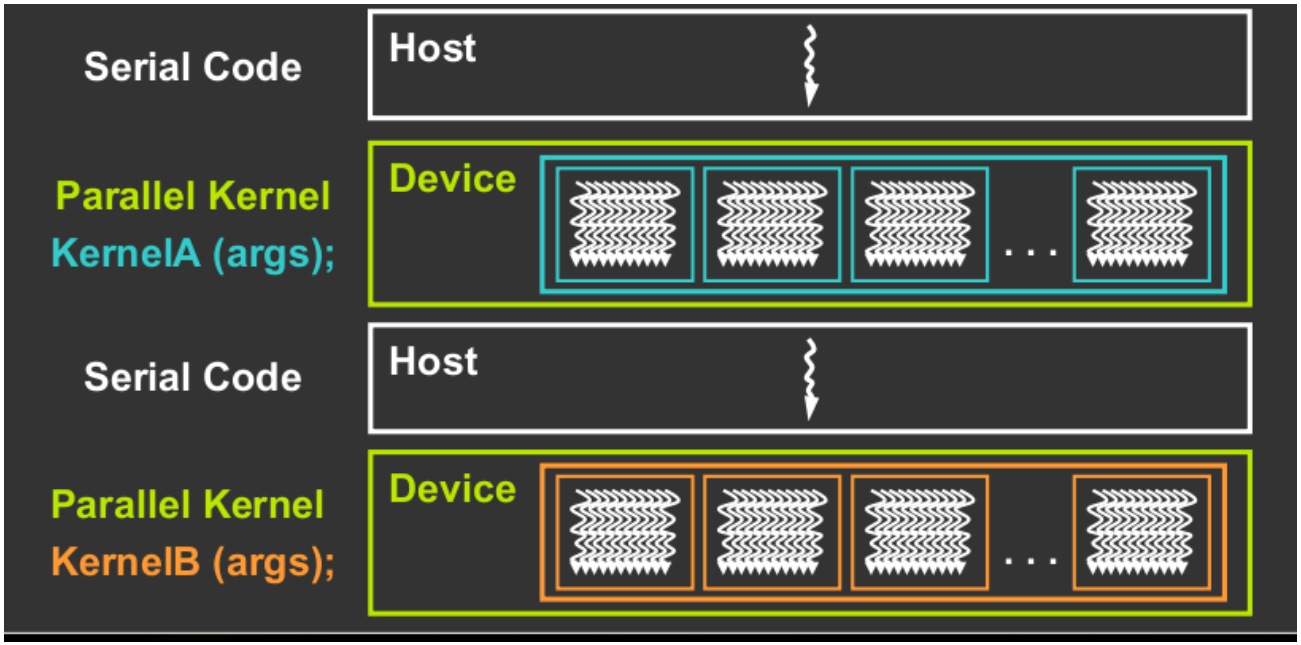
Heterogeneous programming with CUDA

- The parallel code is written for a thread
  - ↳ Each thread is free to execute a unique code path



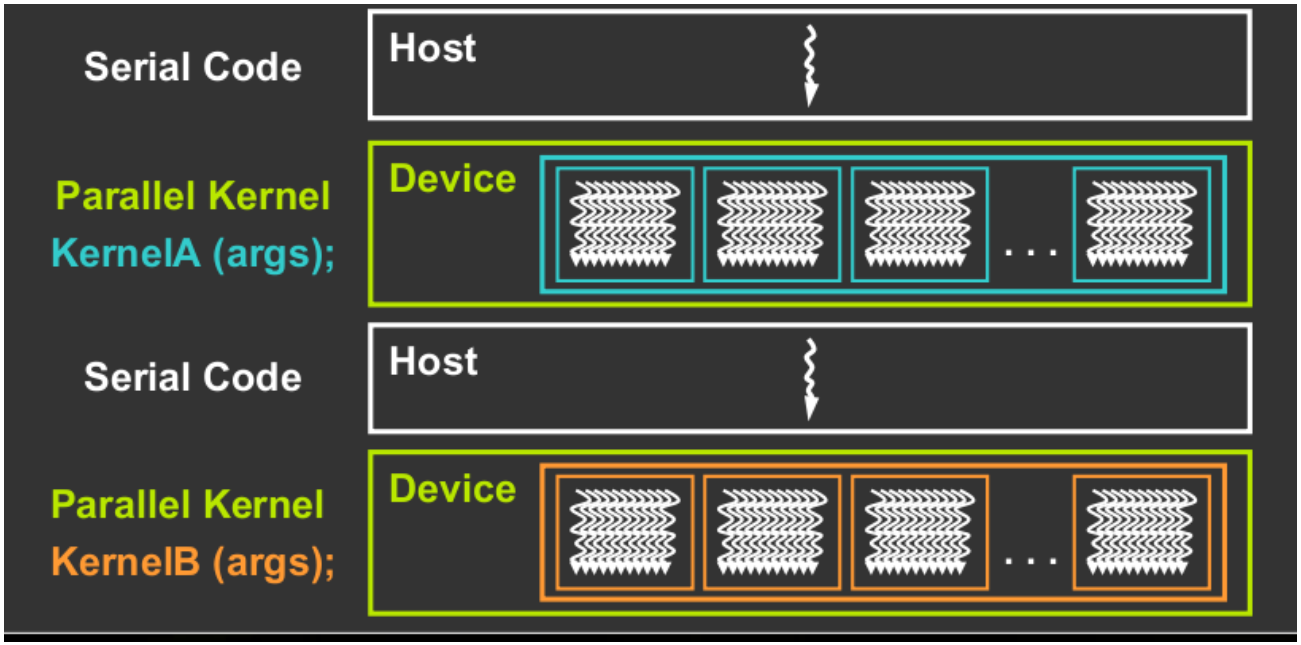
Heterogeneous programming with CUDA

- The parallel code is written for a thread
  - ↳ Each thread is free to execute a unique code path
  - ↳ Built-in **thread and block ID variables** are used to map each thread to a specific data tile (see next slide).



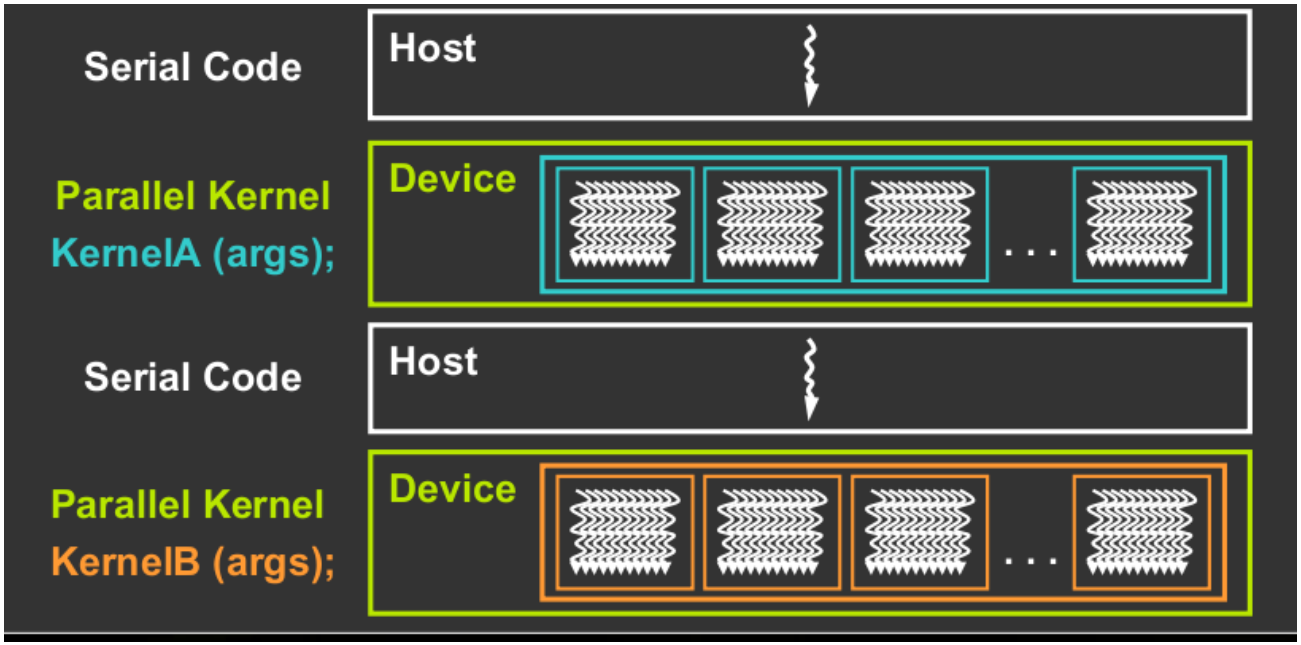
Heterogeneous programming with CUDA

- The parallel code is written for a thread
  - ↳ Each thread is free to execute a unique code path
  - ↳ Built-in **thread and block ID variables** are used to map each thread to a specific data tile (see next slide).
- Thus, each thread executes the same code.



Heterogeneous programming with CUDA

- The parallel code is written for a thread
  - ↳ Each thread is free to execute a unique code path
  - ↳ Built-in **thread and block ID variables** are used to map each thread to a specific data tile (see next slide).
- Thus, each thread executes the same code.
- However, different threads work on different data, based on their thread and block IDs.



CUDA Example: increment array elements (1/2)

Increment N-element vector a by scalar b

Let's assume N=16, blockDim=4 -> 4 blocks

```
int idx = blockDim.x * blockId.x + threadIdx.x;
```

blockIdx.x=0	blockIdx.x=1	blockIdx.x=2	blockIdx.x=3
blockDim.x=4	blockDim.x=4	blockDim.x=4	blockDim.x=4
threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3
idx=0,1,2,3	idx=4,5,6,7	idx=8,9,10,11	idx=12,13,14,15

CUDA Example: increment array elements (2/2)

CPU program

```
void increment_cpu(float *a, float b, int N)
{
    for (int idx = 0; idx<N; idx++)
        a[idx] = a[idx] + b;
}


void main()
{
  .....
    increment_cpu(a, b, N);
}
```

CUDA program

```
__global__ void increment_gpu(float *a, float b, int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if( idx < N)
        a[idx] = a[idx] + b;
}


void main()
{
  .....
    dim3 dimBlock (blocksize);
    dim3 dimGrid( ceil( N / (float)blocksize) );
    increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```

References

[1] M. McCool, J. Reinders, and A. Robison. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[2] J. E. Savage. *Models of computation - exploring the power of computing*. Addison-Wesley, 1998. ISBN: 978-0-201-89539-1.

[3] M. L. Scott. *Programming Language Pragmatics (3. ed.)* Academic Press, 2009. ISBN: 978-0-12-374514-9.

[4] A. Williams. *C++ concurrency in action: practical multithreading; 1st ed.* Shelter Island, NY: Manning Publ., 2012. URL: https://cds.cern.ch/record/1483005.