GPU COMPUTING

@ VU, June 2k24

Ana-Lucia Varbanescu
a.l.varbanescu@utwente.nl
a.l.vabanescu@uva.nl

About me …

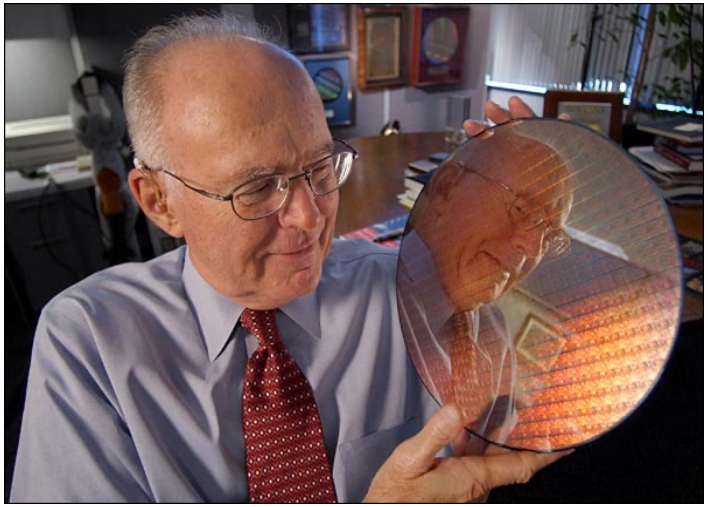
- BSc and MSc in Computer Science and Engineering from POLITEHNICA University of Bucharest, Romania
- PhD in parallel processing (including GPUs) from TUDelft
- Associate professor at UvA (joined in 2013)
  - Research in parallel processing, heterogeneous computing, perfomrance engineering
  - (relevant other) Teaching:
    - Programming Multi- and Many-core Systems
    - Performance Engineering
- Currently: professor at UTwente (joined in 2022)
- Please call me Ana ☺
  - Email: a.l.varbanescu@utwente.nl

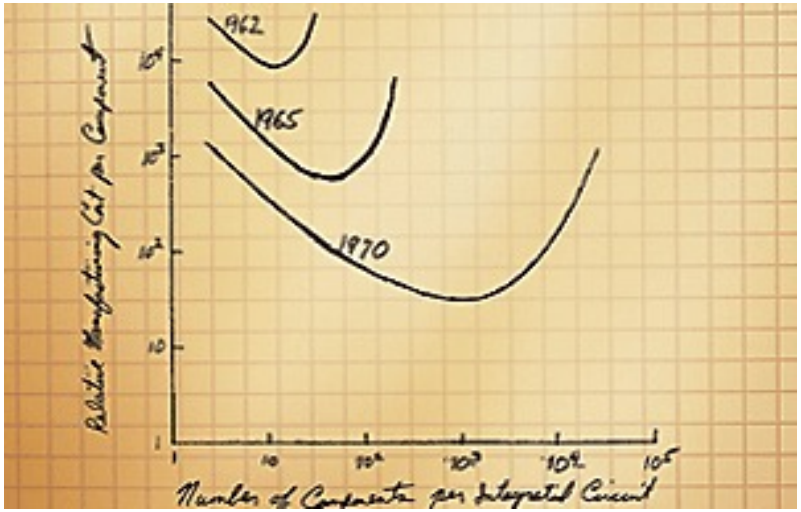
MULTI-/MANY-CORES

Shared memory machines

Moore's Law

- Gordon Moore (co-founder of Intel) predicted in 1965 that the transistor density of semiconductor chips would double roughly every 18 months.





"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year ... Certainly over the short term this rate can be expected to continue, if not to increase...." Electronics Magazine 1965

Traditionally …

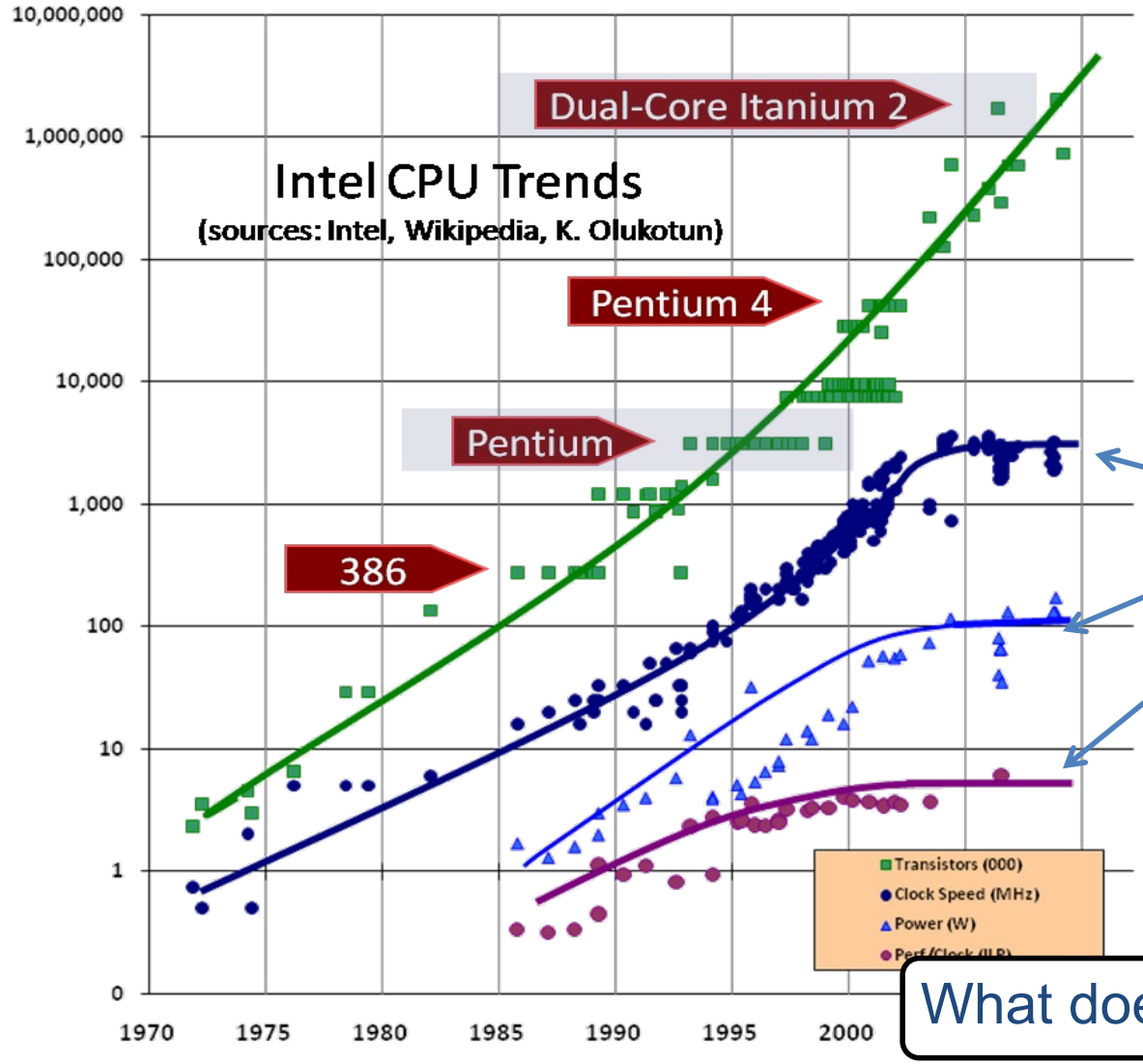
- More transistors = more functionality
- Improved technology = faster clocks = more speed

- Thus, every 18 months, we obtained better and faster processors.

- They were all sequential: they execute one operation per clock cycle.

Not anymore!
We no longer gain performance by “growing” sequential processors …

Evolution of processors



Chip density can increase about 2x every 2 years

BUT

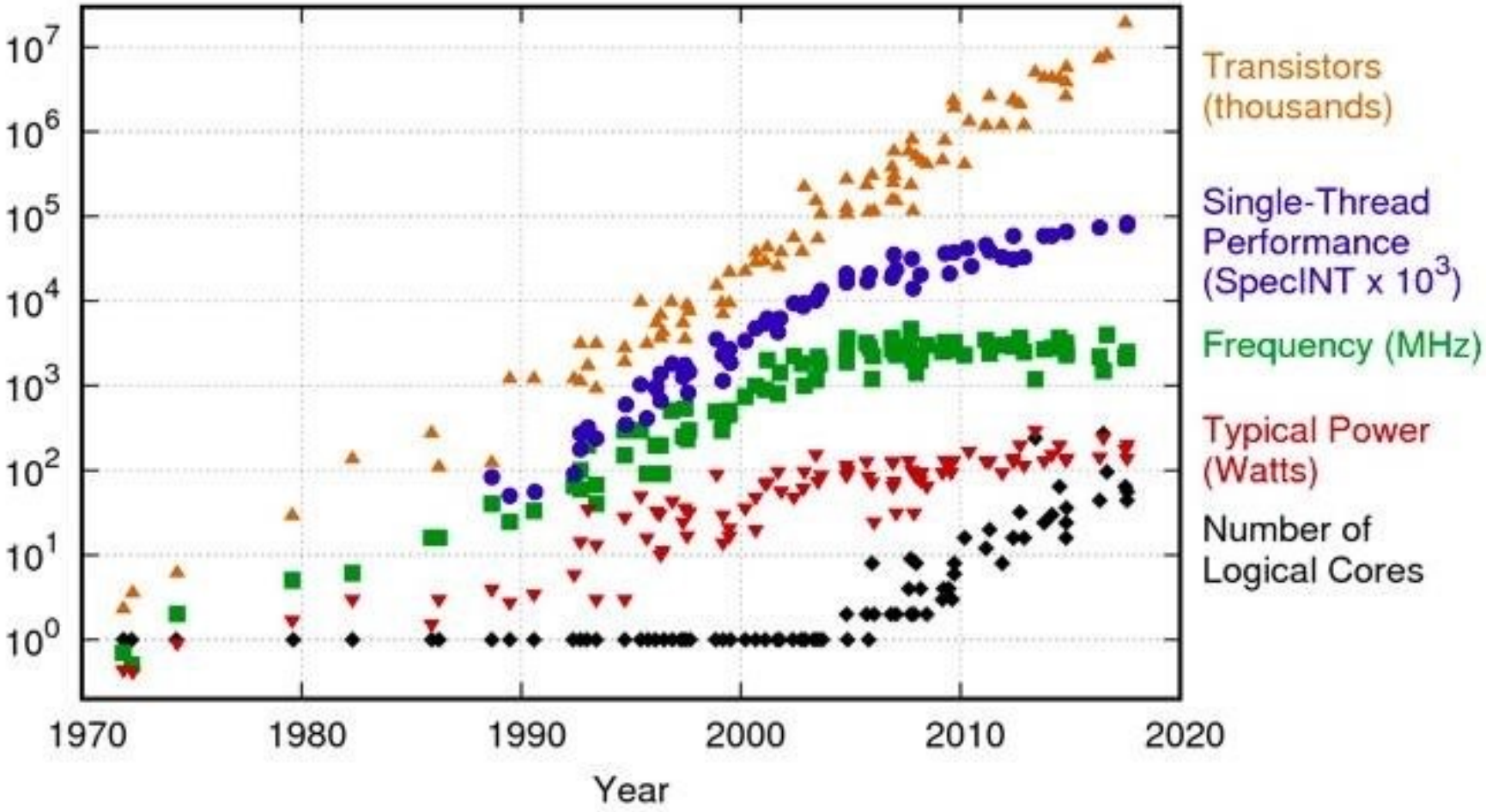
- **Clock speed is not**
- **Power is not**
- **Instruction Level Parallelism is not**

What does this mean in practice?

New ways to use transistors

Improve PERFORMANCE by using parallelism on-chip: multi-core (CPUs) and many-core processors (GPUs).

The shift to multi-core



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

Parallelism ⇔ HPC

- Parallelism is **mandatory** for **high performance**
  - Yesterday: **clusters (and grids)**
  - Today: **multi-/many-core processors**
  - Tomorrow: **massive multi-scale heterogeneous parallelism = clusters using different types of multi-/many-cores**

>95% of computing systems today are parallel!

Main challenges: learn to program parallel machines and learn to use them efficiently!

Parallelism & challenges

- Parallel execution
  - Two or more applications/processes/tasks/jobs/instructions/… that can make progress in parallel.
- Challenges
  - Work and data
    - What tasks **can** run in parallel?
  - Ordering
    - Does the **order** of tasks **matter**?
  - Synchronization
    - Do the tasks need to wait for each other?

HPC pulse

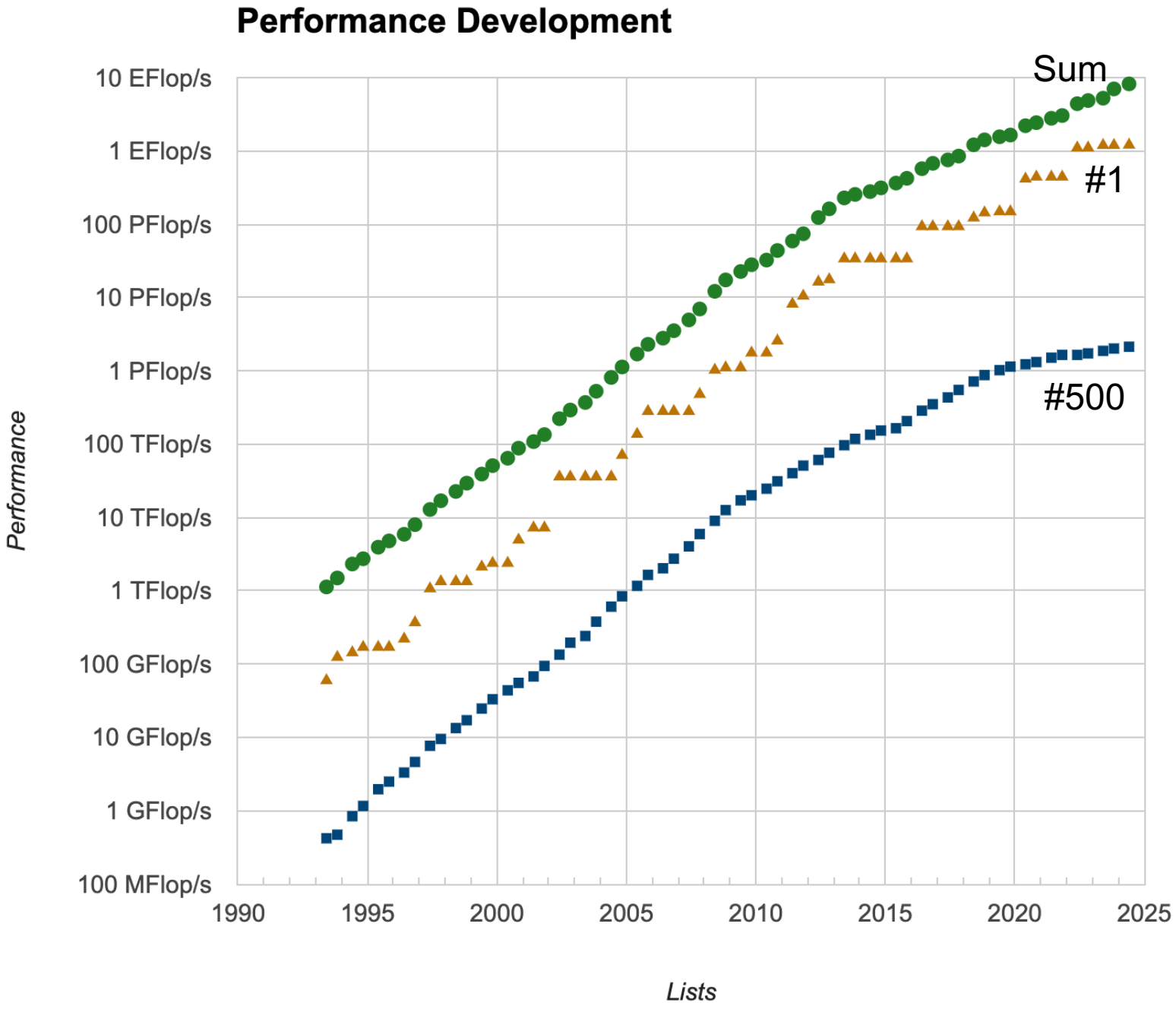
- TOP500 Project*
  - The 500 most powerful computers in the world
- Benchmark: Rmax of LINPACK
  - Solve the Ax=b linear system
    - dense problem
    - matrix A is random
  - Dominated by dense matrix-matrix multiply
- Metric: FLOPS/s
  - Computational throughput: number of floating point operations per second
- Updated twice a year: latest is Nov 2022 ?(60th edition!)

*Read more: www.top500.org

Top500 – June 2024

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,699,904	1,206.00	1,714.81	22,786
2	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	9,264,128	1,012.00	1,980.01	38,698
3	**Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft Azure Microsoft Azure United States	2,073,600	561.20	846.84	
4	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
5	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,752,704	379.70	531.51	7,107

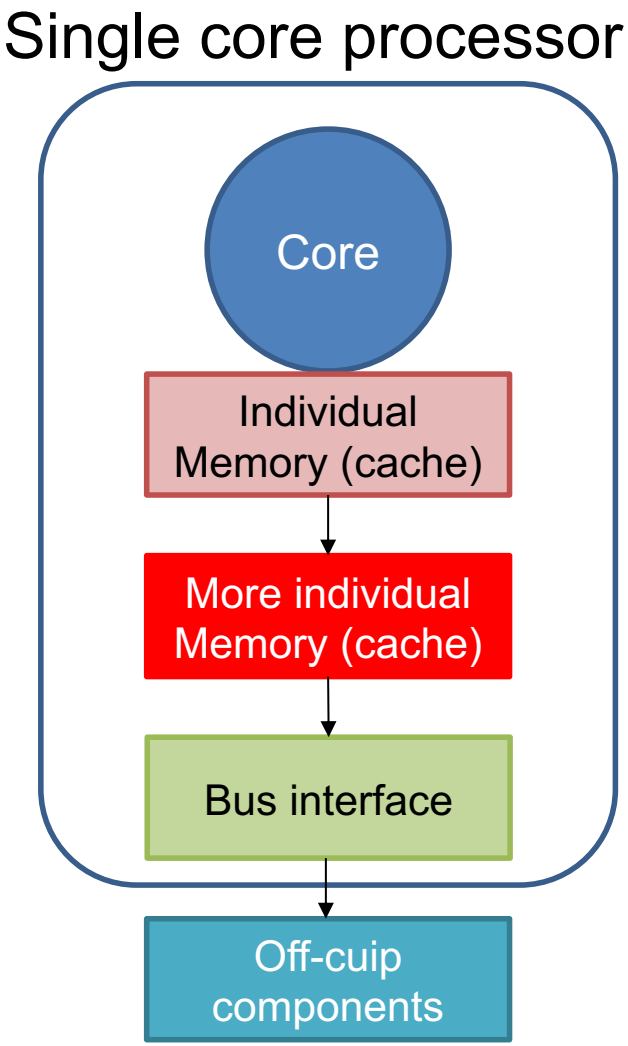
Top500 – June 2024



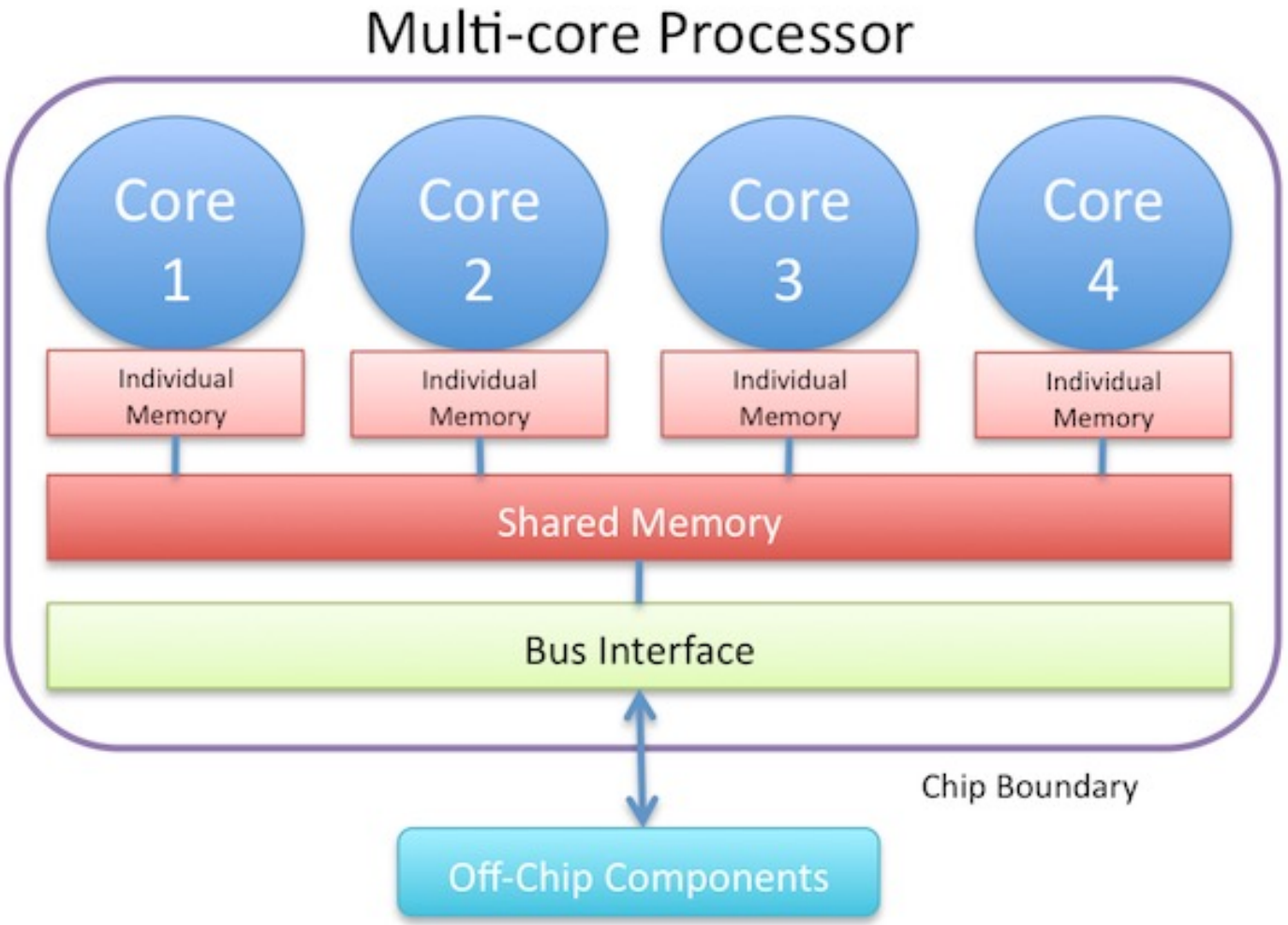
A “CO” VIEW

Shared memory machines

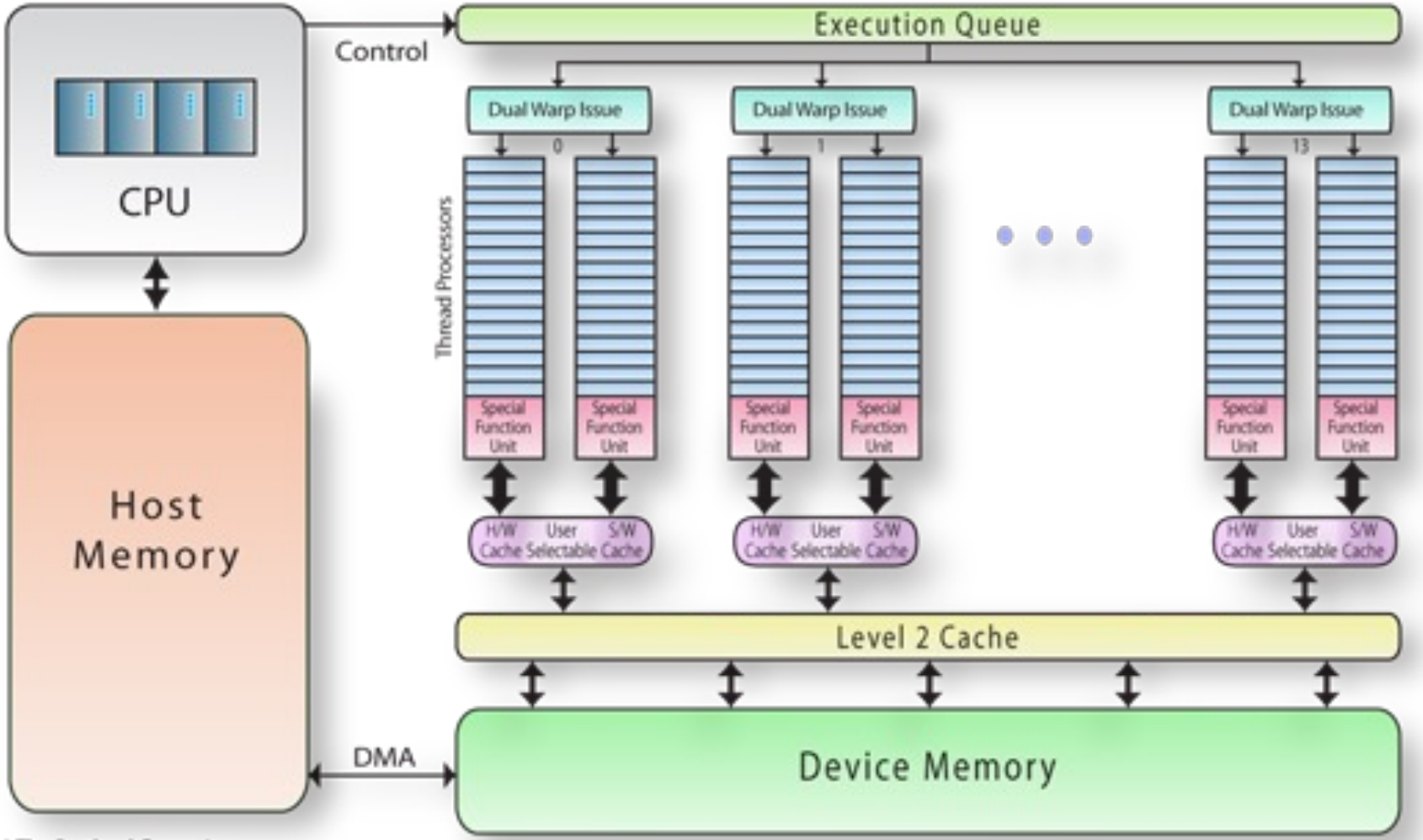
Traditional CPU



A multi-core CPU



A GPU (aka, a “many-core PU”)



BASICS OF PARALLEL PROGRAMMING

Parallel processing & programming

- Application = collection of tasks
- Parallel processing = enable (some of) the tasks to execute in parallel
  - The more the better?
  - Which tasks?
- Parallel machines
  - Enable parallel execution of tasks on hardware resources
- Parallel programming = *build* parallel programs
  - Two problems:
    - “Detect” parallelism => parallel models of computation, algorithms, …
    - Implement parallelism => programming models & tools

In practice

- Application = collection of tasks

- A task can be a process or a **thread**
- Multiple threads run in parallel (=multi-threading)
- Ideally: one thread per core

- More cores? More parallelism!

- How does this impact performance?

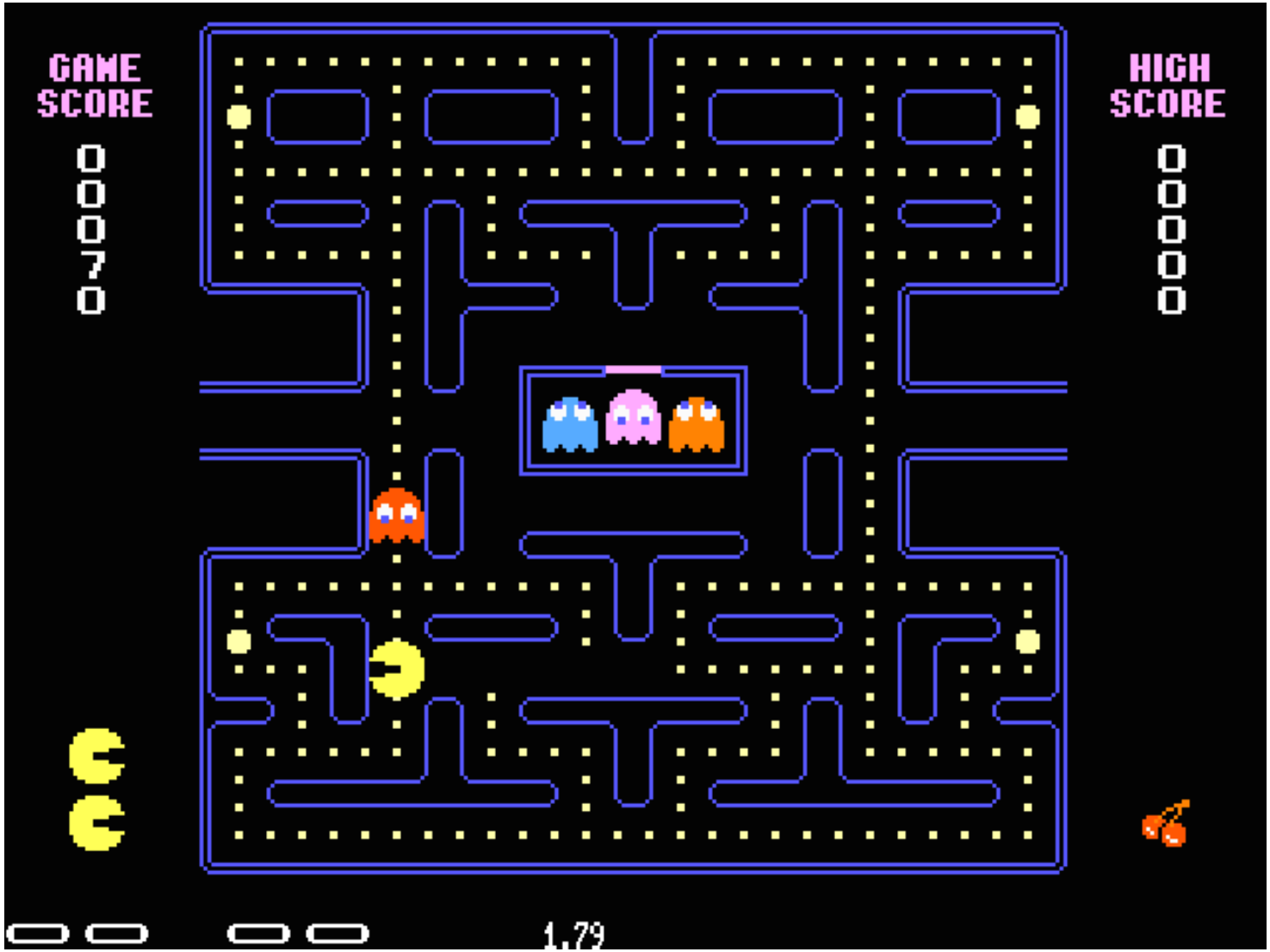
Multithreading challenges

- Parallelism (= compute things in the same time)
  - Goal: Provide tasks, which can be implemented as threads
  - Challenges: load balance, data dependencies
- Communication (= sharing data between tasks)
  - Goal: Determine shared and private variables
  - Challenges: limit communication/shared data accesses
- Synchronization (= dependency between tasks)
  - Goal: Identify task dependencies and ordering limitations
  - Challenges: use the right primitives

GPGPU ?!

Massive parallelism => massive performance

Graphics in the 80s



Graphics in 2018

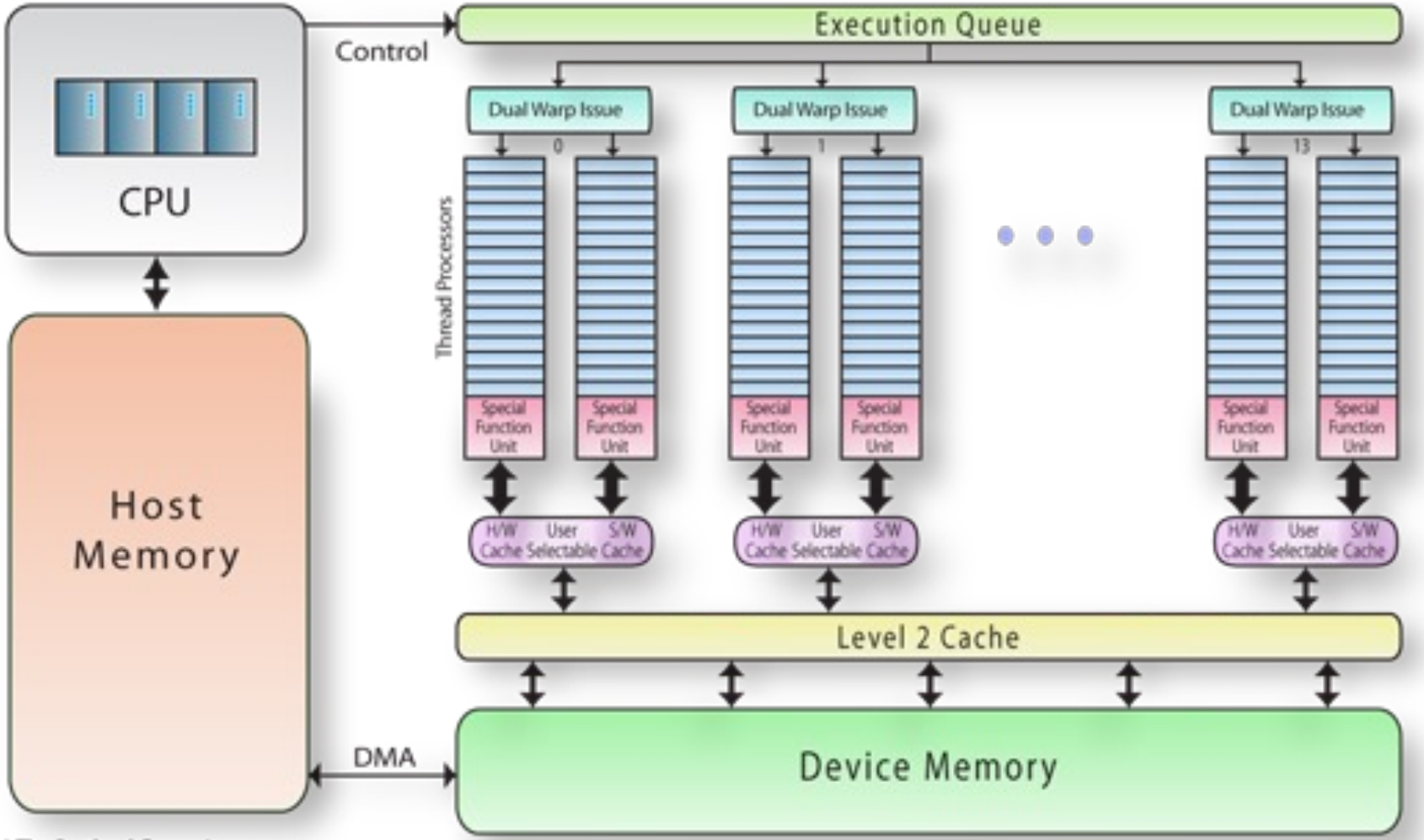
GPUs in movies

- From Ariel in Little Mermaid to Brave

So …

- GPUs are a steady market
  - Gaming
  - CAD-like activities
    - Traditional or not …
  - Visualisation
    - Scientific or not …
- GPUs are increasingly used for other types of applications
  - Number crunching in science, finance, image processing
  - (fast) Memory operations in big data processing

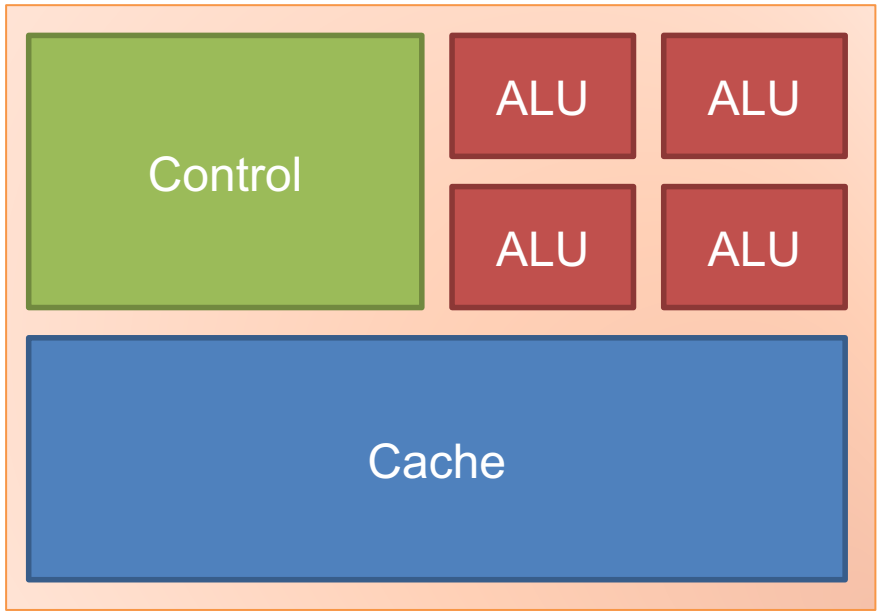
A GPU Architecture



CPU vs. GPU

- Which one is better?

CPU vs. GPU

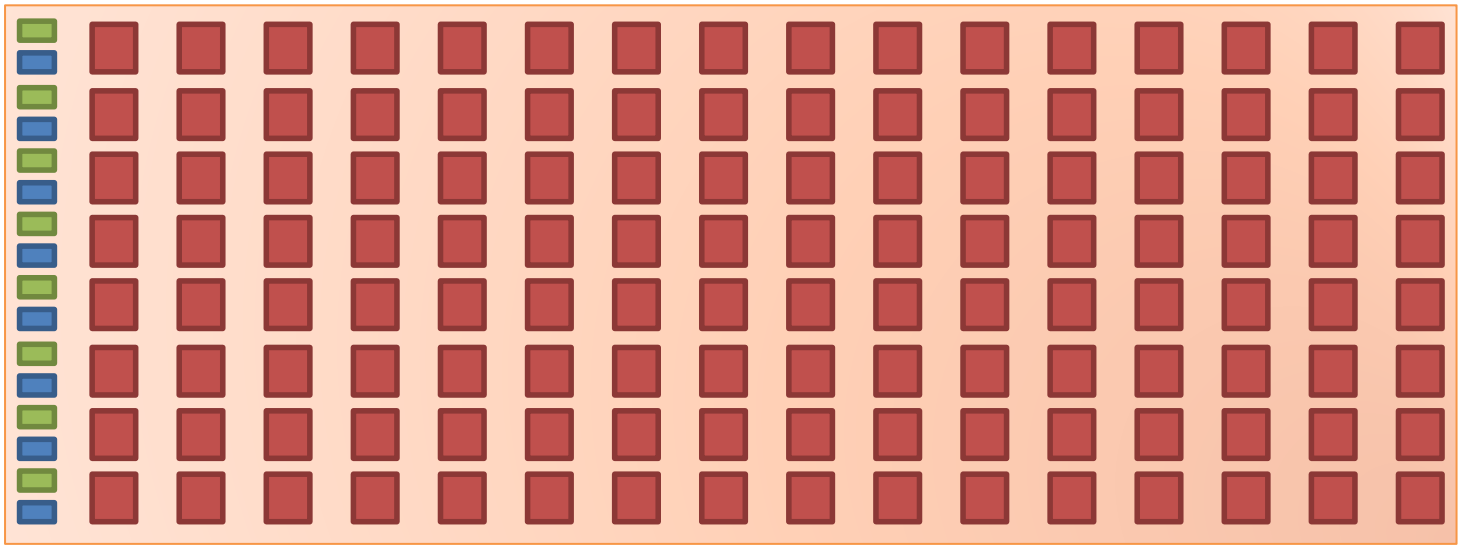


CPU

Few complex cores
Lots of on-chip memory
Lots of control logic

GPU

many
simple cores,
little memory,
little control



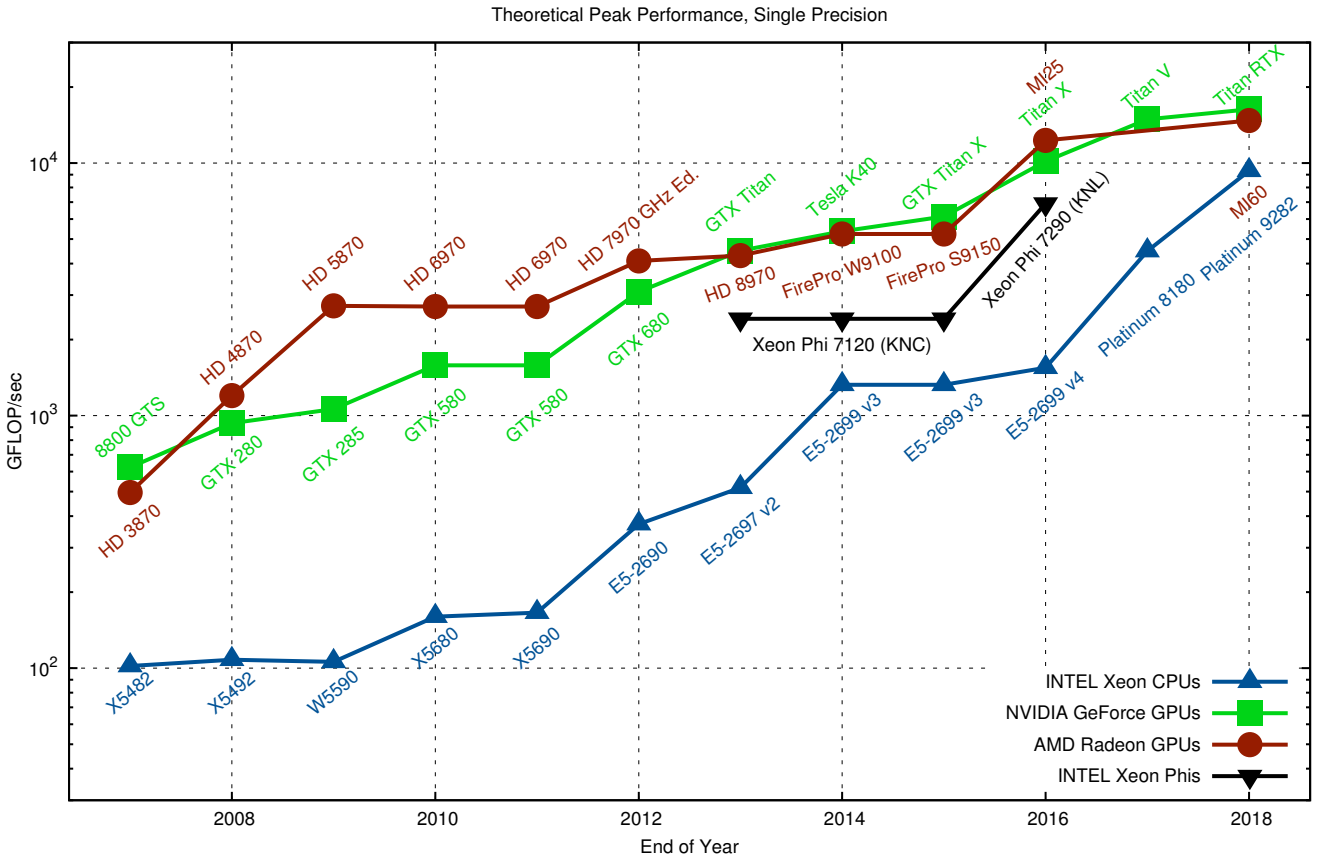
Why so different?

- Different goals produce different designs!
  - CPU must be good at everything
  - GPUs focus on massive parallelism
    - Less flexible, more specialized
- CPU: minimize latency experienced by 1 thread
  - big on-chip caches
  - sophisticated control logic
- GPU: maximize throughput of all threads
  - # threads in flight limited by resources => lots of resources (registers, etc.)
  - multithreading can hide latency => no big caches
  - share control logic across many threads

CPU vs. GPU

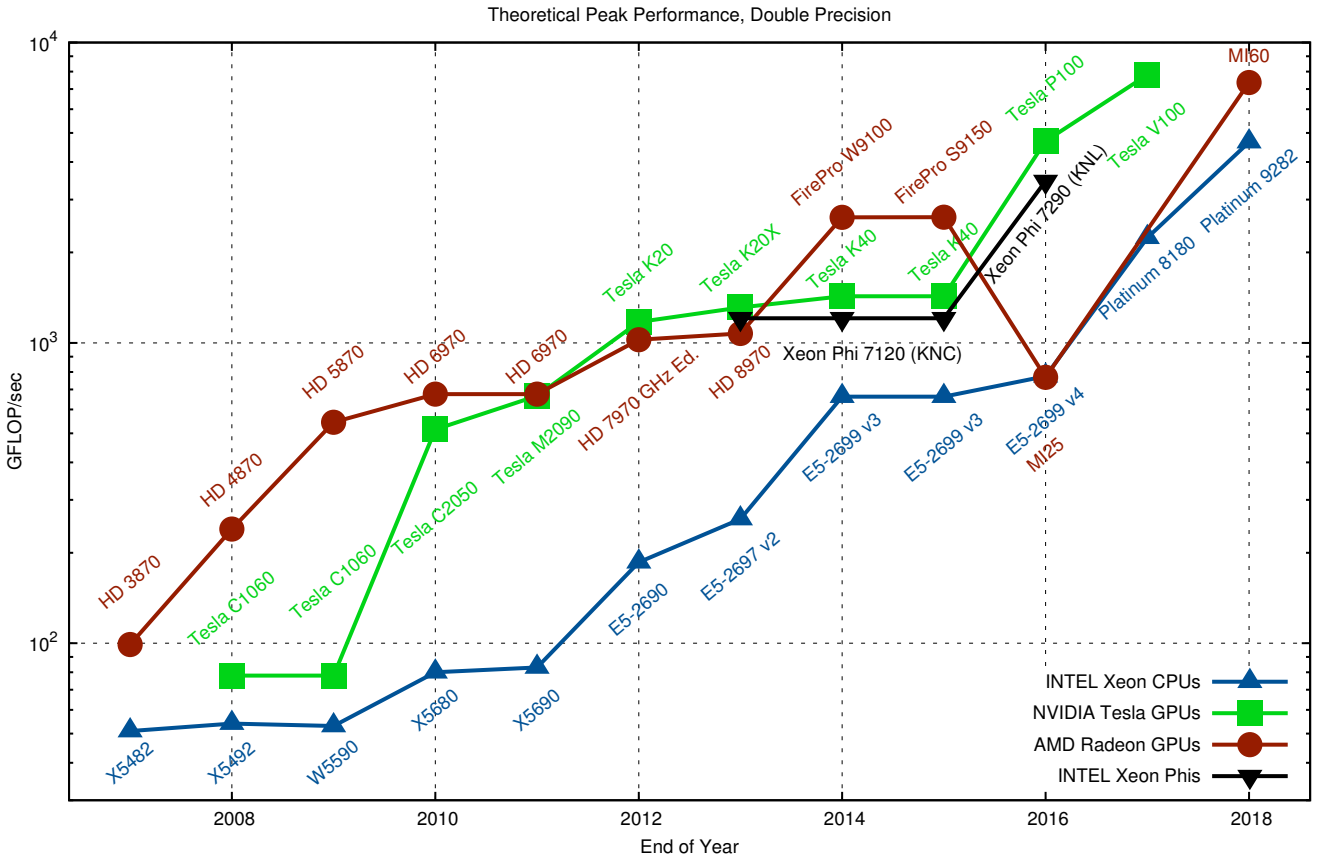
- Movie
- The Mythbusters
  - Jamie Hyneman & Adam Savage
  - Discovery Channel
- Appearance at NVIDIA's NVISION 2008

multi vs *many* cores (SP-FLOPs)



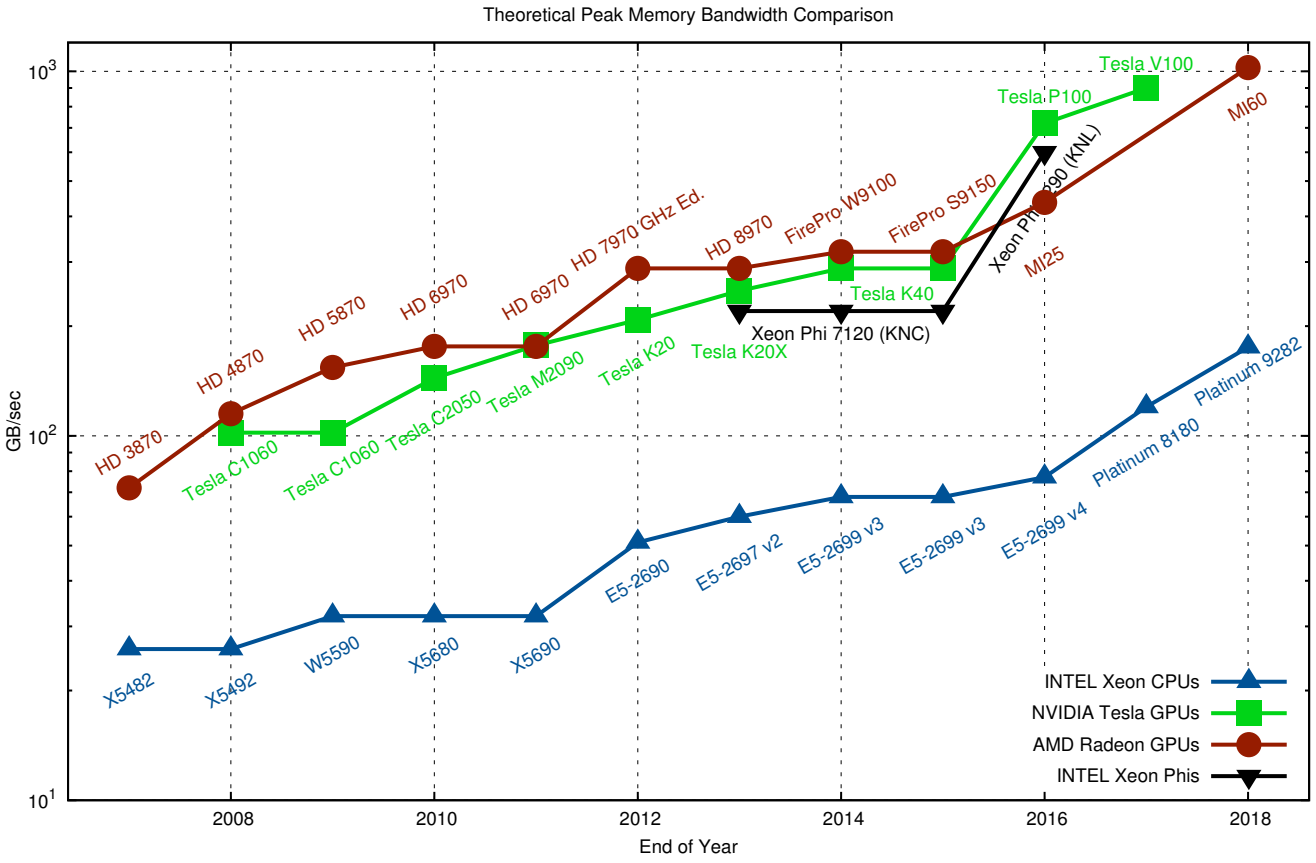
https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

multi vs *many* cores (DP-FLOPs)



https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

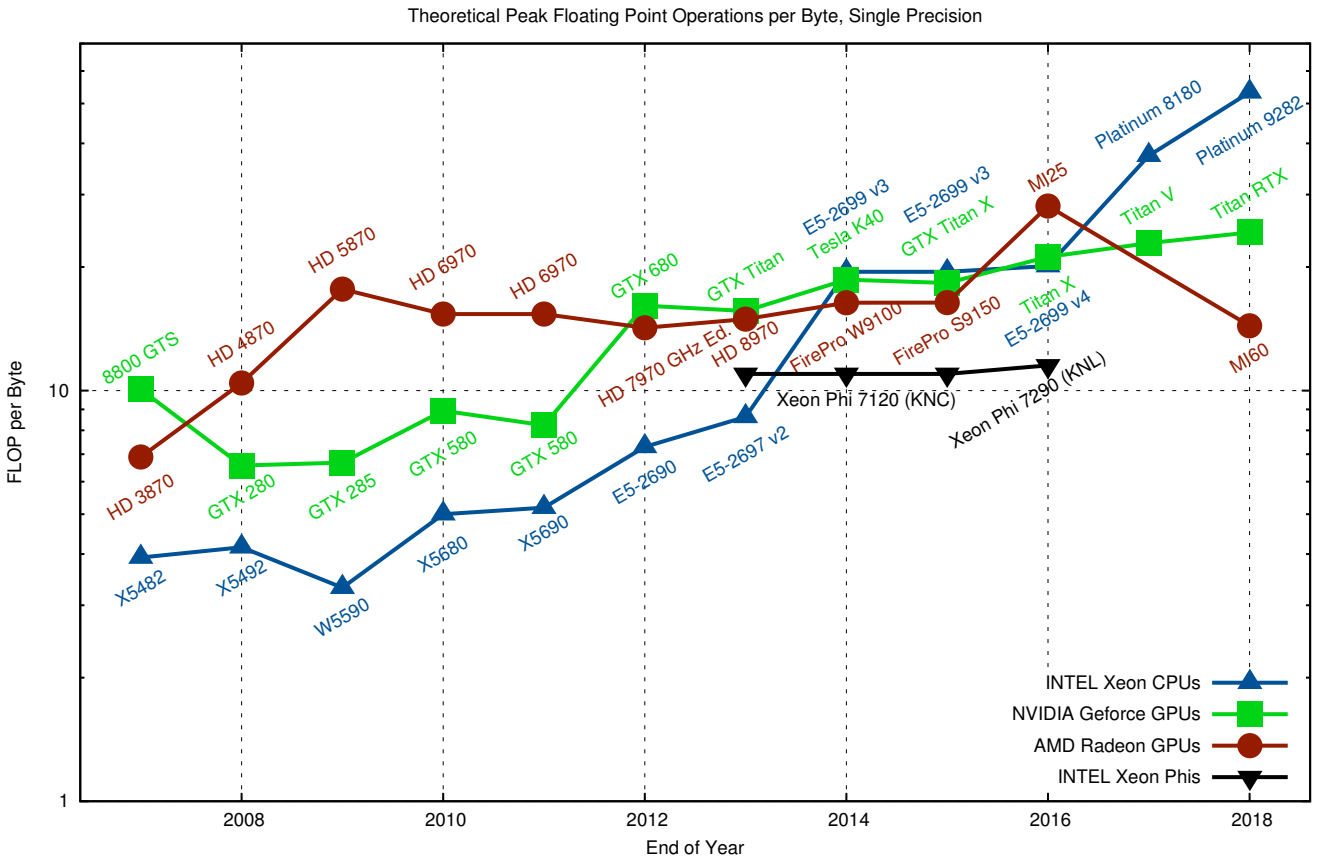
multi vs *many* cores (GB/s)



https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

Balance ?

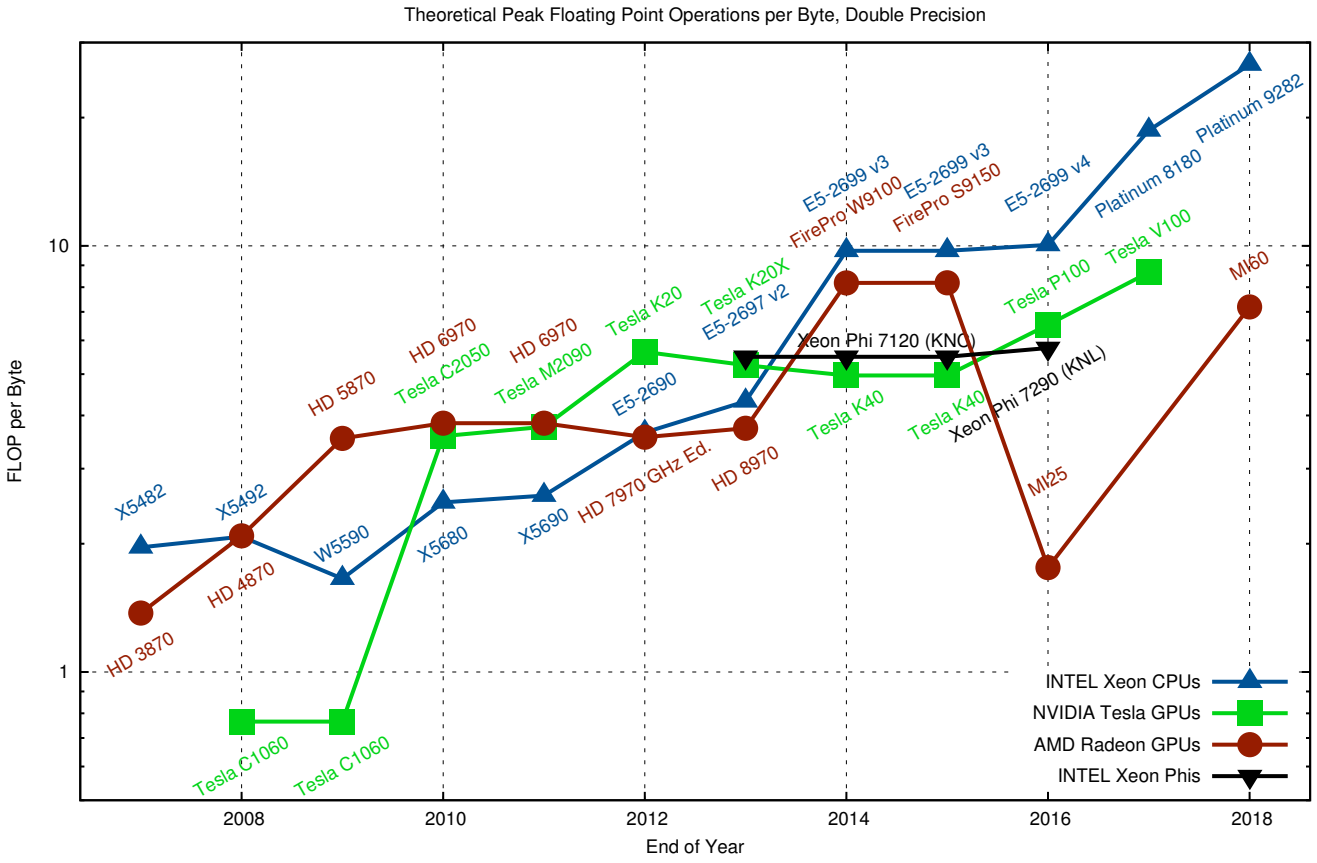
FLOPs/Byte (SP) !



https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

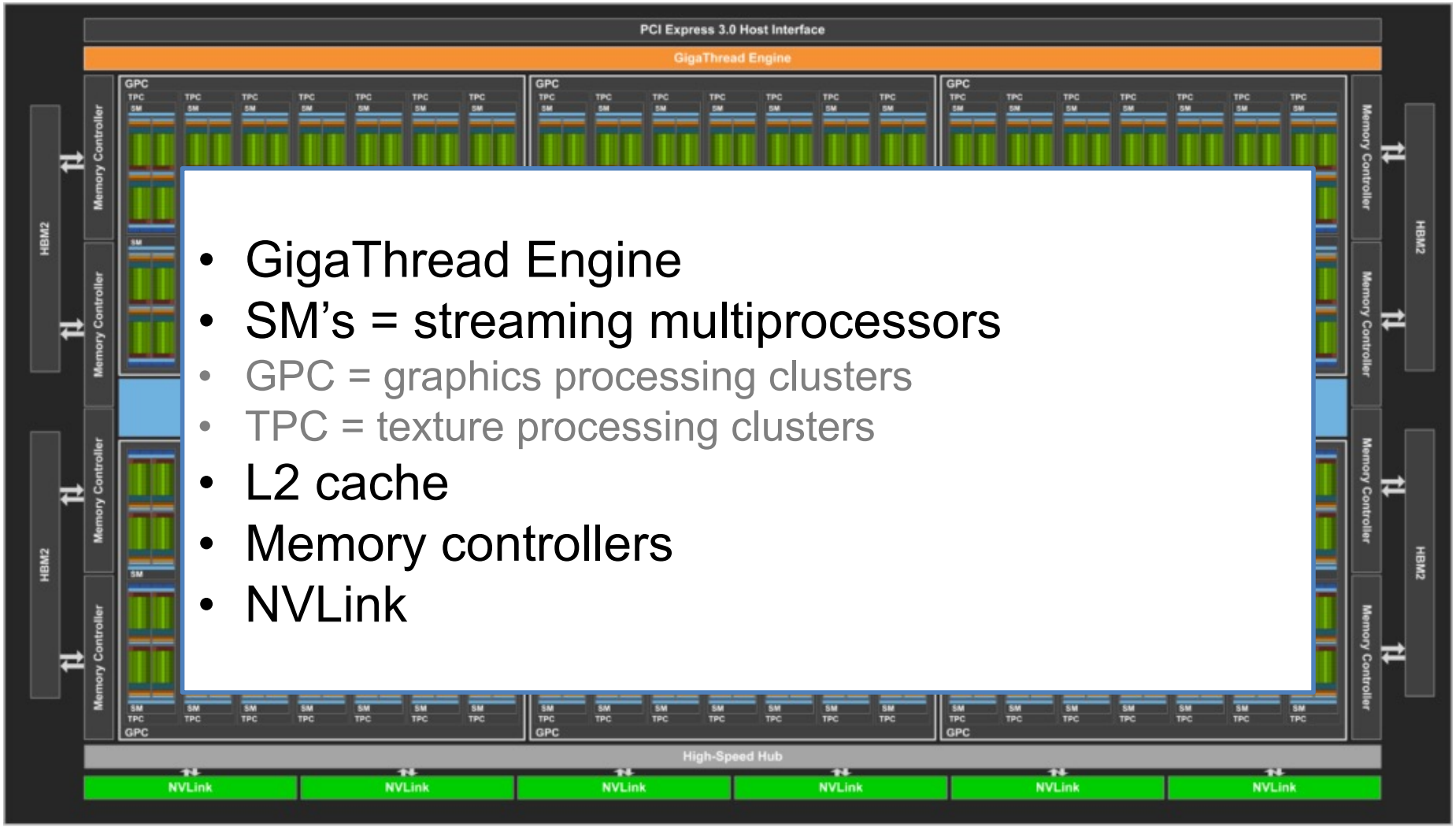
Balance ?

FLOPs/Byte (DP) !



https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

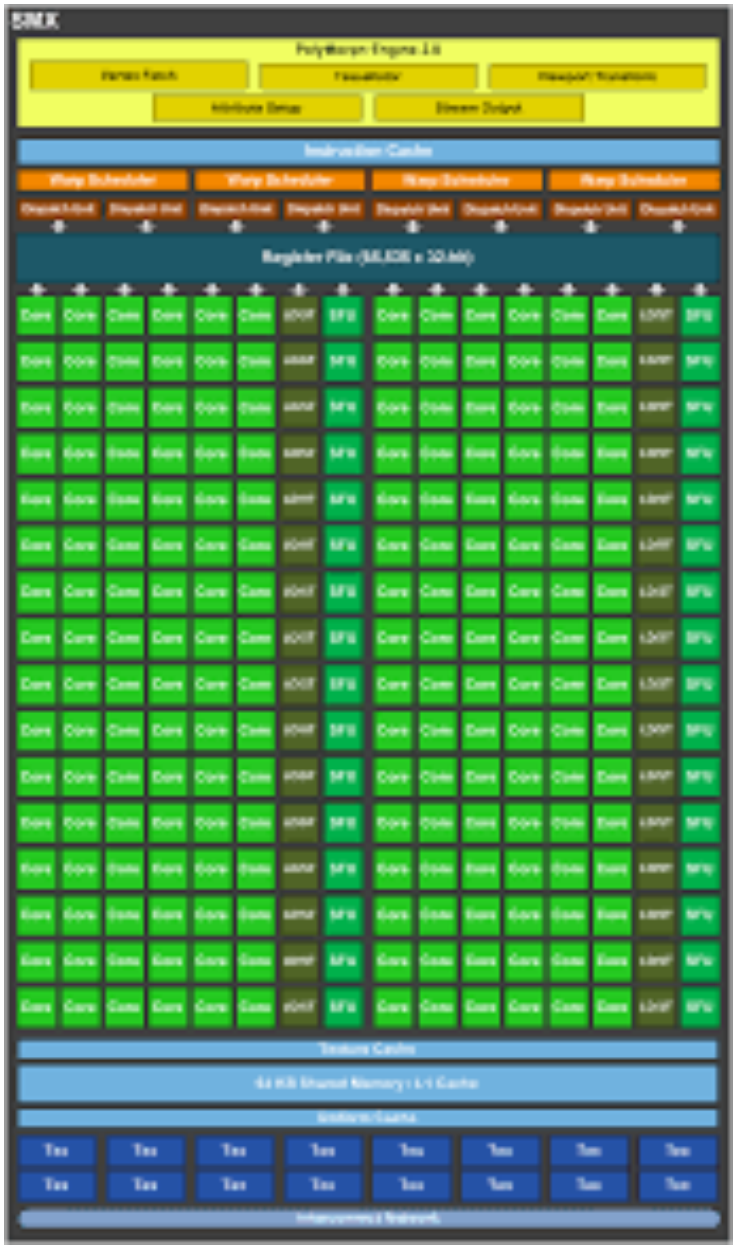
Inside an NVIDIA GPU architecture



- GigaThread Engine
- SM's = streaming multiprocessors
- GPC = graphics processing clusters
- TPC = texture processing clusters
- L2 cache
- Memory controllers
- NVLink

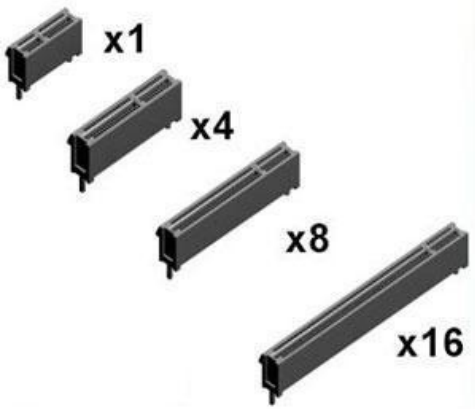
Inside an SM

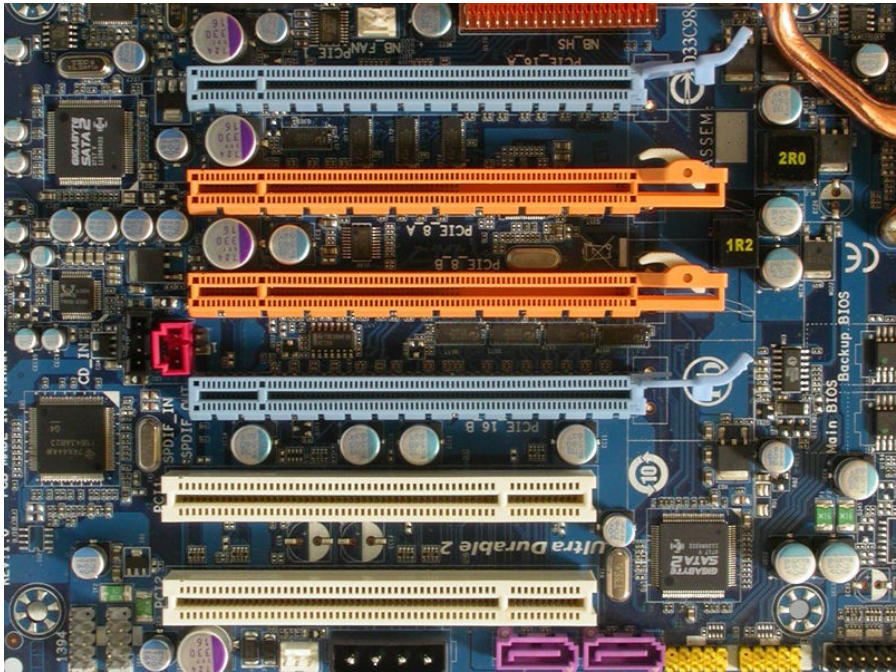
- Different types of cores
  - CUDA Cores (INT/FP32)
  - LD/ST
  - Special function units
  - DP Units (Pascal)
  - Tensor units (Volta)
- Register file
- Warp scheduler
- Data caches
- Instruction buffers/caches
- Texture units



GPU Integration into the host system

- Typically PCI Express
- Current v6.0: 64 GT/s
  - 121 GB/s in each direction is possible in 16-lane configuration.





NVIDIA GPUs (8+ years)

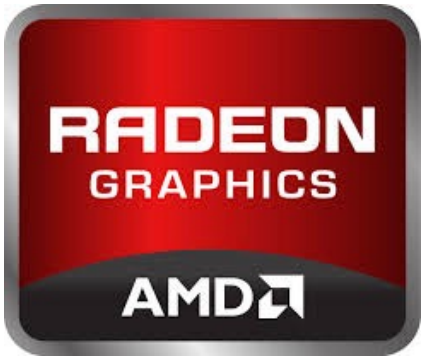
	Fermi	Kepler	Maxwell	Pascal	Volta
GPU	GTX480	GK180	GM200	GP100	GV100
Compute capability (CC)	2.x	3.5	5.2	6.0	7.0
SMs					80
TPC					40
FP3					64
FP6					32
Cloc					30
Peak [TFLOPs]					15.7
Peak FP64 [TFLOPs]	0.168	1.68	.21	5.3	7.8

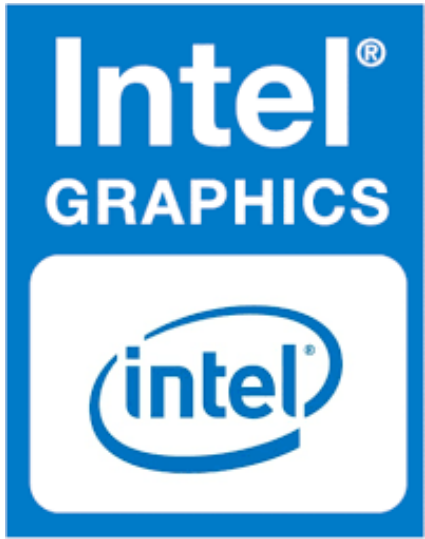
Tesla K40m uses GK110B, which has:

- 2880 cores (15 SMX x 192 cores/SMX)
- 700-900 MHz
- 12 GB RAM
- PCIe 3

Other players on the market

- **AMD** (former ATI)
  - Much better performance
  - Programmed using OpenCL (standard!)
  - Poorer software drivers and infrastructure (so far)
  - A lot less libraries and tools
  - Much smaller community effort
- **arm** (formerly ARM ☺)
  - Low-power devices (mobile platforms mostly)
  - Programmed using OpenCL
  - Lower performance than ATI and Intel, by choice
- **Intel**
  - To support own CPUs with integrated graphics
  - Programmed using OpenCL





All GPUs …

- Have a similar architecture
  - Massively parallel
  - Simple cores
  - Complex memory system
- Are programmed in a similar way
  - Fine-grain (SIMD/SIMT) parallelism
- Programming models ?
  - OpenCL is the de-facto standard for GPU programming
  - Lots of efforts for C++
  - Many other libraries and models on top of CUDA / OpenCL

PROGRAMMING GPU'S

Programming many-cores

= parallel programming:

- Choose/design algorithm
- Parallelize algorithm
  - Expose enough layers of parallelism
  - Minimize communication, synchronization, dependencies
  - Overlap computation and communication
- Implement parallel algorithm
  - Choose parallel programming model
  - (?) Choose many-core platform
- Tune/optimize application
  - Understand performance bottlenecks & expectations
  - Apply platform specific optimizations
  - (?) Apply application & data specific optimizations

Programming GPUs

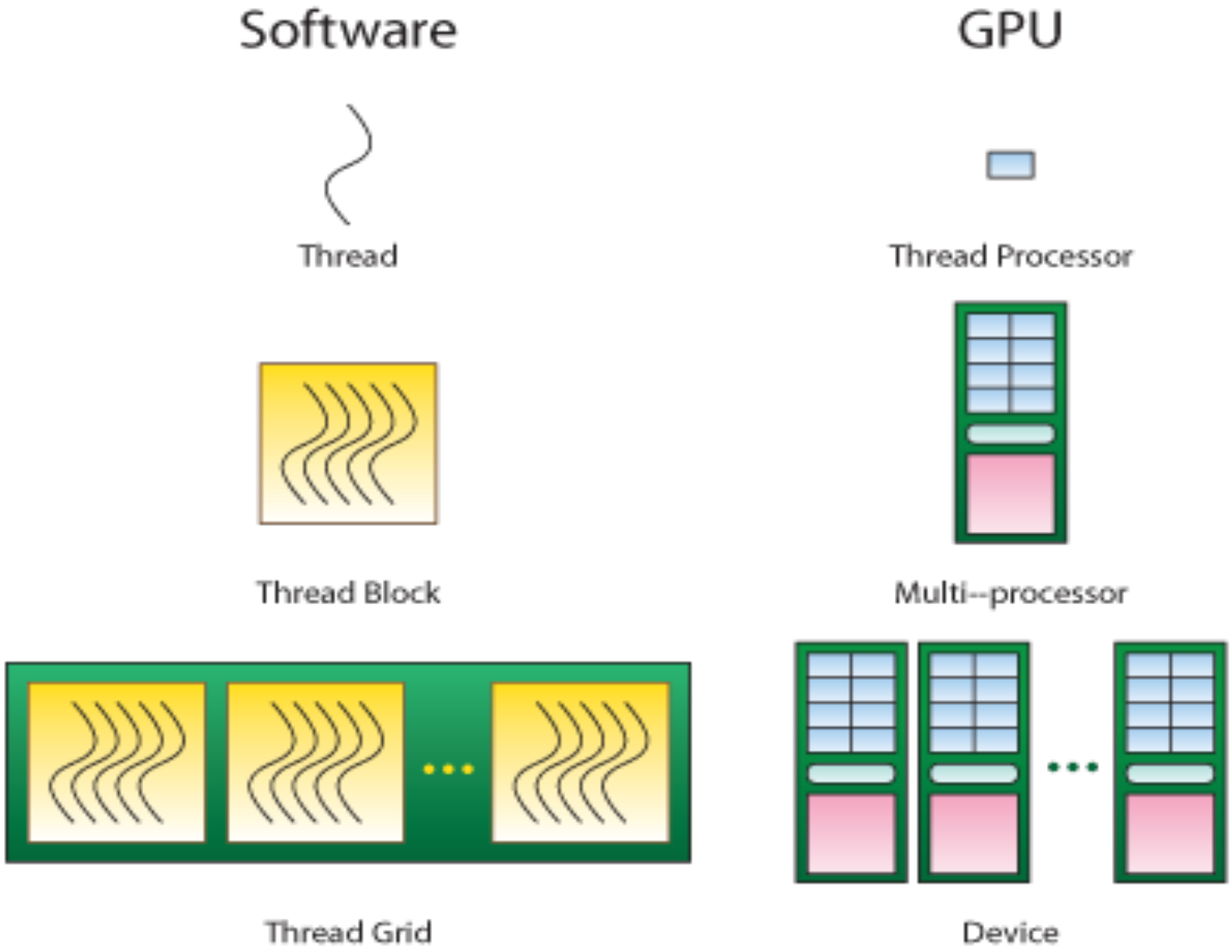
- Low-level models
  - CUDA
  - OpenCL
  - and their variations:
    - pyCL, pyCUDA, jCUDA, …
- …. and increasing level of abstraction
  - SyCL (based on C++)
  - OpenACC (pragma-based, portable)
  - OpenMP (pragma-based, portable)
- … and going domain-specific
  - TensorFlow – machine learning/deep learning
  - Forma – DSL for image processing and stencils
  - … many more …

CUDA: The principles

CUDA

- **CUDA**: Compute Unified Device Architecture
  - C/C++ extensions
    - Other wrappers exist
- Straightforward mapping of PM to hardware
  - Hierarchy of threads (map to cores)
    - Configurable at logical level
  - Various memory spaces (map to physical mem. spaces)
    - Usable via variable scopes
- **SIMT**: single instruction multiple threads
  - Have 1000s threads running concurrently
  - Hardware multi-threading
    - GPU threads are lightweight

CUDA Model of Parallelism



CUDA: Hierarchy of threads

- Each thread executes the same (*kernel*) code
  - **One thread runs on one core**
- Threads are logically grouped into thread blocks
  - Threads in the same block can cooperate
  - Threads in different blocks cannot cooperate
  - **One block runs on one SM**
- All thread blocks are logically organized in a Grid
  - 1D or 2D or 3D
  - Threads and blocks have unique IDs

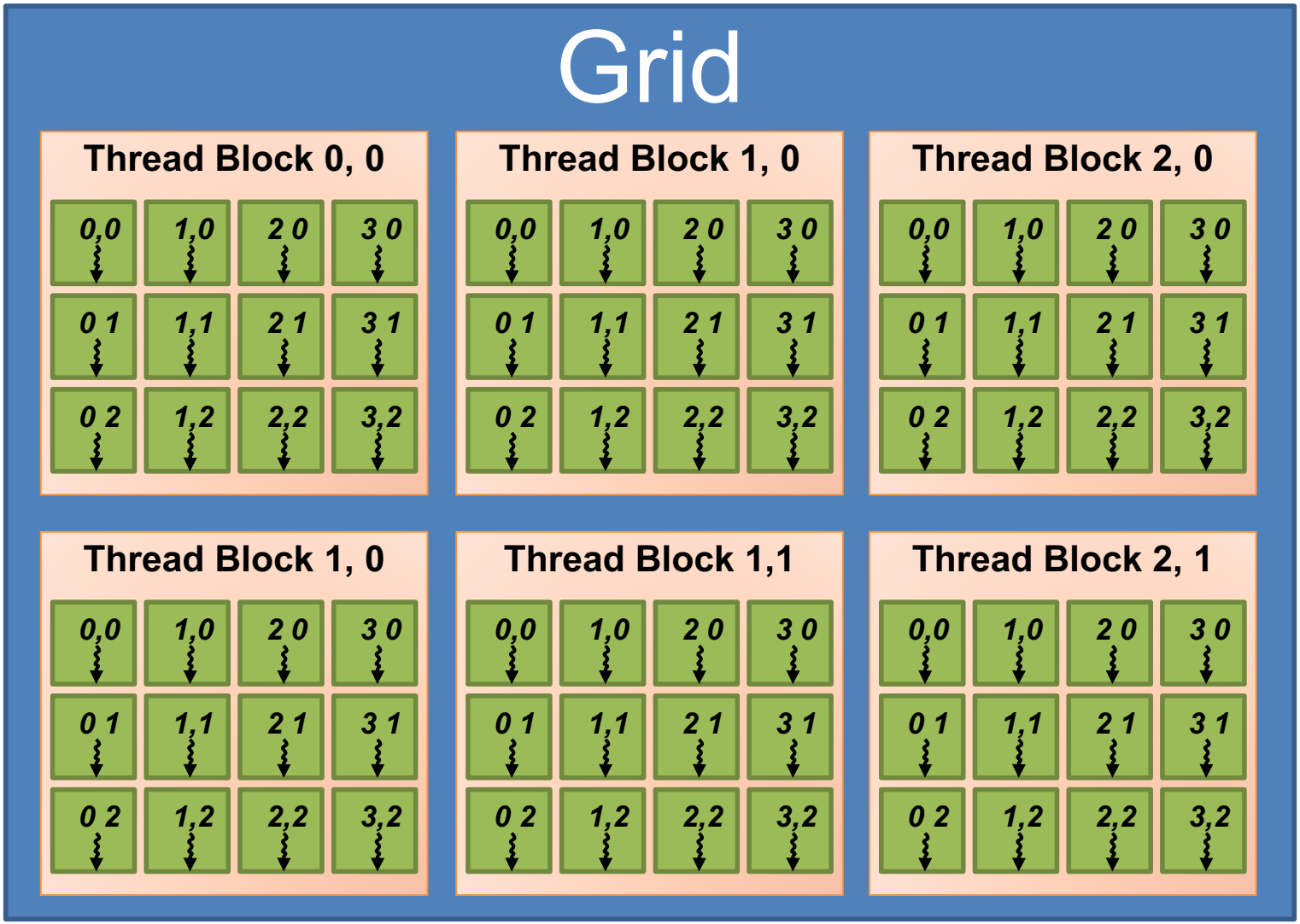
A grid specifies in how many instances the *kernel* is being run

Parallelization for GPUs

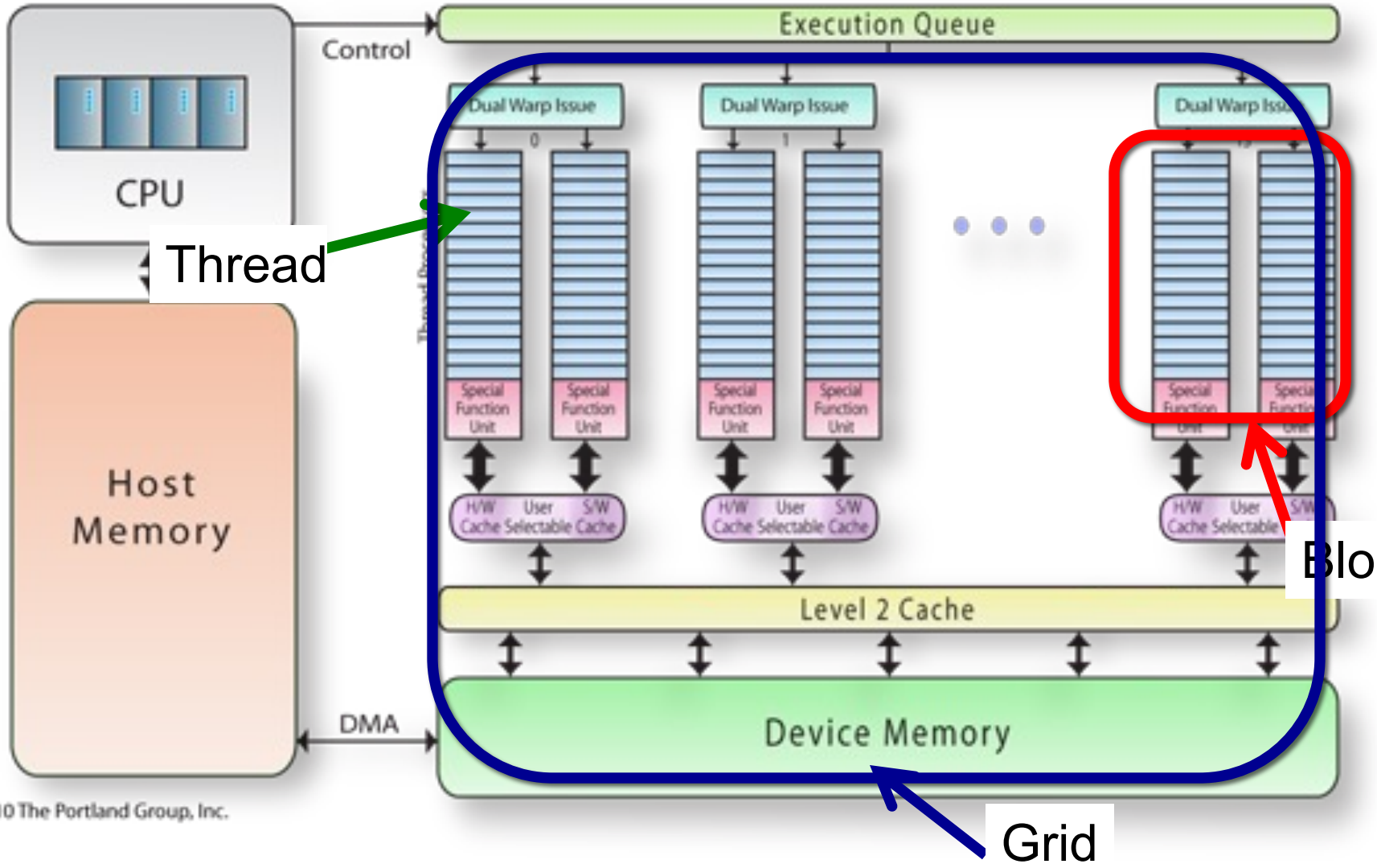
- Parallelization = find a mapping of the problem on the machine (model) such that you maximize concurrent execution.

- For GPUs (fine-grain data parallelism)
  - Map your data/work to threads
    - Write the computation for 1 thread!
  - Organize threads in blocks and blocks in grids
  - Let the hardware scheduler do the rest

This is your “application” space!



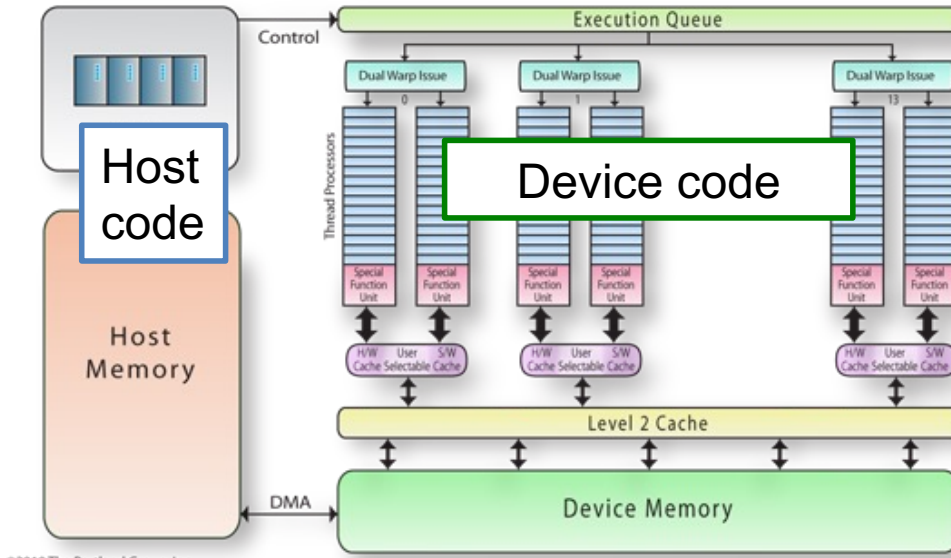
This is your hardware space



CUDA: The practicalities

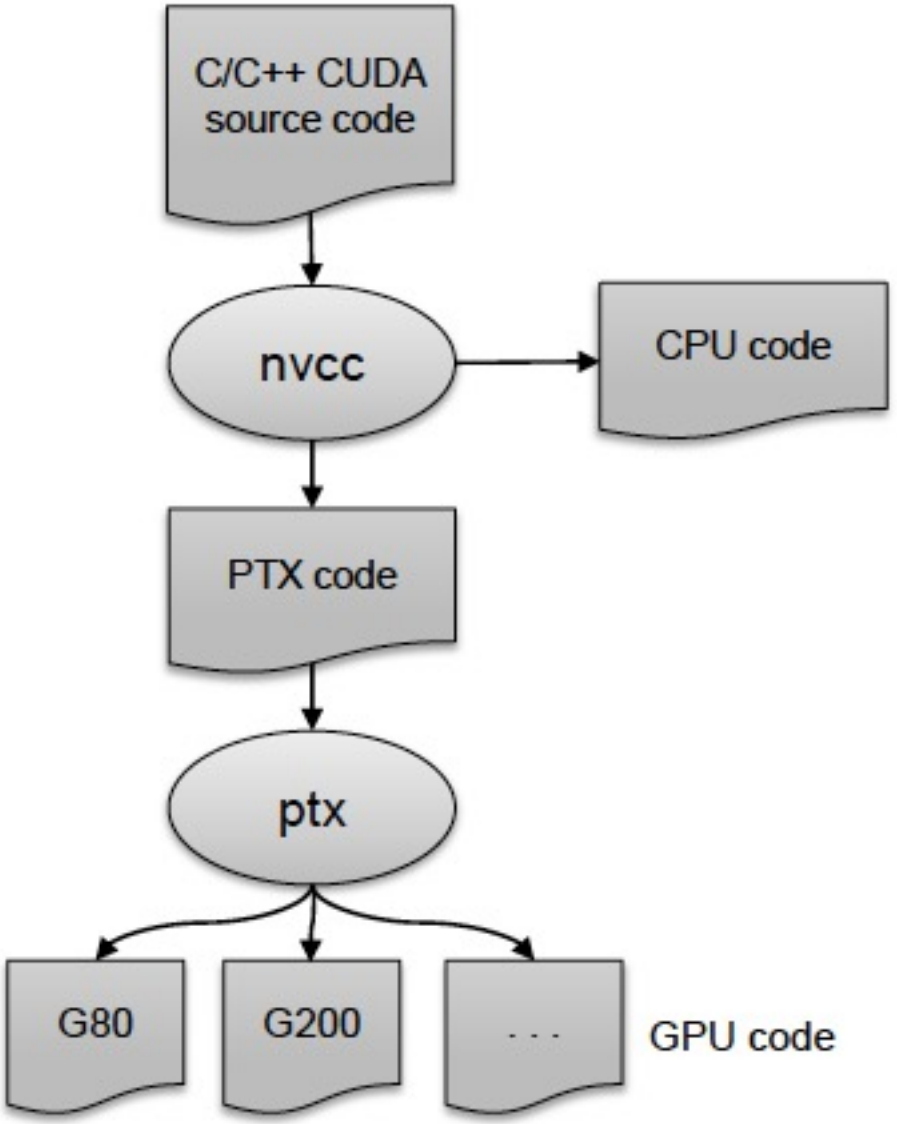
CUDA program organization

- Two types of code:
  - **Device code** = GPU code = kernel(s)
    - Sequential program
    - Write for 1 thread, execute for all
  - **Host code** = CPU code
    - Instantiate grid + run the kernel
    - Memory allocation, management, deallocation
    - C/C++/Java/Python/…
- Host-device communication
  - Explicit / implicit
    - Via PCI/e
    - Via NVLink (where available)



Compiling CUDA

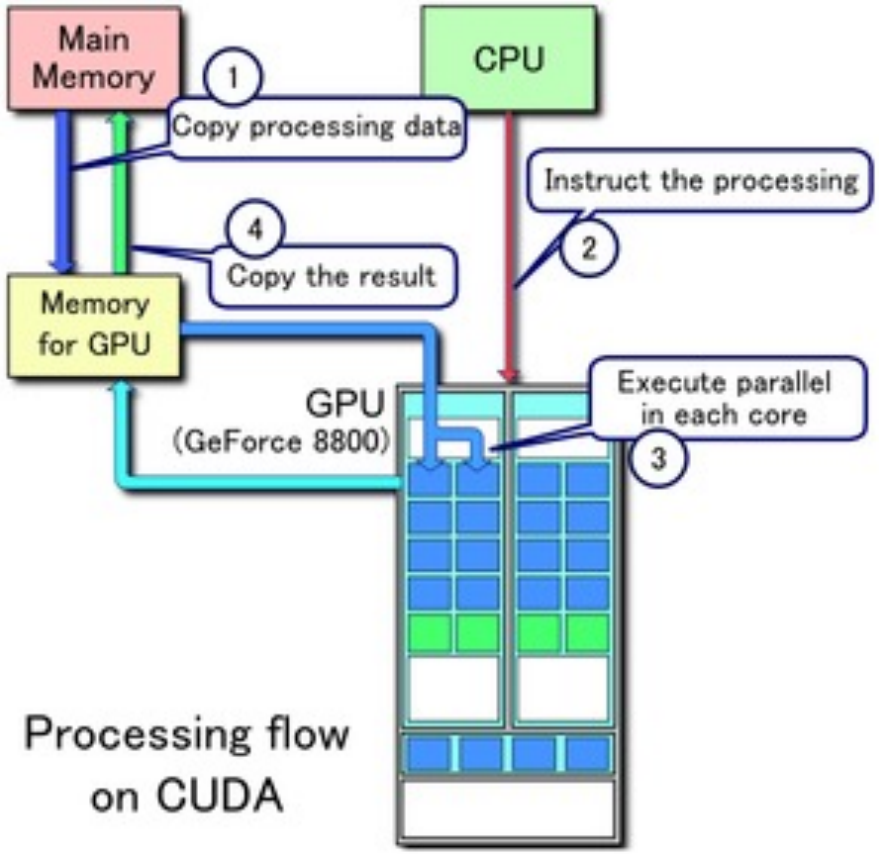
- Use nvcc
- nvcc separates source code:
  - **device code** (runs on GPU)
    - further processed by NVIDIA compiler
  - **host code** (runs on CPU)
    - further processed by host compile (g++, cl.exe)



Execution flow

- Device code executes on CPU
- Kernel code executes on GPU

- GPU memory allocation
- Transfer data CPU→GPU
- CPU calls GPU kernel
- GPU kernel executes
- Transfer data GPU→CPU
- GPU memory release
- [Repeat]



CUDA: kernel code dummy example

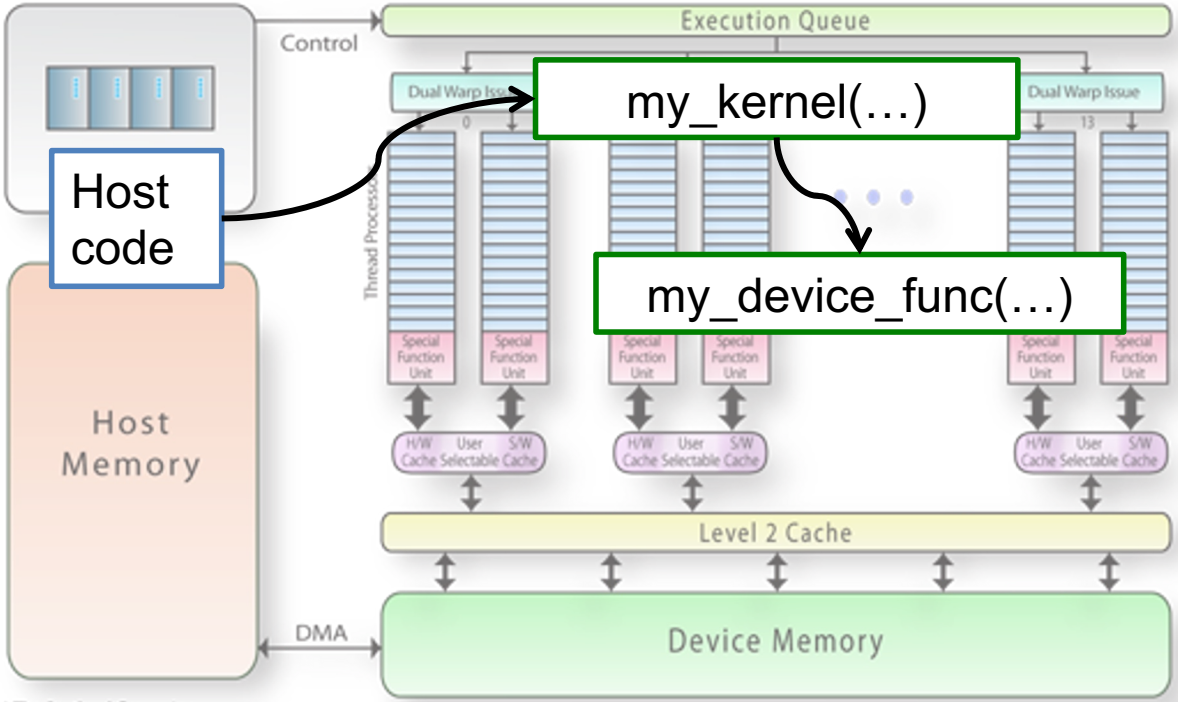
```
__global__ myKernel (int n, int *dataGPU) {
   size_t id = blockIdx.x * blockDim.x +
                    threadIdx.x;
   if (id < n) dataGPU[id]=id;
   process(dataGPU);
}

__device__ process(int *dataGPU) {
  size_t id = blockIdx.x * blockDim.x +
                    threadIdx.x;
  if (whoAmI % 4 == 0)
     dataGPU[whoAmI] = -1*dataGPU[whoAmI];
}
```

CUDA: kernel and device functions

- Device function qualifiers:

```
__global__ void my_kernel(<args>) { }
__device__ float my_device_func() { }
```

- Host functions: no qualifier



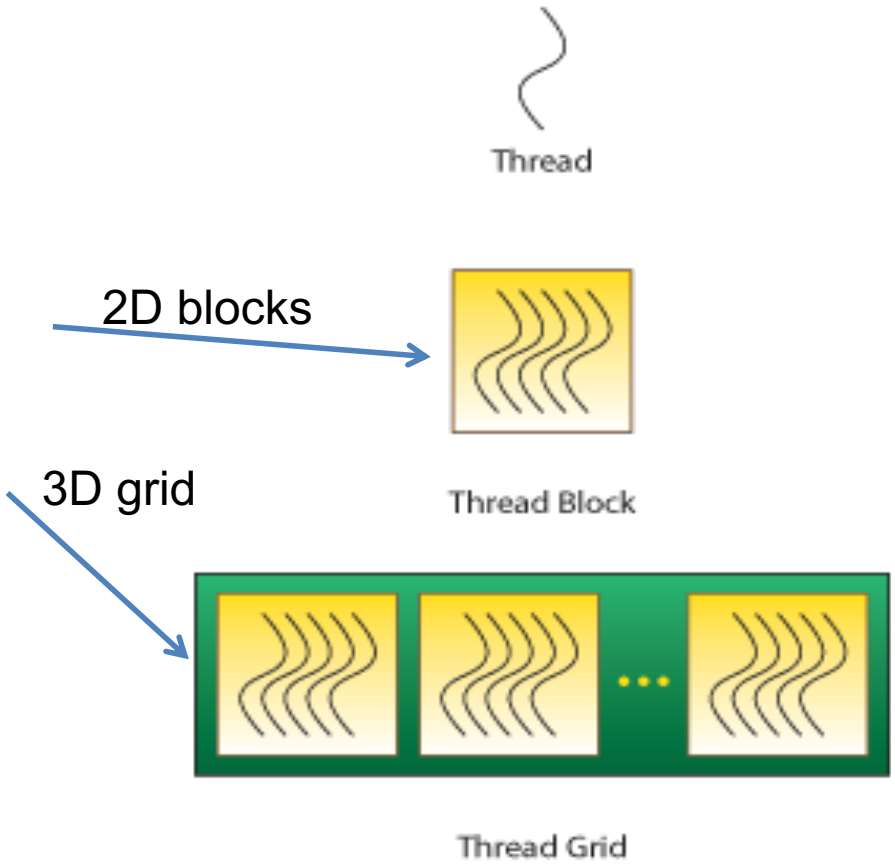
CUDA: host code dummy example

```
int n = 1024;
int nbytes = n * sizeof(int);
int* dataCPU = (int *)malloc(nbytes);
int* dataGPU;

cudaMalloc(&dataGPU, nbytes);

cudaMemcpy(dataGPU, dataCPU, nbytes,
                          cudaMemcpyHostToDevice);
myKernel<<<n/128,128>>>(n, dataGPU);
cudaMemcpy(dataCPU, dataGPU, nbytes,
                          cudaMemcpyDeviceToHost);
cudaFree(dataGPU);
free(dataCPU);
```

CUDA: kernel launch (from host)

- Execution configuration:

```
// 5000 thread blocks
dim3 myBlockSize(100,50);
// 256 threads per block
dim3 myGridSize(4,8,8);
```




```
//Launch kernel
my_kernel <<<myGridSize, myBlockSize>>> (<args>);
```

CUDA: geometry & IDs

- Built-in variables and functions **valid in device code**, used to retrieve the "geometry" of the grid.

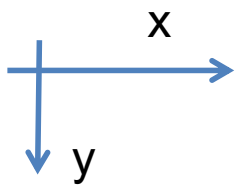
```
dim3 gridDim;    // Grid   dimension
dim3 blockDim;   // Block  dimension
dim3 blockIdx;   // Block  index
dim3 threadIdx; // Thread index
```

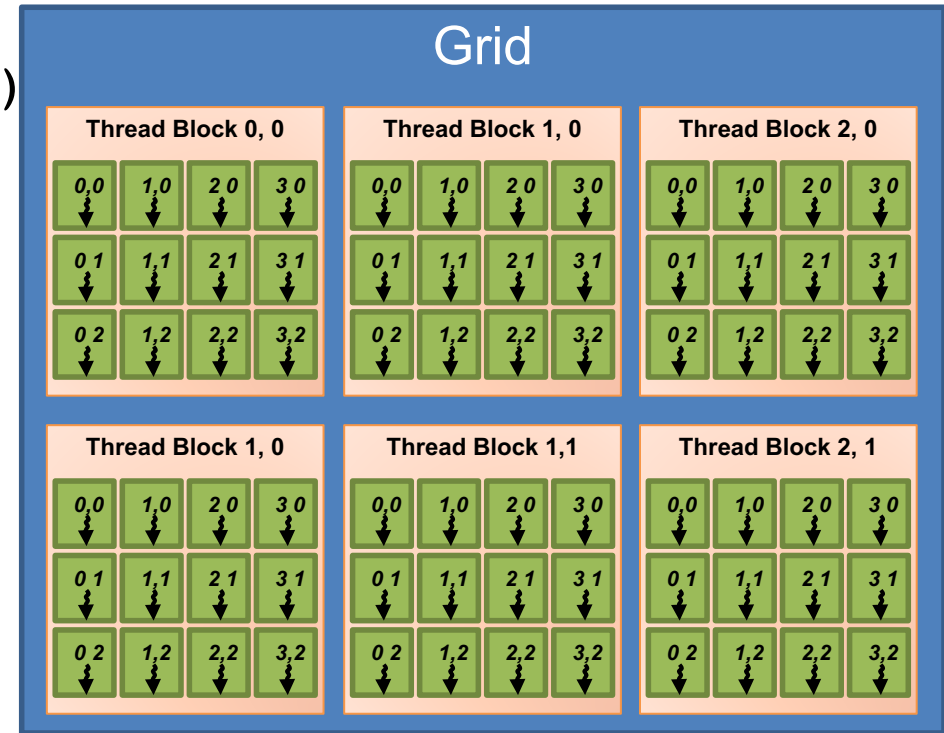
- What is the threadID?
  - At block-level: **`threadIdx`**
  - At grid-level: **`f(threadIdx, blockIdx, blockDim)`**

Kernels and grids

`myKernel<<<numBlocks,threadsPerBlock>>>(…);`

- `dim3 threadsPerBlock(4,3);`
  - `threadsPerBlock.x = 4`
  - `threadsPerBlock.y = 3`
  - `Each thread: (threadIdx.x,threadIdx.y)`
- `dim3 numBlocks(3,2);`
  - `blockDim.x = 3`
  - `blockDim.y = 2`
  - `Each block: (blockIdx.x,blockIdx.y)`
- `Total threads: 72`

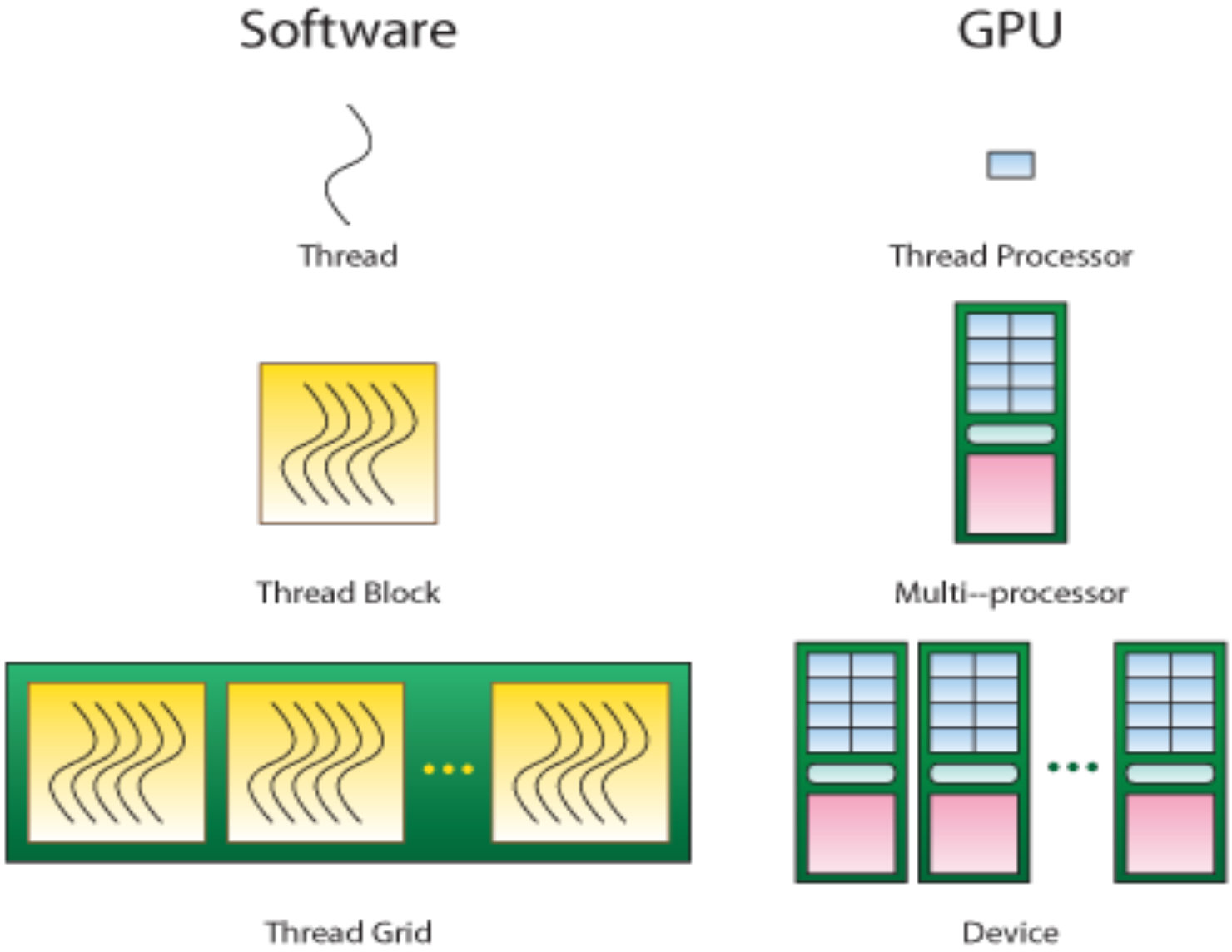




CUDA

- **CUDA**: Compute Unified Device Architecture
  - C/C++ extensions
    - Other wrappers exist
- Straightforward mapping of PM to hardware
  - Hierarchy of threads (map to cores)
    - Configurable at logical level
  - Various memory spaces (map to physical mem. spaces)
    - Usable via variable scopes
- **SIMT**: single instruction multiple threads
  - Have 1000s threads running concurrently
  - Hardware multi-threading
    - GPU threads are lightweight

CUDA Model of Parallelism



CUDA: Hierarchy of threads

- Each thread executes the same (*kernel*) code
  - **One thread runs on one core**
- Threads are logically grouped into thread blocks
  - Threads in the same block can cooperate
  - Threads in different blocks cannot cooperate
  - **One block runs on one SM**
- All thread blocks are logically organized in a Grid
  - 1D or 2D or 3D
  - Threads and blocks have unique IDs

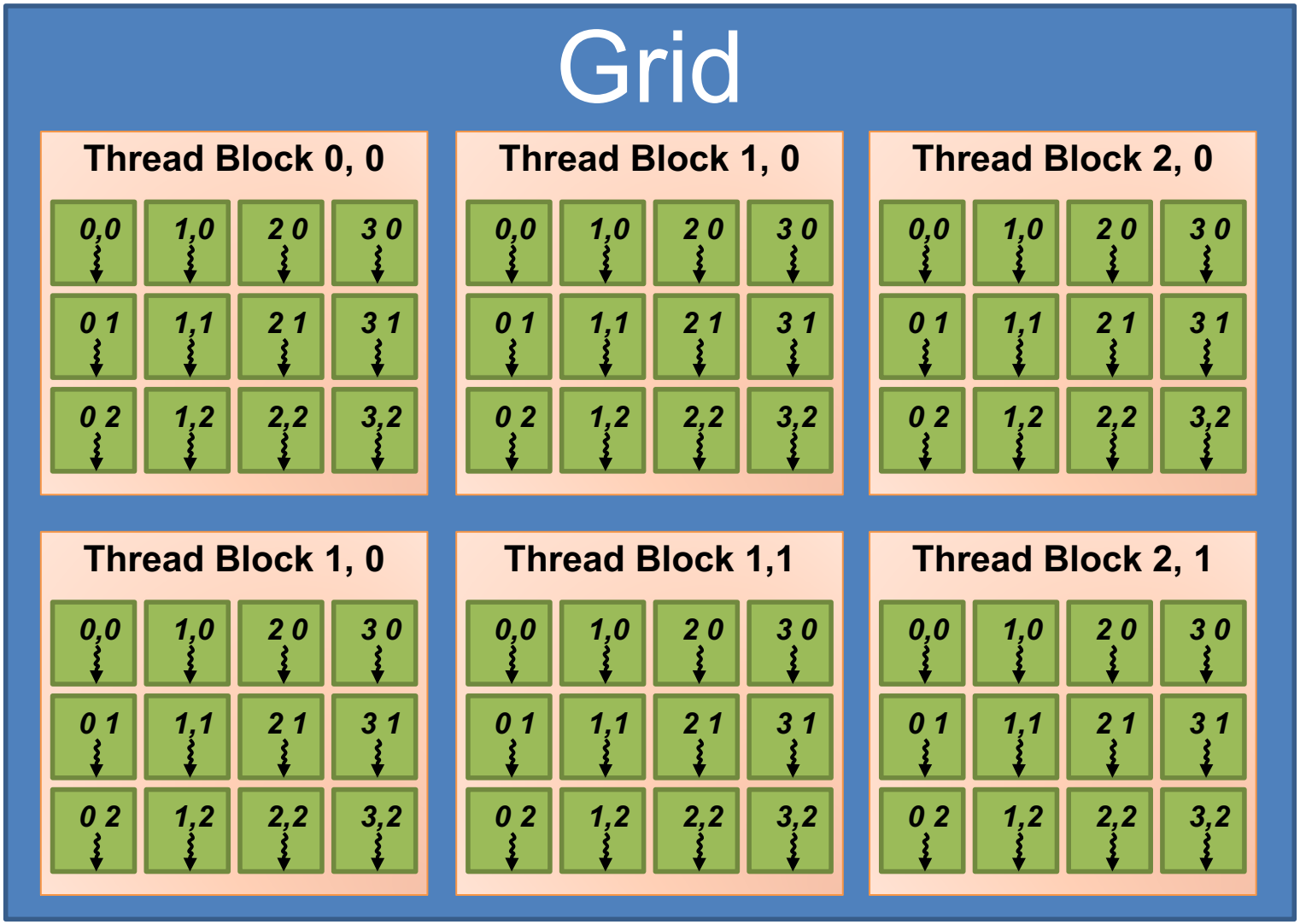
A grid specifies in how many instances the *kernel* is being run

Parallelization for GPUs

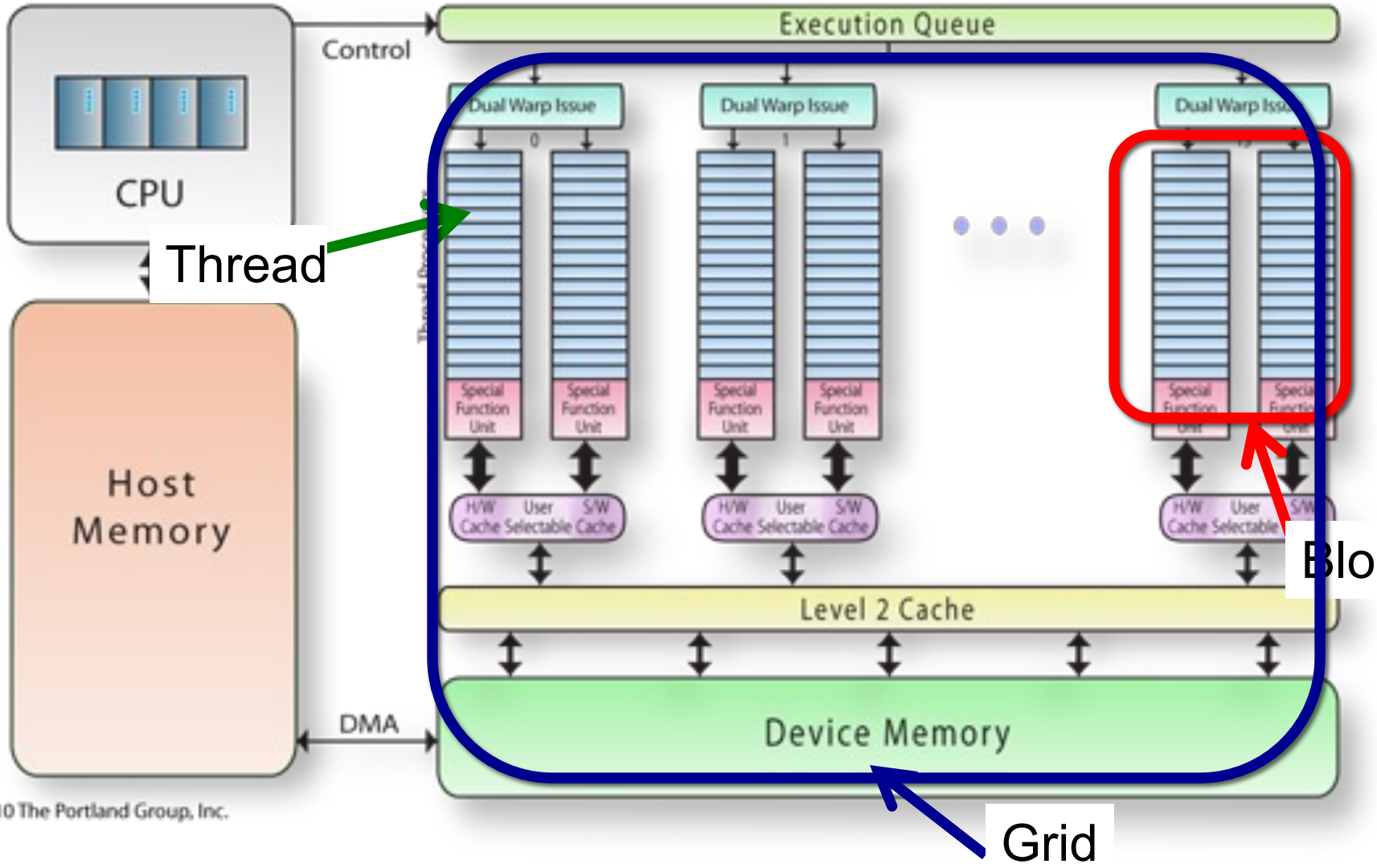
- Parallelization = find a mapping of the problem on the machine (model) such that you maximize concurrent execution.

- For GPUs (fine-grain data parallelism)
  - Map your data/work to threads
    - Write the computation for 1 thread!
  - Organize threads in blocks and blocks in grids
  - Let the hardware scheduler do the rest

This is your “application” space!



This is your hardware space



EXAMPLE: VECTOR-ADD

Design the CUDA algorithm

- Identify kernel(s)
- Determine mapping of operations and data to threads
- Write kernel(s)
  - Sequential code
  - Written per-thread
- Determine block geometry
  - Threads per block, blocks per grid
  - Number of grids (>= number of kernels)
- Write host code
  - Memory initialization and copying to device
  - Kernel(s) launch(es)
  - Results copying to host
- Optimize the kernels

Vector add: sequential

```
void vector_add(int size, float* a, float* b, float* c) {
    for(int i=0; i<size; i++) {
        c[i] = a[i] + b[i];
    }
}
```

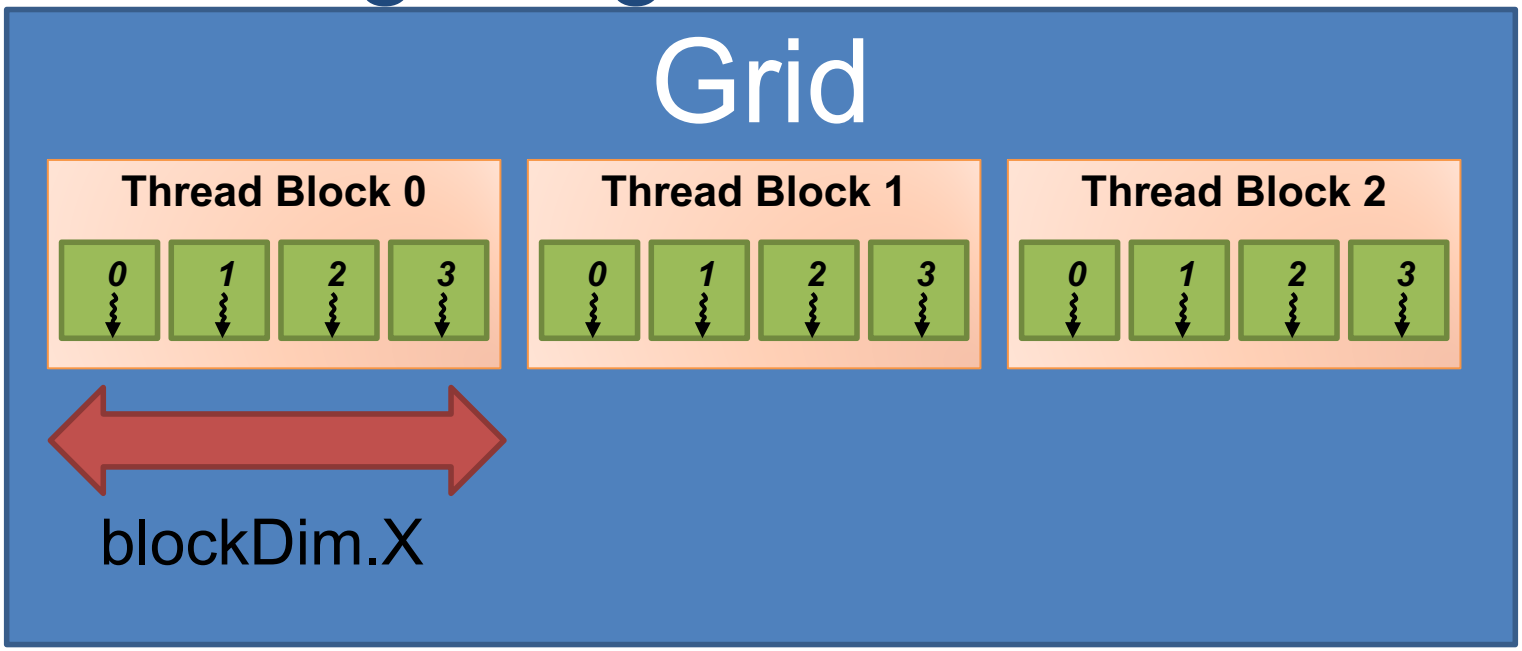
How do we parallelize this?

- What does each thread compute?
  - One addition per thread
  - Each thread deals with *different* elements
  - How do we know which element?
    - Compute a mapping of the grid to the data
      - Any mapping will do!

Vector add: kernel

```
// compute vector sum c = a + b
// each thread performs one pair-wise addition
__global__ void vector_add(float* A, float* B, float* C) {
    int i = ?
    C[i] = A[i] + B[i];
}
```

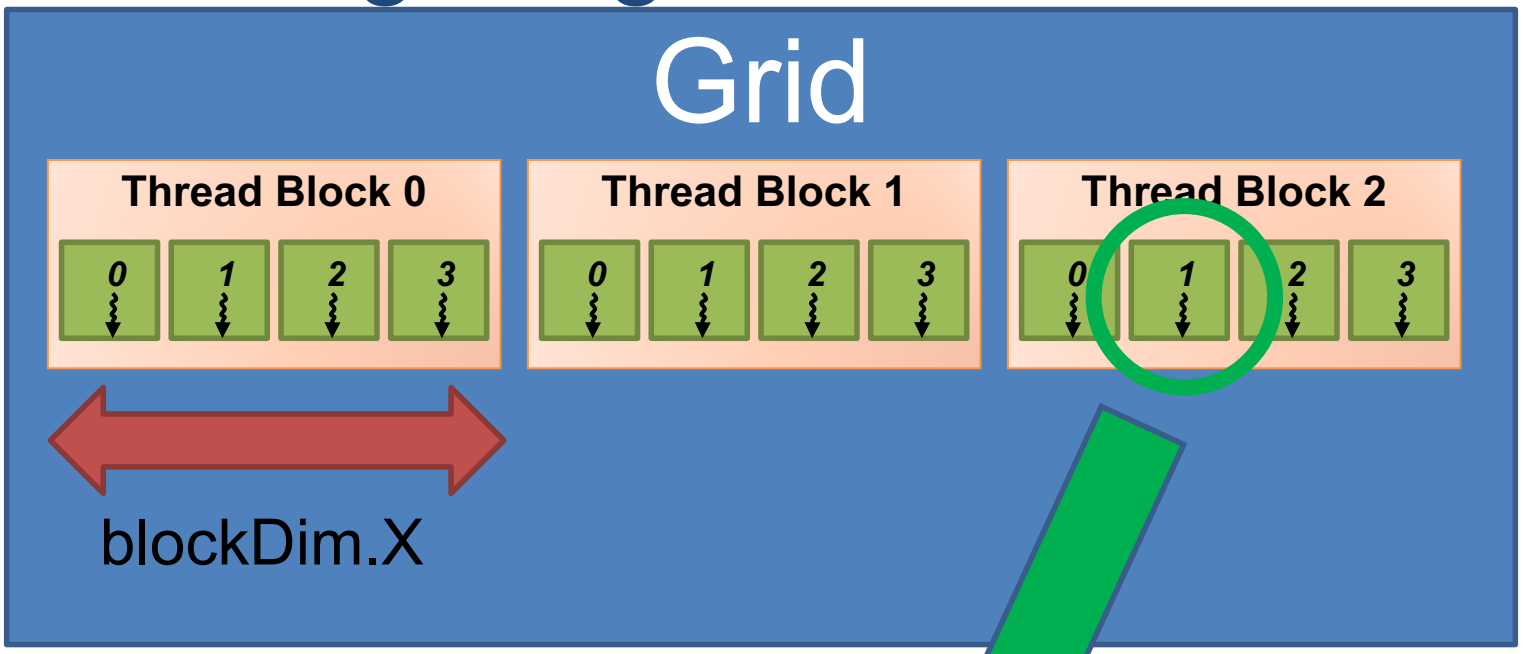
Calculating the global thread index



- "global" thread index:

```
blockDim.x * blockIdx.x + threadIdx.x;
```

Calculating the global thread index



- "global" thread index:

```
blockDim.x * blockIdx.x + threadIdx.x;
```

```
4          *          2          +          1          = 9
```

Vector add: Kernel

```
// compute vector sum c = a + b
// each thread performs one pair-wise addition
__global__ void vector_add(float* A, float* B, float* C) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

Done with the kernel!

Vector add: Launch kernel

```
// compute vector sum c = a + b
// each thread performs one pair-wise addition
__global__ void vector_add(float* A, float* B, float* C) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

GPU code

```
int main() {
  // initialization code here ...
 N = 5120;
  // launch N/256 blocks of 256 threads each
  vector_add<<< N/256, 256 >>>(deviceA, deviceB, deviceC);
  // cleanup code here ...
}
```

Host code

(can be in the same file)

Vector add: Launch kernel

```
// compute vector sum c = a + b
// each thread performs one pair-wise addition
__global__ void vector_add(float* A, float* B, float* C) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

GPU code

What if N = 5000?

Host code

```
int main() {
  // initialization code here ...
 N = 5000;
  // launch N/256 blocks of 256 threads each
  vector_add<<< N/256, 256 >>>(deviceA, deviceB, deviceC);
  // cleanup code here …}
```

(can be in the same file)

Vector add: Launch kernel

```
// compute vector sum c = a + b
// each thread performs one pair-wise addition
__global__ void vector_add(float* A, float* B, float* C) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i<N) C[i] = A[i] + B[i];
}
```

GPU code

What if N = 5000?

Host code

```
int main() {
  // initialization code here ...
 N = 5000;
  // launch N/256 blocks of 256 threads each
  vector_add<<< N/256+1,256 >>>(deviceA, deviceB, deviceC);
  // cleanup code here … }
```

(can be in the same file)

Vector add: Host

```
int main(int argc, char** argv) {
  float *hostA, *deviceA, *hostB, *deviceB, *hostC,
*deviceC;
  int size = N * sizeof(float);

  // allocate host memory
  hostA = malloc(size);
  hostB = malloc(size);
  hostC = malloc(size);

  // initialize A, B arrays here...

  // allocate device memory
  cudaMalloc(&deviceA, size);
  cudaMalloc(&deviceB, size);
  cudaMalloc(&deviceC, size);
```

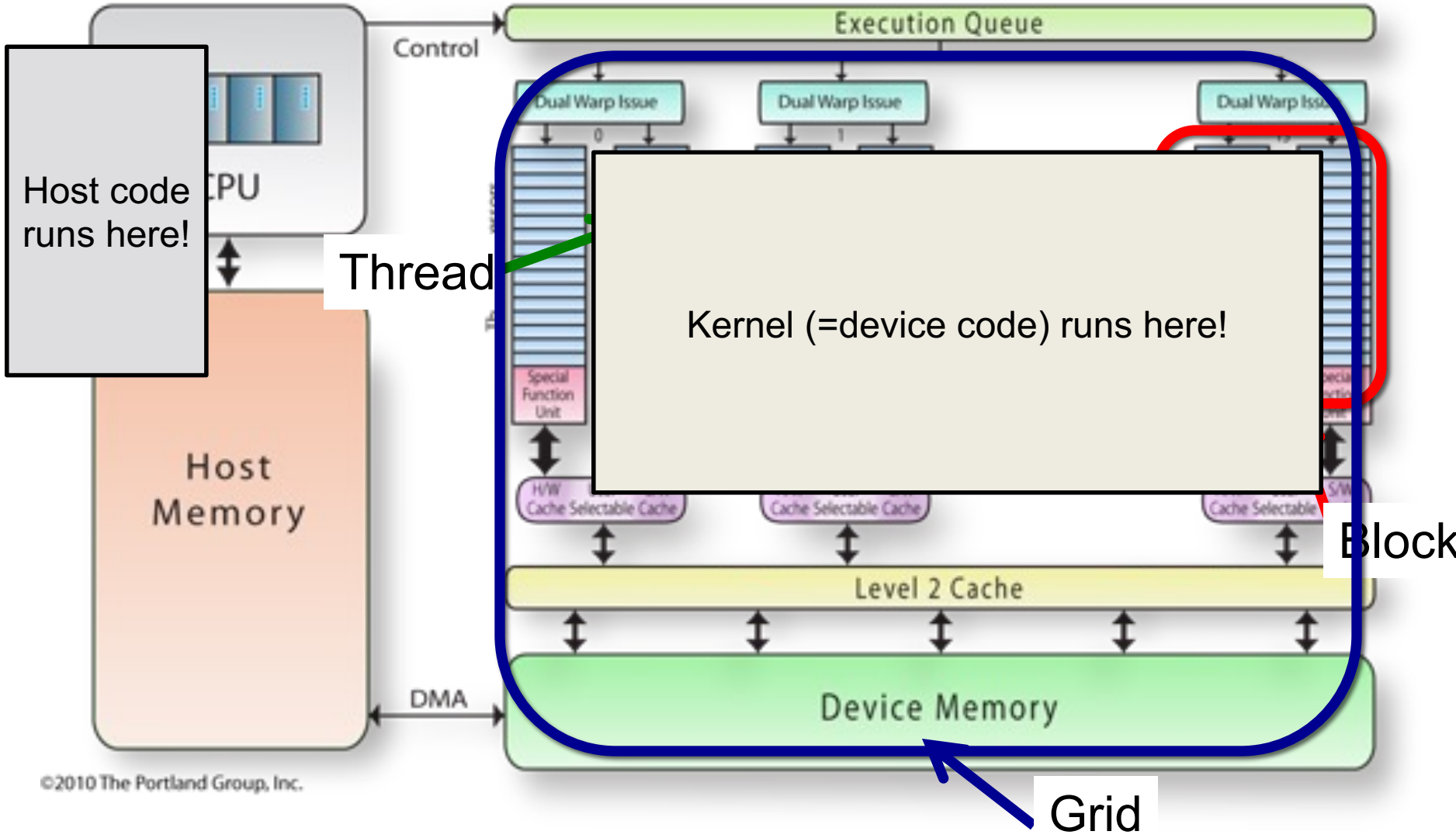
Vector add: Host

```
 // transfer the data from the host to the device
 cudaMemcpy(deviceA, hostA, size,
cudaMemcpyHostToDevice);
 cudaMemcpy(deviceB, hostB, size,
cudaMemcpyHostToDevice);

 // launch N/256 blocks of 256 threads each
 vector_add<<<N/256, 256>>>(deviceA, deviceB,
deviceC);

 // transfer the result back from the GPU to the
host
 cudaMemcpy(hostC, deviceC, size,
cudaMemcpyDeviceToHost);
}
```

Done with the host code!

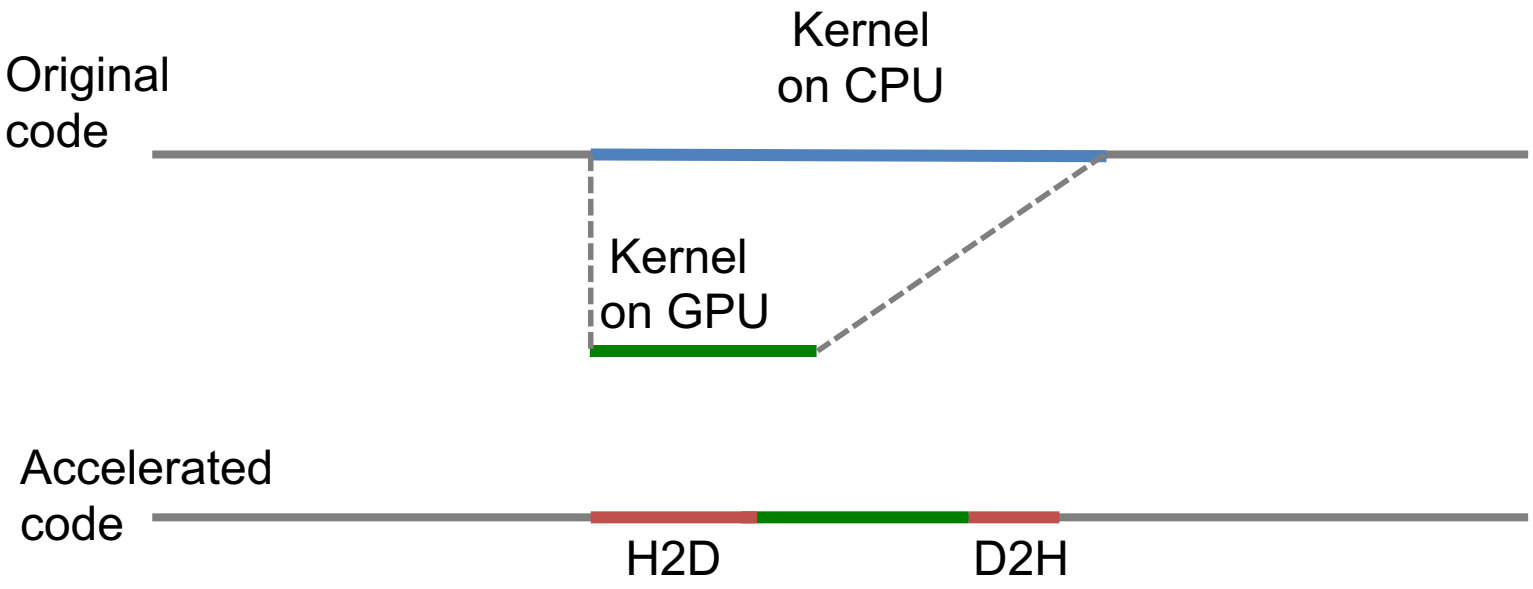
PERFORMANCE

Architecture



How to reason about performance?

- Original code: CPU
- Accelerated code: CPU + GPU



Speed-up?

Vector add: performance

```
Adding two vectors of 1048576 integer elements.
vector-add (sequential):                    = 0.00786386 seconds
vector-add (kernel):              = 0.000302897 seconds
vector-add (memory):              = 0.00911151 seconds
results OK!
Wall Clock (sequential):                    = 0.0214828 seconds
```

- Wall-clock time: 0.0215 s
  - Non accelerated part: 0.0136s
- Sequential execution time: 0.0079 s
- Accelerated execution time: 0.0003
- D2H + H2D transfer time: 0.0091s
- Kernel speed-up ~26x

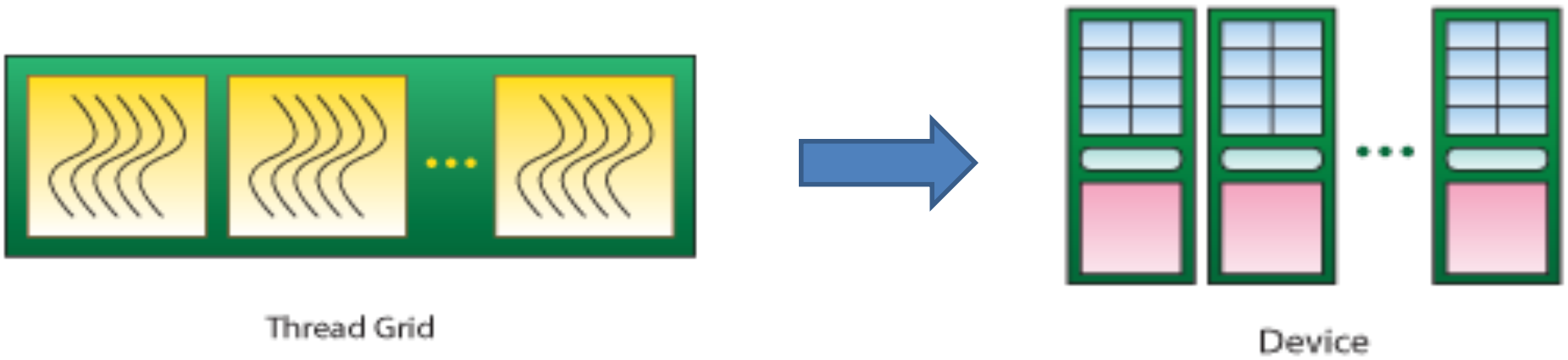
- Speedup: 0.0215 / (0.0136 + 0.0091 + 0.0003) = 0.93x !!

ADVANCED CONCEPTS

Scheduling

- Application = Grid of blocks (1D, 2D, 3D)
- Block = Collection of threads (1D, 2D, 3D)

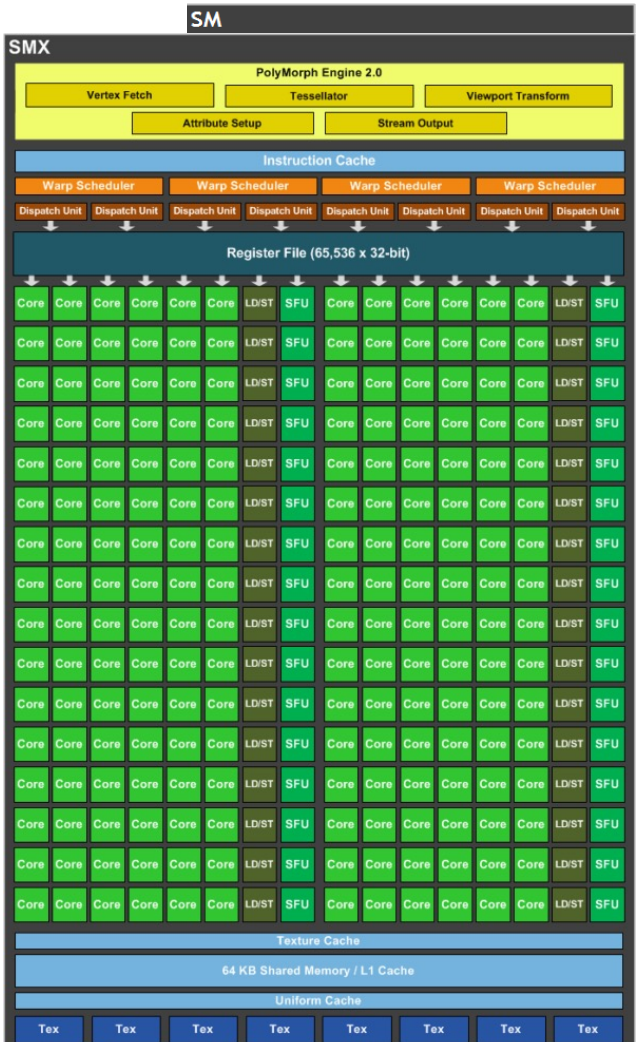
- *Scheduling* =
  - *Mapping* and *ordering* the application blocks on hardware resources.
- *Context switching* =
  - Swapping *the state and data* of blocks that replace each other



Block Scheduling

- One block runs on one SM
  - Dispatched by the GigaThread Engine
  - Runs to completion without preemption

- *Undefined block execution order*
  - Any order should be valid
  - May run concurrently OR sequentially

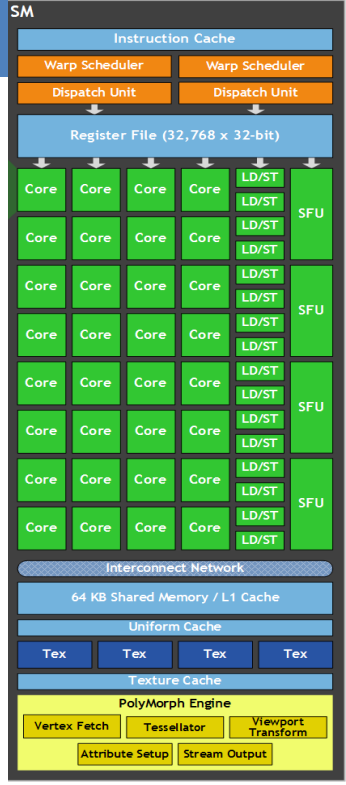


Warp Scheduling

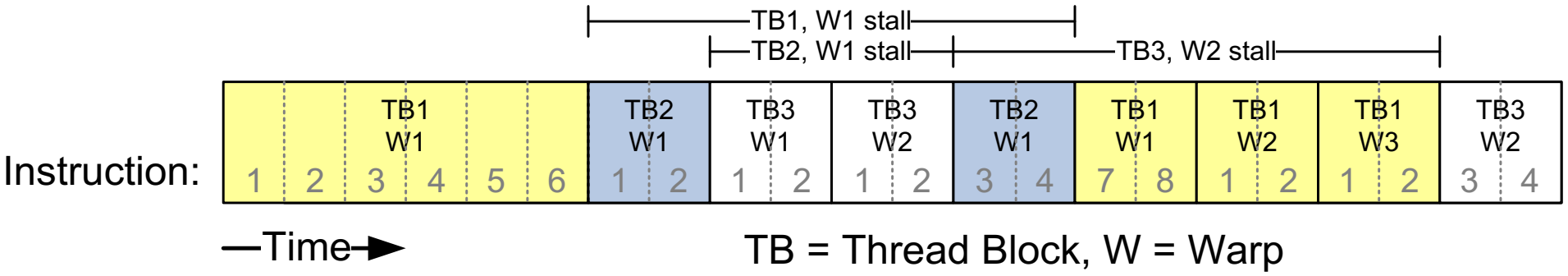
- Blocks are divided into warps
  - “wavefronts” for AMD
- Threads in a warp execute in lock-step

- Warps’ threads are ***mapped*** on cores
  - Concurrent warps per SM are limited by the number of cores

- *Undefined warp execution order*
  - Any order should be valid
  - May run concurrently OR sequentially



Warp scheduling: an example



- Key things to remember
  - Very fast context switching
  - Stalled warps are immediately replaced
  - Eligible warps = warps ready to execute (all data is available)
  - Warp-to-execute is selected from eligible warps
  - No fairness guarantee!

Stalling warps

- What happens if all warps are stalled?
  - No instruction issued → performance lost
- Most common reason for stalling?
  - Waiting on global memory
- If your code reads global memory every couple of instructions
  - You should try to maximize **occupancy**

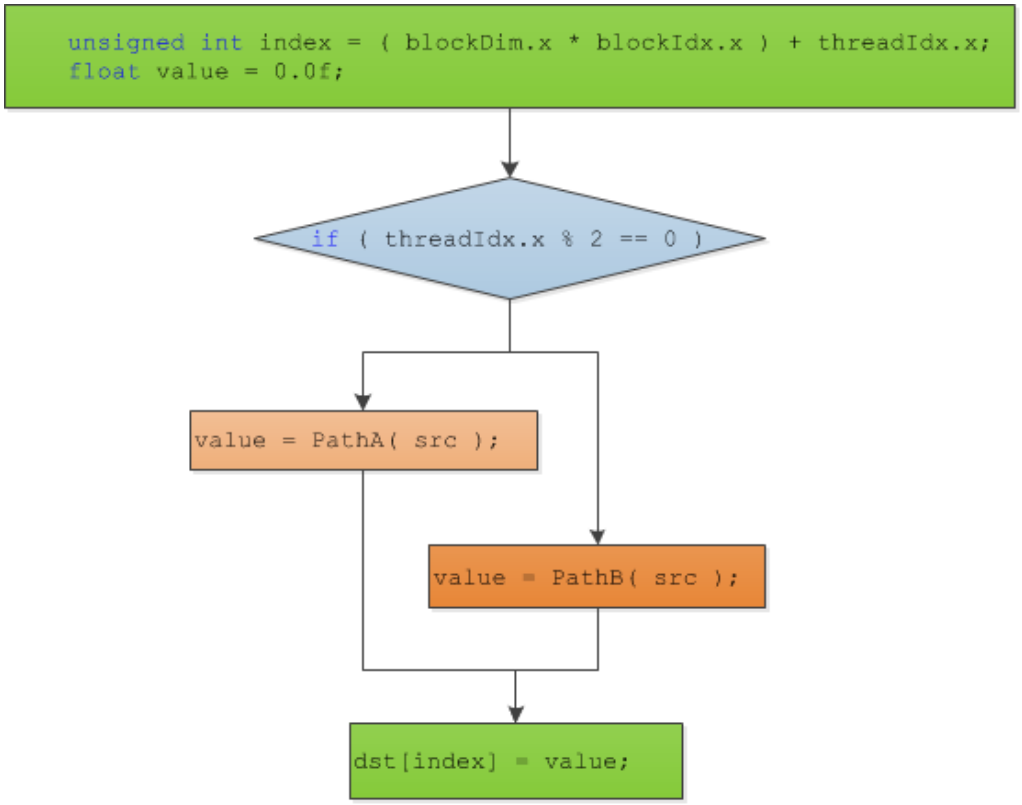
CUDA: THREAD DIVERGENCE

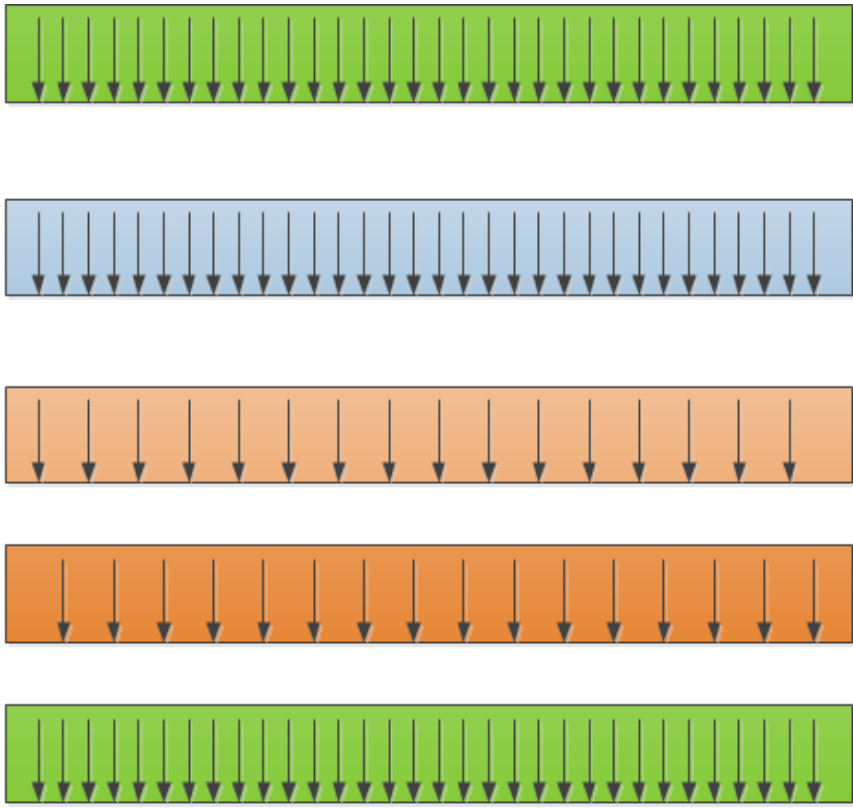
Thread divergence

“I heard GPU branching is expensive. Is this true?”

```
__global__ void Divergence(float* dst,float* src )
{
    float value = 0.0f;

    if ( threadIdx.x % 2 == 0 )
// active threads : 50%
        value = src[0] + 5.0f;
    else
// active threads : 50%
        value = src[0] – 5.0f;

    dst[index] = value;
}
```

Execution

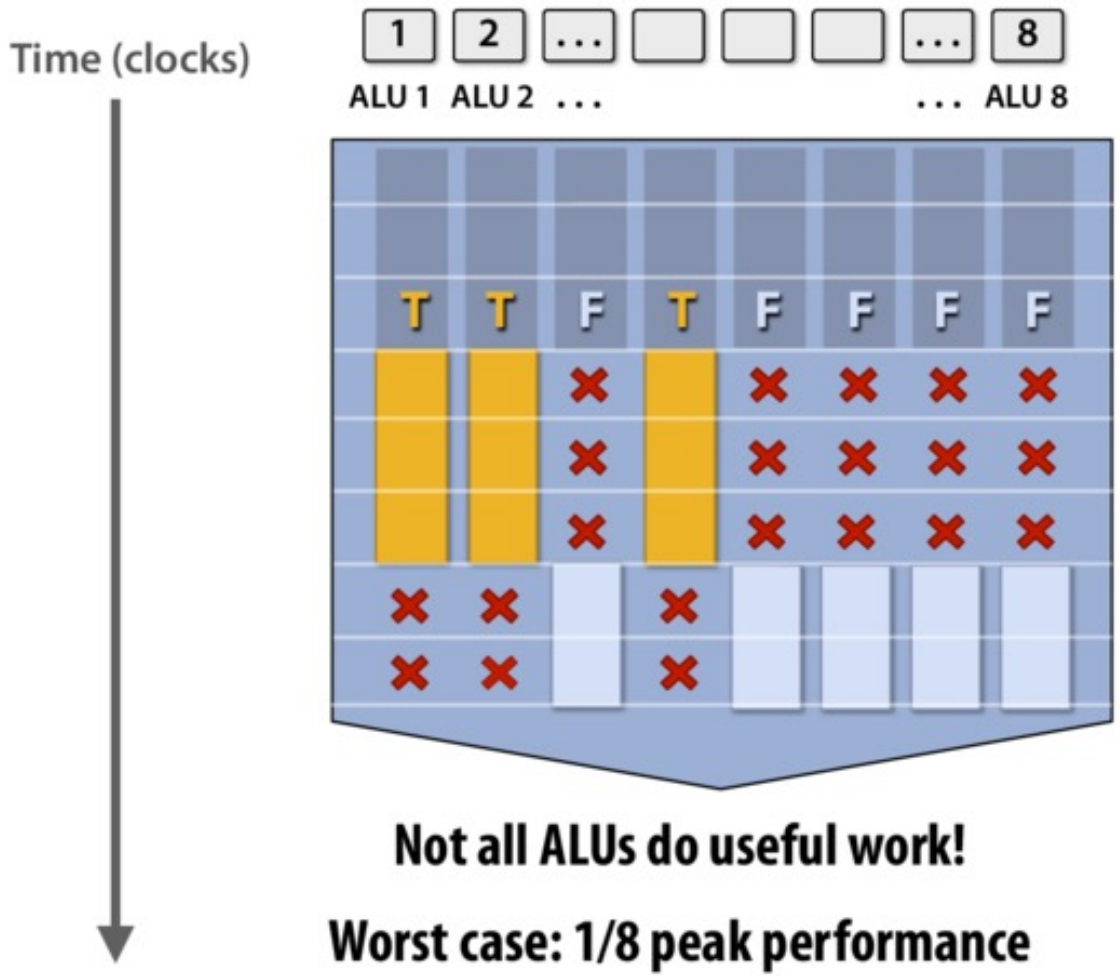




Worst case performance loss:

50% compared with the non divergent case.

Another example



(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

```
<unconditional code>

float x = A[i];

if (x > 0) {
   float tmp = exp(x,5.f);
   tmp *= kMyConst1;
   x = tmp + kMyConst2;
} else {
   float tmp = kMyConst1;
   x = 2.f * tmp;
}

<resume unconditional code>

result[i] = x;
```

Performance penalty?

- Depends on the amount of divergence
  - Worst case: 1/32 performance
    - When each thread does something different
- Depends on whether branching is data- or ID- dependent
  - If ID – consider grouping threads differently
  - If data – consider sorting
- Non-diverging warps => NO performance penalty
  - In this case, branches are not expensive …

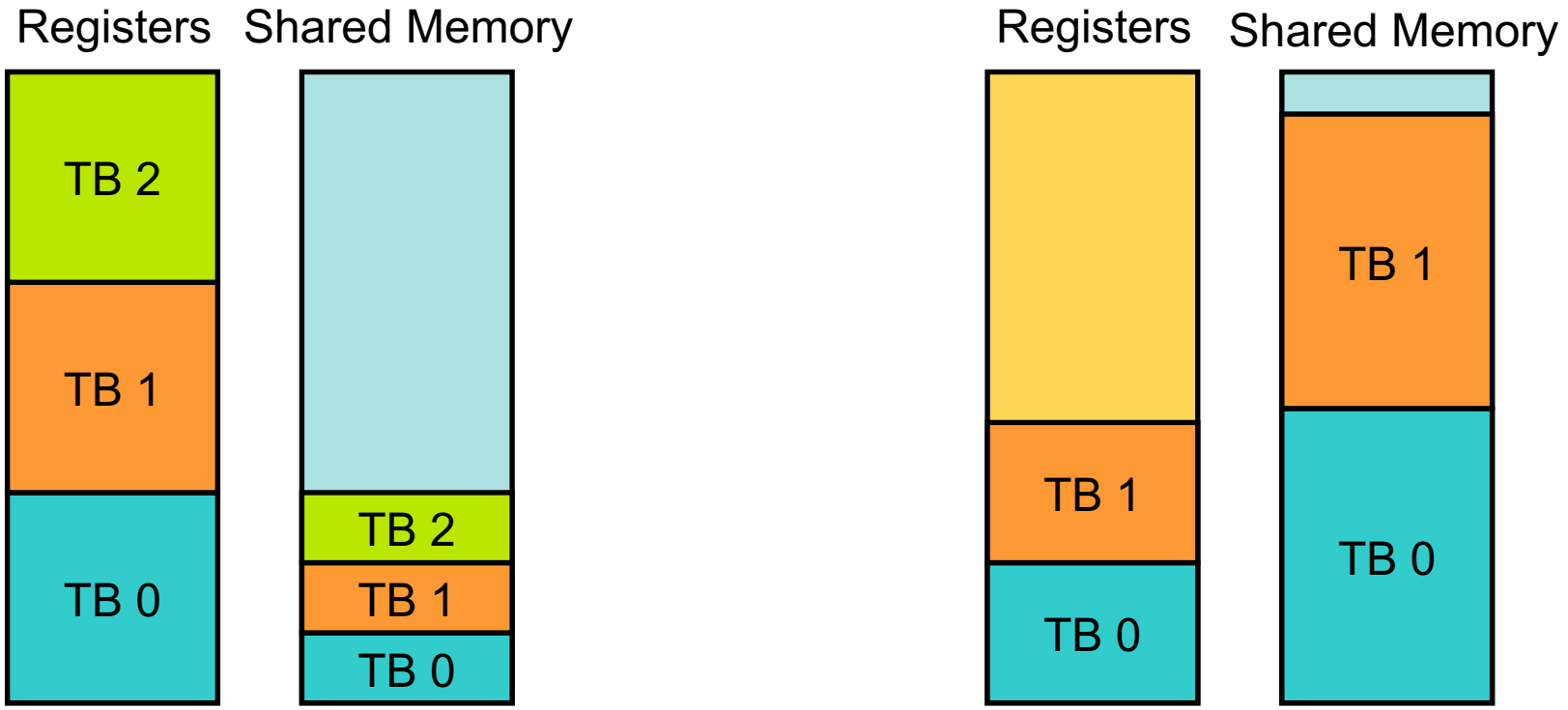
CUDA: OCCUPANCY

Occupancy

Occupancy = Active Warps / Maximum Active Warps

- Remember: resources are allocated for the entire block!
- Resources are finite
  - Utilizing too many resources per thread may limit the occupancy
- Potential occupancy limiters:
  - Register usage
  - Shared memory usage
  - Block size

Resource Limits



- Pool of registers and shared memory per SM
  - Each thread block grabs registers & shared memory
  - If one or the other is fully utilized => no more thread blocks

How do you know what you're using?

- Use compiler flags to get register and shared memory usage
  - "`nvcc -Xptxas -v`"
- Use the NVIDIA Profiler
- Plug those numbers into CUDA Occupancy Calculator

- Maximize occupancy for improved performance
  - Empirical rule! Don't overuse!

CUDA_Occupancy_calculator.xlsm - Microsoft Excel

Security Warning Macros have been disabled. Options...

MyRegCount | 25

1.) Select Compute Capability (click):	1.3	(Help)	
2.) Enter your resource usage:			
Threads Per Block	128	(Help)	
Registers Per Thread	25		
Shared Memory Per Block (bytes)	640		
(Don't edit anything below this line)			
3.) GPU Occupancy Data is displayed here and in the graphs:			
Active Threads per Multiprocessor	512	(Help)	
Active Warps per Multiprocessor	16		
Active Thread Blocks per Multiprocessor	4		
Occupancy of each Multiprocessor	50%		

Physical Limits for GPU Compute Capability:	1.3
Threads per Warp	32
Warps per Multiprocessor	32
Threads per Multiprocessor	1024
Thread Blocks per Multiprocessor	8
Total # of 32-bit registers per Multiprocessor	16384
Register allocation unit size	512
Register allocation granularity	block
Shared Memory per Multiprocessor (bytes)	16384
Shared Memory Allocation unit size	512
Warp allocation granularity (for register allocation)	2

Allocation Per Thread Block	
Warps	4
Registers	3584
Shared Memory	1024

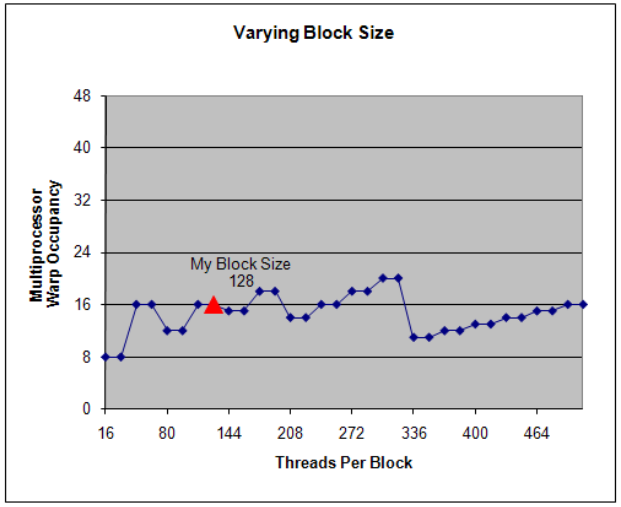
These data are used in computing the occupancy data in blue

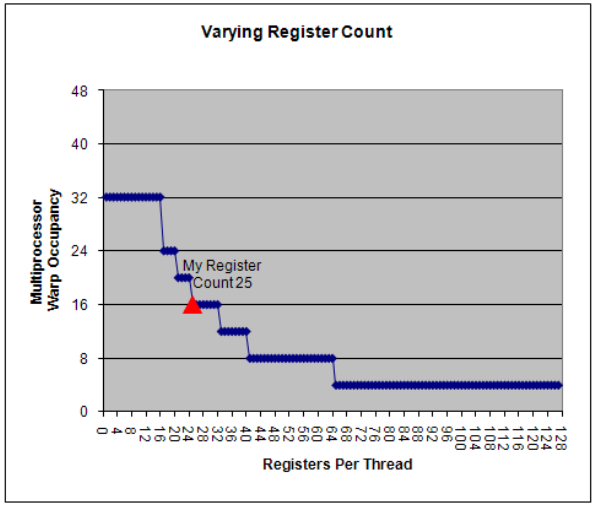
Maximum Thread Blocks Per Multiprocessor	Blocks
Limited by Max Warps / Blocks per Multiprocessor	8
Limited by Registers per Multiprocessor	4
Limited by Shared Memory per Multiprocessor	16

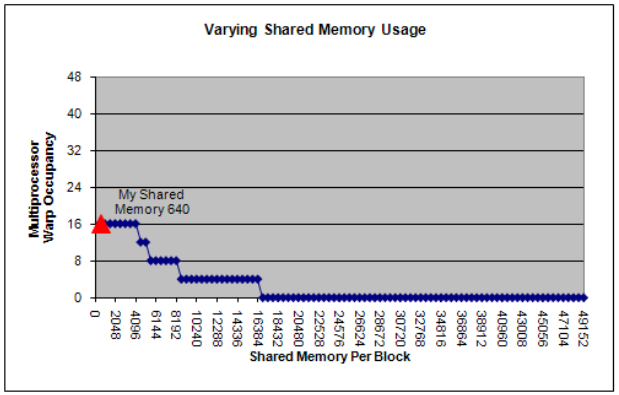
Thread Block Limit Per Multiprocessor highlighted RED

CUDA Occupancy Calculator	
Version:	2.0
Copyright and License	

The other data points represent the range of possible block sizes, register counts, and shared memory allocation.







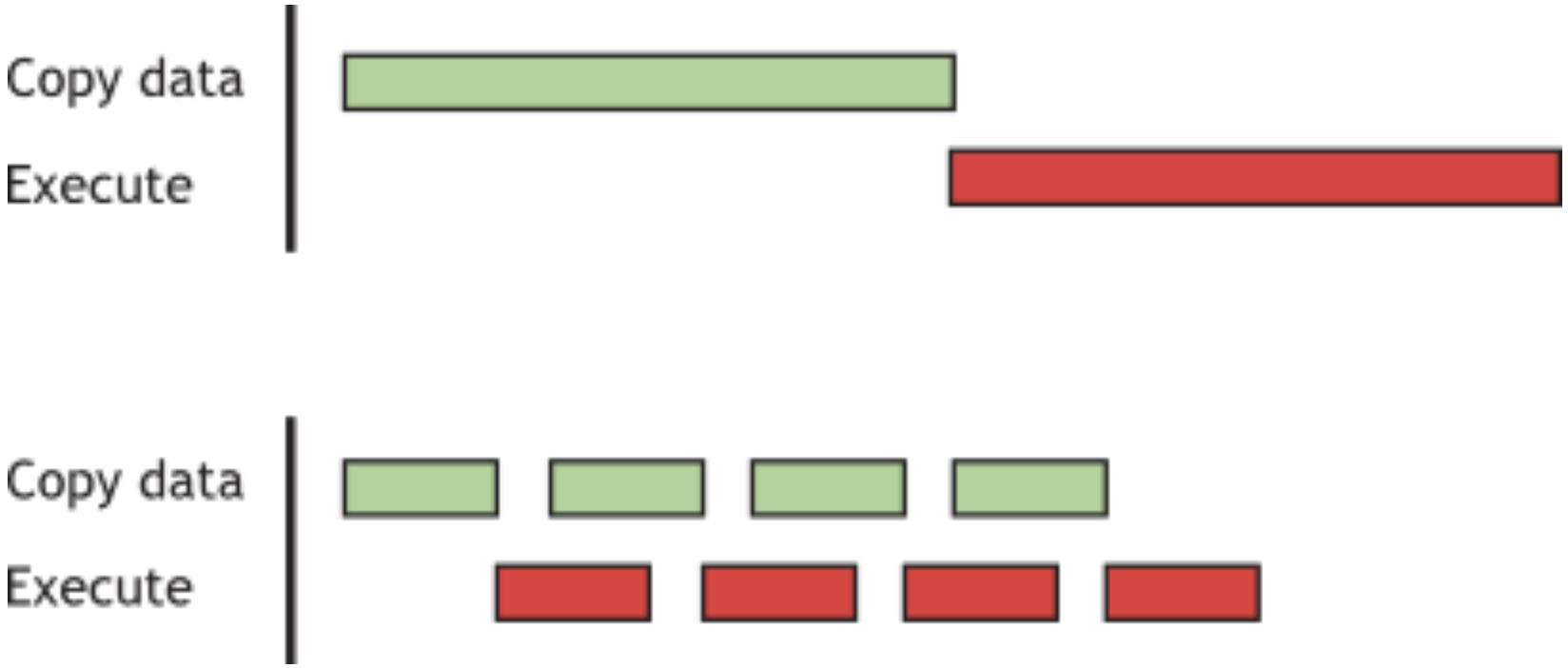
Calculator | Help | GPU Data | Copyright & License

Ready

CUDA: STREAMS

Overlap Computation and Communication

- Main idea: while executing a kernel, bring data in for the next kernel:



What are streams?

- **Stream** = a sequence of operations that execute on the device in the order in which they are issued by the host code.

- Same stream: In-Order execution
- Different streams: Out-of-Order execution

- Default stream = Synchronizing stream
  - No operation in the default stream can begin until all previously issued operations in any stream on the device have completed.
  - An operation in the default stream must complete before any other operation in any stream on the device can begin.

Default stream: example

```
cudaMemcpy(d_a, a, numBytes,cudaMemcpyHostToDevice);
 increment<<<1,N>>>(d_a);
 CpuFunction(b);
cudaMemcpy(a, d_a, numBytes,cudaMemcpyDeviceToHost);
```

- All operations happen in the same stream
- Device (GPU)
  - Synchronous execution
    - all operations execute (in order), one after the previous has finished
  - Unaware of `CpuFunction()`
- Host (CPU)
  - Launches `increment` and regains control
  - *May* execute `CpuFunction` *before* increment has finished
  - Final copy starts *after* both `increment` and `CpuFunction()` have finished

Non-default streams

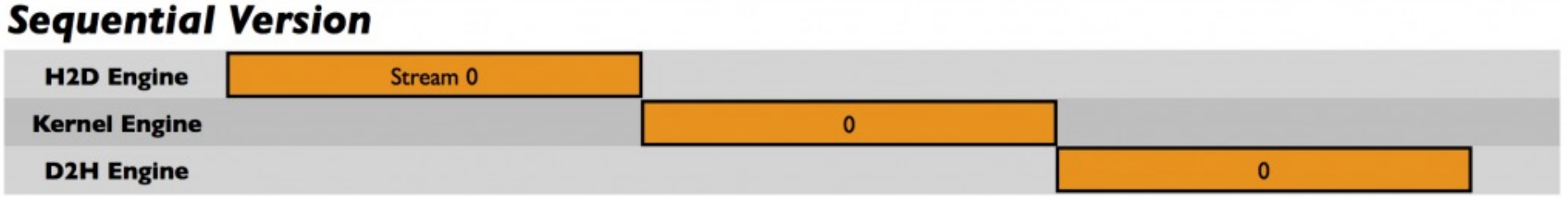
- Enable asynchronous execution and overlaps
  - Require special creation/deletion of streams
    - `cudaStreamCreate(&stream1)`
    - `cudaStreamDestroy(stream1)`
  - Special memory operations
    - `cudaMemcpyAsync(deviceMem, hostMem, size, cudaMemcpyHostToDevice, stream1)`
  - Special kernel parameter (the 4th one)
    - `increment<<<1, N, 0, stream1>>>(d_a)`
- Synchronization
  - All streams
    - `cudaDeviceSynchronize()`
  - Specific stream:
    - cudaStreamSyncrhonize(stream1)

Computation vs. communication

```
//Single stream, numBytes = 16M, numElements = 4M
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
kernel<<blocks,threads>>(d_a, firstElement);
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

C1060 (pre-Fermi): 12.9ms

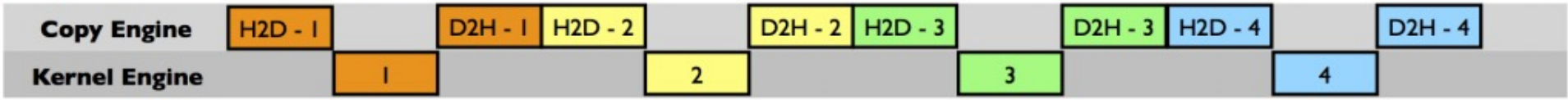
C2050 (Fermi): 9.9ms



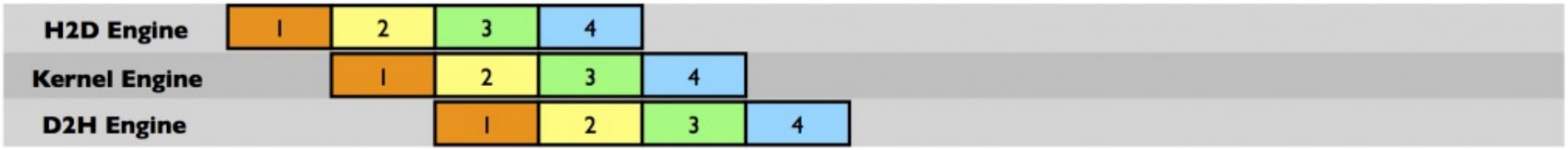
Computation-communication overlap[1]*

```
for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  cudaMemcpyAsync(&d_a[offset], &a[offset], streamBytes,
stream[i]);
  kernel<<blocks,threads,0,stream[i]>>(d_a, offset);
  cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes,
stream[i]);
}
```

C1060 (pre-Fermi): 13.63 ms (worse than sequential)



C2050 (Fermi): 5.73 ms (better than sequential)



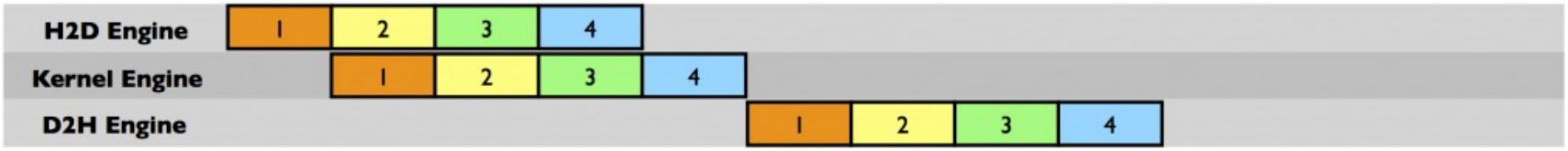
https://github.com/parallel-forall/code-samples/blob/master/series/cuda-cpp/overlap-data-transfers/async

Computation-communication overlap[2]*

```
for (int i = 0; i < nStreams; ++i) offset[i]=i * streamSize;
for (int i = 0; i < nStreams; ++i)
   cudaMemcpyAsync(&d_a[offset[i]], &a[offset[i]], streamBytes,
      cudaMemcpyHostToDevice, stream[i]);

for (int i = 0; i < nStreams; ++i)
      kernel<<blocks,threads,0,stream[i]>>(d_a, offset);

for (int i = 0; i < nStreams; ++i)
   cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes,
      cudaMemcpyDeviceToHost, stream[i]);
```

C1060 (pre-Fermi): 8.84 ms (better than sequential)

C2050 (Fermi): 7.59 ms (better than sequential, worse than v1)



SHARING & SYNCHRONIZATION

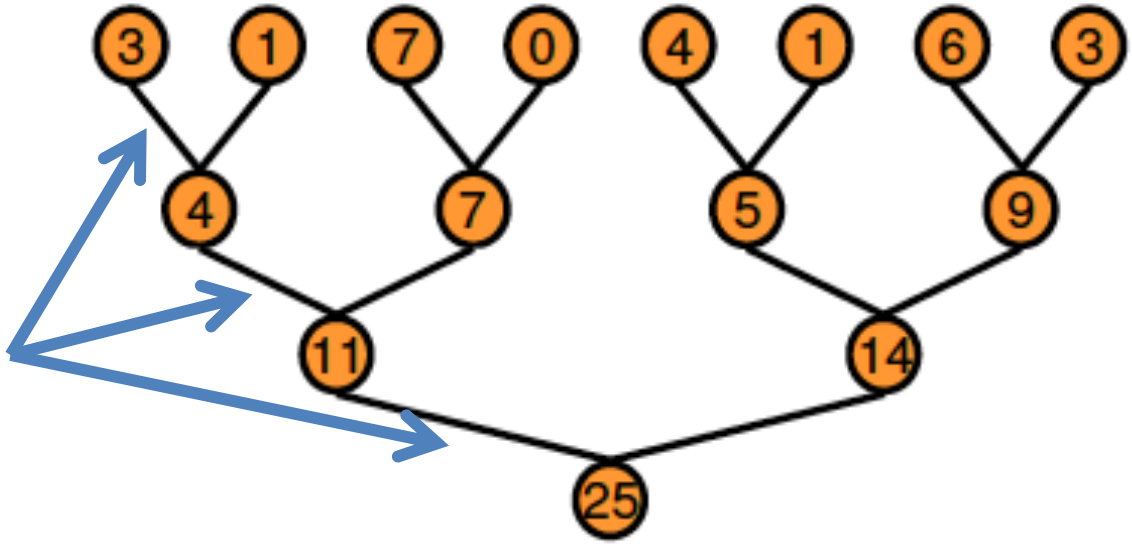
Global synchronization

- We launch many more blocks than physical SM's.
- Each block might/should have more threads than the SM's cores

```
__global__ void my_kernel() {
    step1; // compute some values in a global array
    // wait for *all* threads to finish
    __my_global_barrier();
    step2; // use the array
}

int main() {
  dim3 blockSize(32, 32);
  dim3 gridSize(100, 100, 100);
  my_kernel<<<gridDim, blockDim>>>();
}
```

An example: parallel reduction

- Given an array with data, “reduce” it to a single value
  - The sum of all elements
  - The min/max of all elements
- Sequentially: O(n)
- In parallel?
  - Tree-based algo.
    - O(log n)
  - Requires a barrier after each step

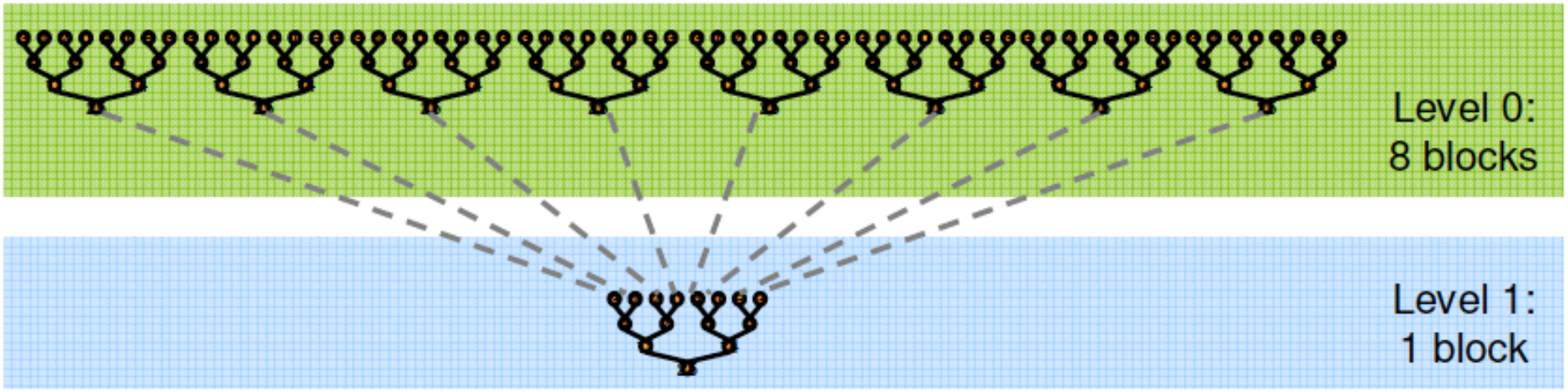


Parallel reduction in CUDA*

- One element per thread
- We need to use multiple blocks
  - Large arrays
  - Good GPU utilization
- We need global synchronization
  - Synchronization inside blocks is possible.
  - Synchronization between blocks is not possible!
- Solution: decompose into multiple kernels
  - Kernel launch serves as a global synchronization point
  - Kernel launch has negligible HW overhead, low SW overhead

*http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf

Parallel reduction in CUDA*



- Other optimizations
  - Use shared memory
  - Increase granularity
  - Avoid branching
  - Improve data access patterns

*http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf

Memory consistency

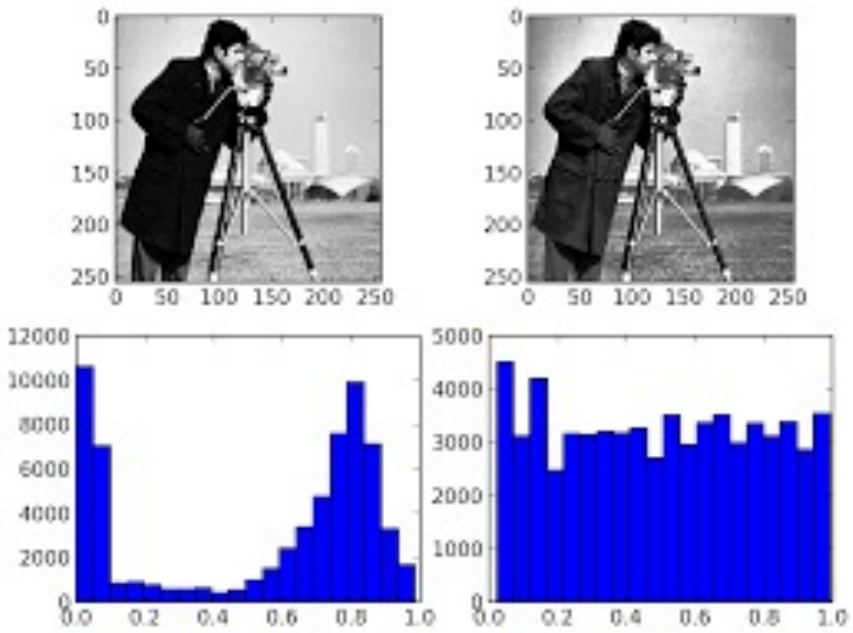
- Device (global) memory is not serially consistent
  - No ordering guarantees in shared/global memory Rd/Wr
- Share data between streaming multiprocessors
  - Potential write hazards!
- Use **atomics** to avoid data races for global (and shared) memory variables!
- Evolution:
  - Fermi has reasonable atomics for both shared and global memory
  - Kepler increases *global memory atomics* performance vs. Fermi
  - Maxwell uses native support for shared memory atomics
    - Much faster than Fermi and Kepler

Atomics

- Guarantee that only a single thread has access to a piece of memory during an operation
  - Ordering is still arbitrary
- Different types of atomic instructions
  - Add, Sub, Exch, Min, Max, Inc, Dec, CAS, And, Or, Xor
- Both for device memory and shared memory
- Much more expensive than load + operation + store

An example: image histogram

- The histogram of an image: the distribution of the pixels in the image.
  - In practice: count the pixels of each color
  - Useful image feature detection for image recognition.



An example: image histogram

```
// Determine frequency of colors in a picture.
// Colors have already been converted into integers
// between 0 and 255.
// Each thread looks at one pixel,
// and increments a counter

__global__ void histogram(int* colors, int* buckets)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int c = colors[i];
    buckets[c] += 1;  // incorrect!
}
```

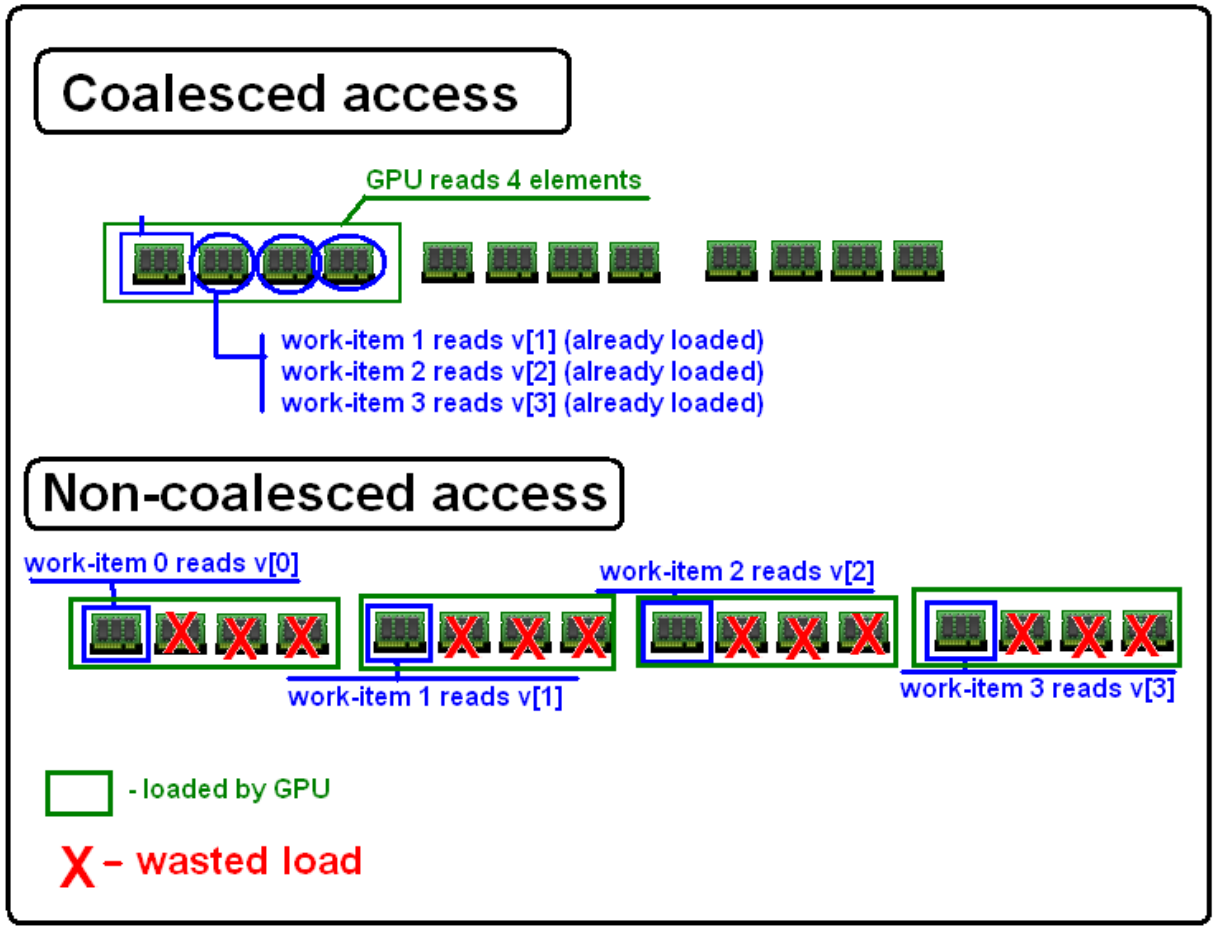
An example: image histogram

```
// Determine frequency of colors in a picture.
// Colors have already been converted into integers
// between 0 and 255.
// Each thread looks at one pixel,
// and increments a counter atomically

__global__ void histogram(int* colors, int* buckets)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int c = colors[i];
    atomicAdd(&buckets[c], 1);
}
```

CUDA: MEMORY COALESCING

Memory Coalescing

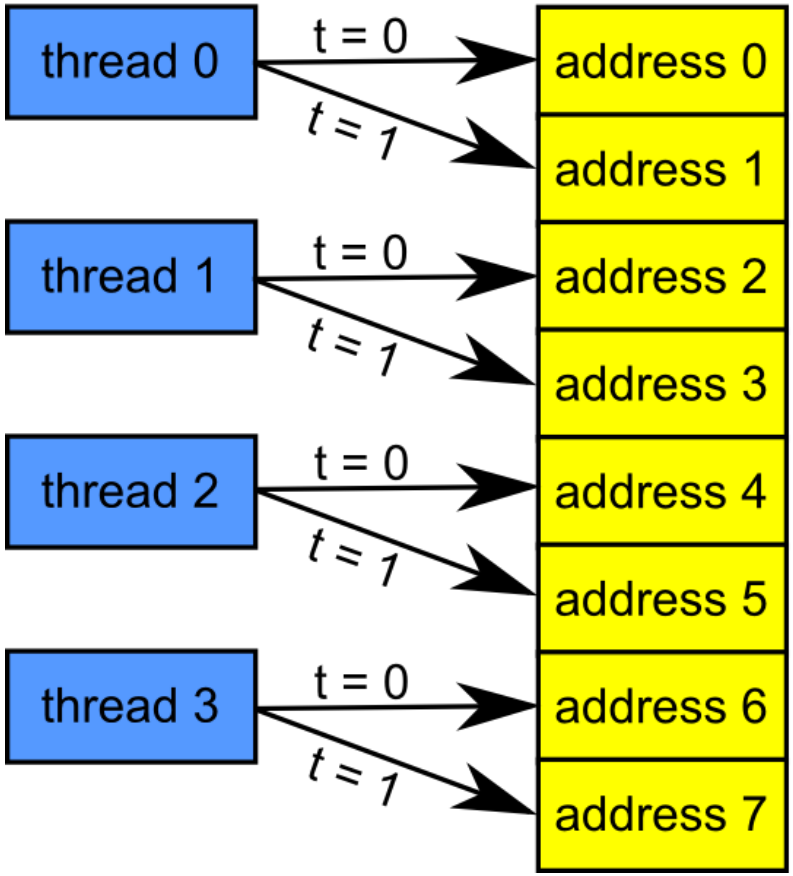
- **Memory coalescing** refers to combining multiple **memory** accesses into a single transaction



Caching vs. Coalescing

traditional multi-core

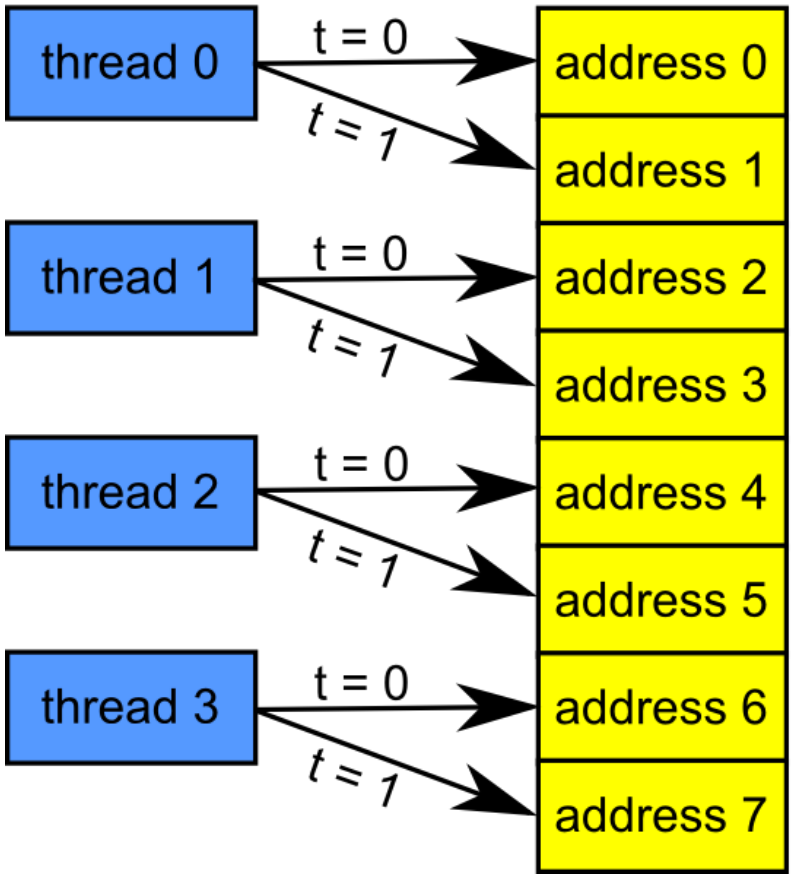
optimal memory access pattern



Caching

Caching vs. Coalescing

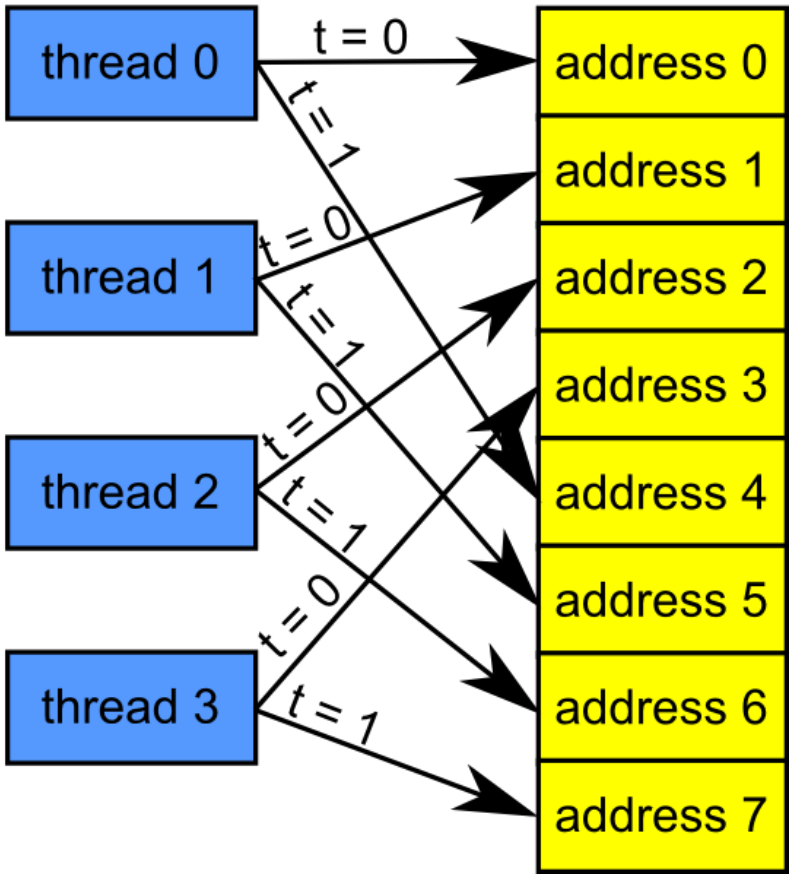
traditional multi-core
optimal memory access pattern



Caching

vs.

many-core GPU
optimal memory access pattern



Coalescin

Consider the stride of your accesses

	Input[0]	Input[1]	Input[2]	Input[3]	Input[4]	Input[5]	Input[6]	Input[7]
Stride1	T0	T1	T2	T3	T4	T5	T6	T7
Stride2		T0		T1		T2		T3
“random”			T7		T1			T4

```
__global__ void foo(int* input, float3* input2) {
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  // Stride 1, full bandwidth used!
  int a = input[i];
  // Stride 2, 50% of the bandwidth is wasted
  int b = input[2*i+1];
  // “Random” stride – ?? up to 7/8 bandwidth wasted
  int c = input[f(i)];
}
```

Example: Array of Structures (AoS)

```
Struct AoS{
    int key;
    int value;
    int flag;
};

record *d_AoS_data;
cudaMalloc((void**)&d_AoS_data, ...);

kernel {
 threadID = blockDim.x * blockIdx.x + threadIdx.x;
 // …
 d_AoS_data[threadID].value += i; // wastes bandwidth!
 // …
}
```

Example: Structure of Arrays (SoA)

```
Struct SoA {
    int* keys;
    int* values;
    int* flags;
};

SoA d_SoA_data;
cudaMalloc((void**)&d_SoA_data.keys, ...);
cudaMalloc((void**)&d_SoA_data.values, ...);
cudaMalloc((void**)&d_SoA_data.flags, ...);

kernel {
  threadID = blockDim.x * blockIdx.x + threadIdx.x;
…
  d_SoA_data.values[threadID] += i; // full
 bandwidth!
… }
```

Memory Coalescing*

- Group memory accesses in as few memory transactions as possible.
  - 128-byte
- Stride 1 access patterns are preferred!
  - Other patterns can still get benefits
- Structure of arrays is often better than array of structures
- Unpredictable/ irregular access patterns
  - Case-by-case performance impact
- No coalescing => performance loss ~10x or more !
  - Caching might improve this impact …

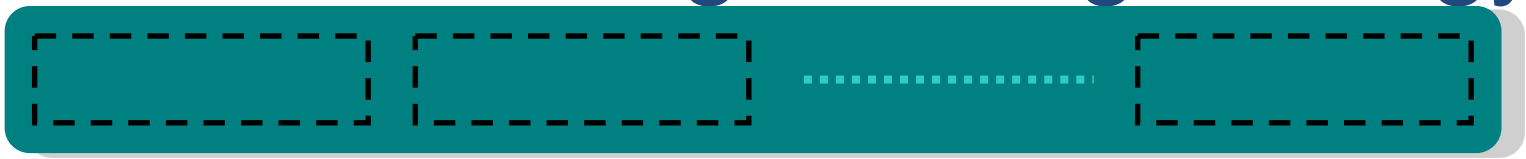
*http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#axzz3nJ0kBsWe

CUDA: USING SHARED MEMORY

Using shared memory

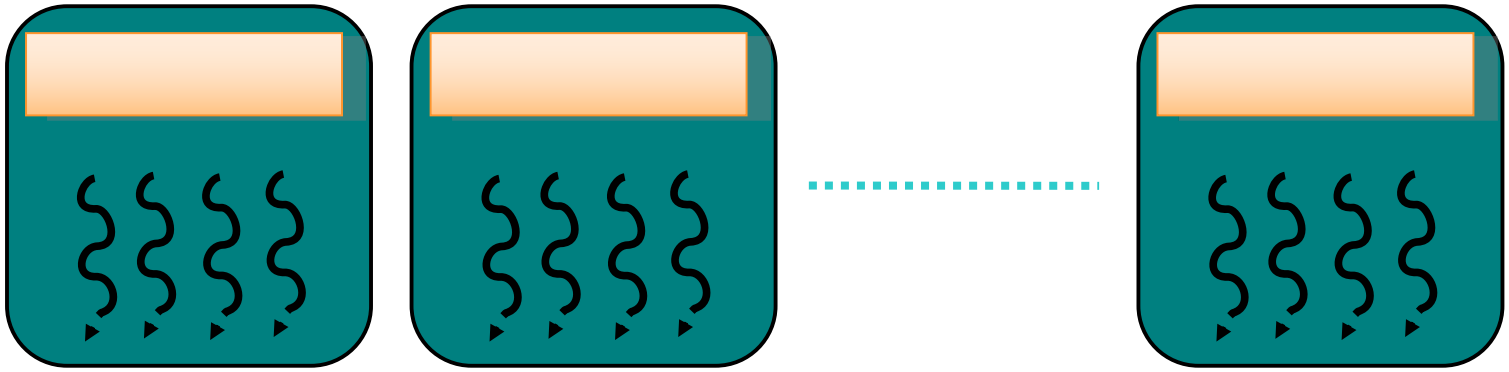
- Equivalent with providing software caching
  - Explicit: Load data to be re-used in shared memory
  - Use it for computation
  - Explicit: Store results back to global memory
- All threads in a block share memory
  - Load/Store: using all threads
  - Barrier: __syncthreads
    - Guard against using uninitialized data – not all threads have finished loading data to shared memory
    - Guard against corrupting live data – not all threads have finished computing

A Common Programming Strategy



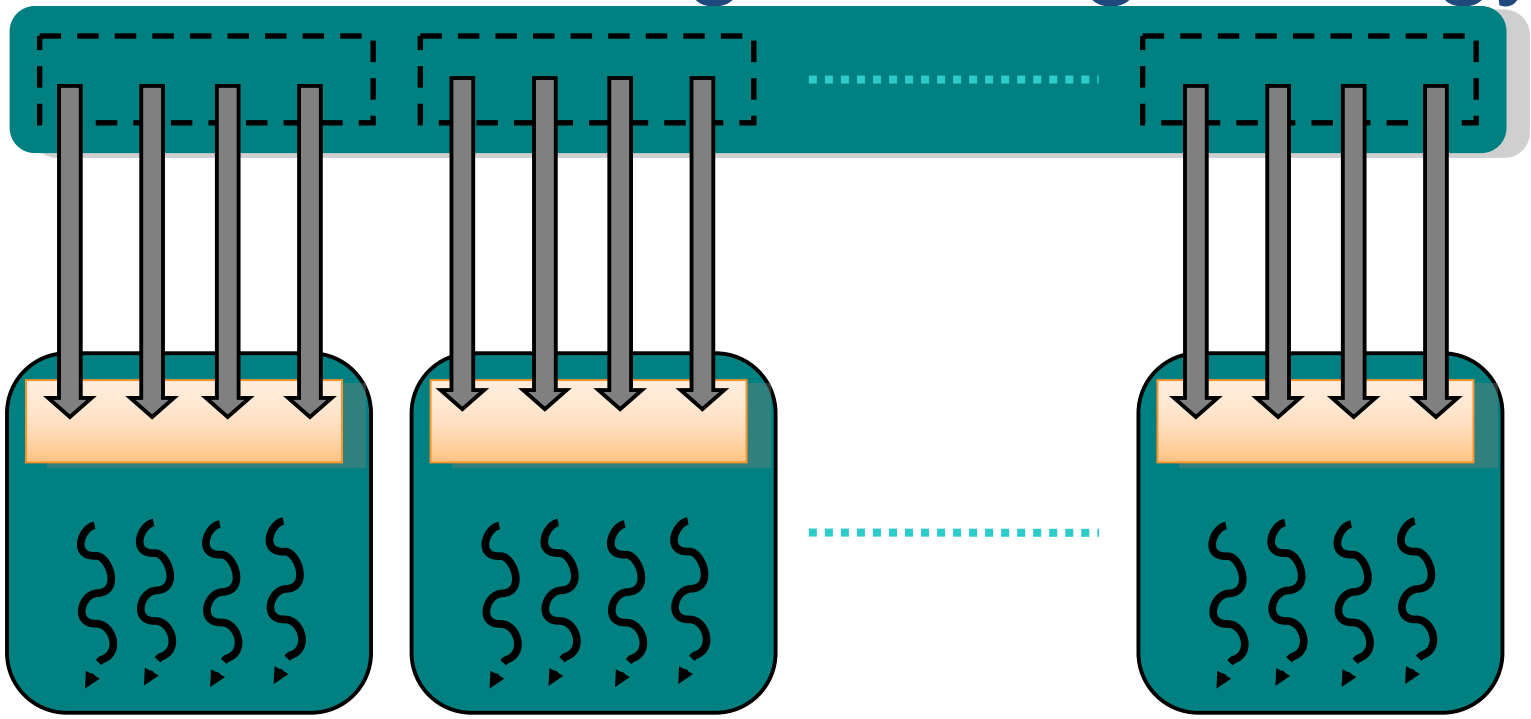
- Partition data into subsets that fit into shared memory

A Common Programming Strategy



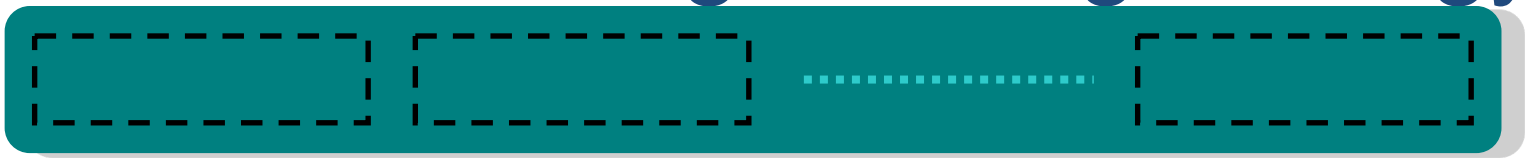
- Handle each data subset with one thread block

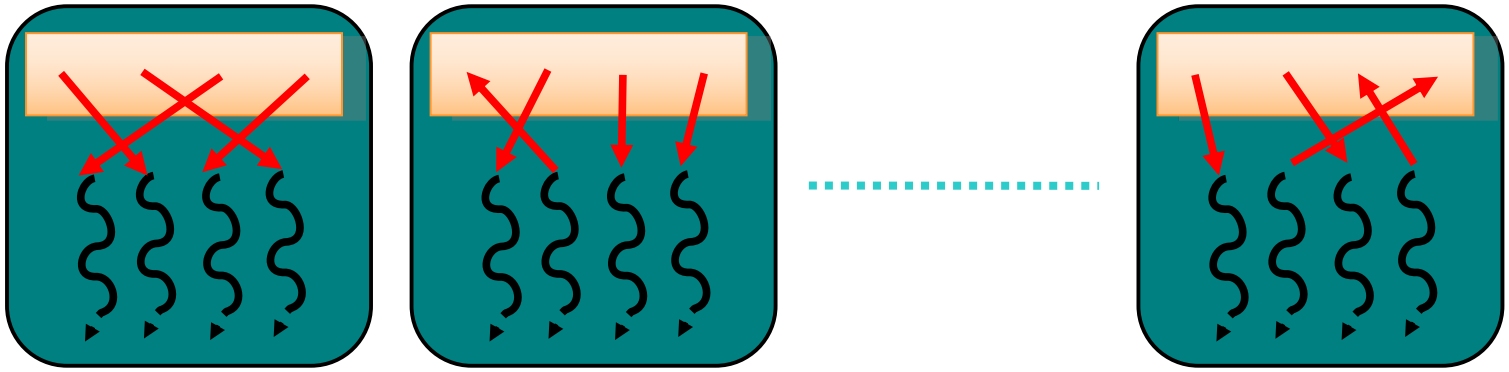
A Common Programming Strategy



- Load the subset from device memory to shared memory, using multiple threads to exploit memory-level parallelism

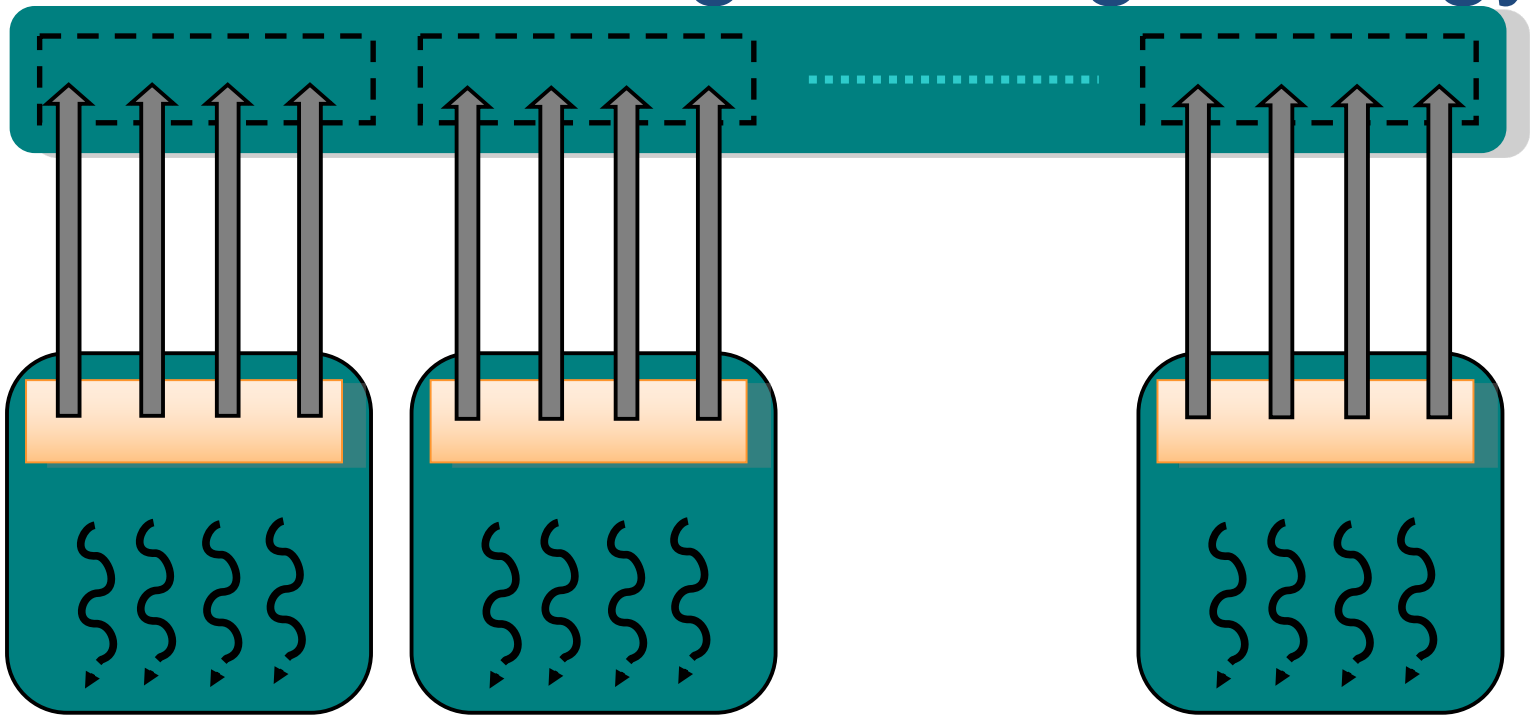
A Common Programming Strategy





- Perform the computation on the subset from shared memory

A Common Programming Strategy



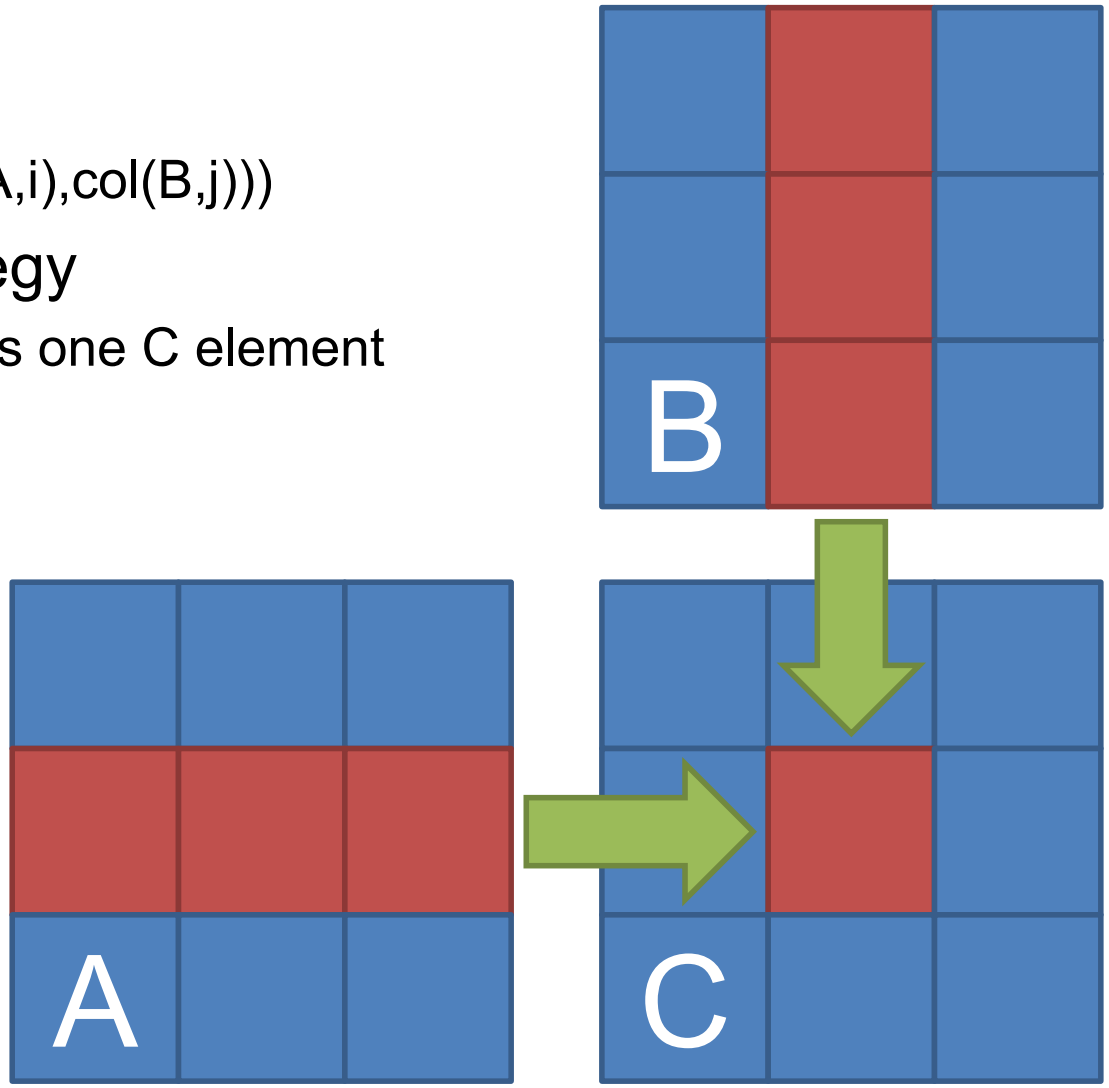
- Copy the result from shared memory back to device memory

Caches vs. Shared Memory

- Since Fermi, NVIDIA GPUs feature BOTH hardware **L1 caches** and **shared memory** per SM
  - They share the same space
    - ¾ Cache + ¼ Shared Memory OR
    - ¼ Cache + ¾ Shared Memory
- L1 Cache
  - Hardware caching enabled
    - The HW decides what goes in or out and when
- Shared memory
  - Software manages what goes in/out
  - Allows more complex access patterns to be cached

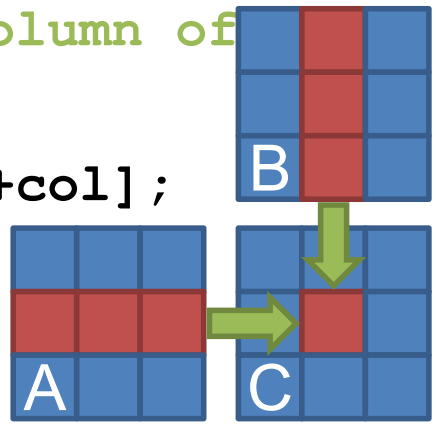
Example: Matrix multiplication

- C = A * B
  - C(i,j) = sum(dot(row(A,i),col(B,j)))
- Parallelization strategy
  - Each thread computes one C element
  - 2D kernel



Matrix multiplication implementation

```
__global__ void mat_mul(float *a, float *b,
                        float *c, int width)
{
  // calc row & column index of output element
  int row = blockIdx.y*blockDim.y + threadIdx.y;
  int col = blockIdx.x*blockDim.x + threadIdx.x;

 float result = 0;

  // do dot product between row of a and column of
  for(int k = 0; k < width; k++) {
      result += a[row*width+k] * b[k*width+col];
 }
  c[row*width+col] = result;
}
```

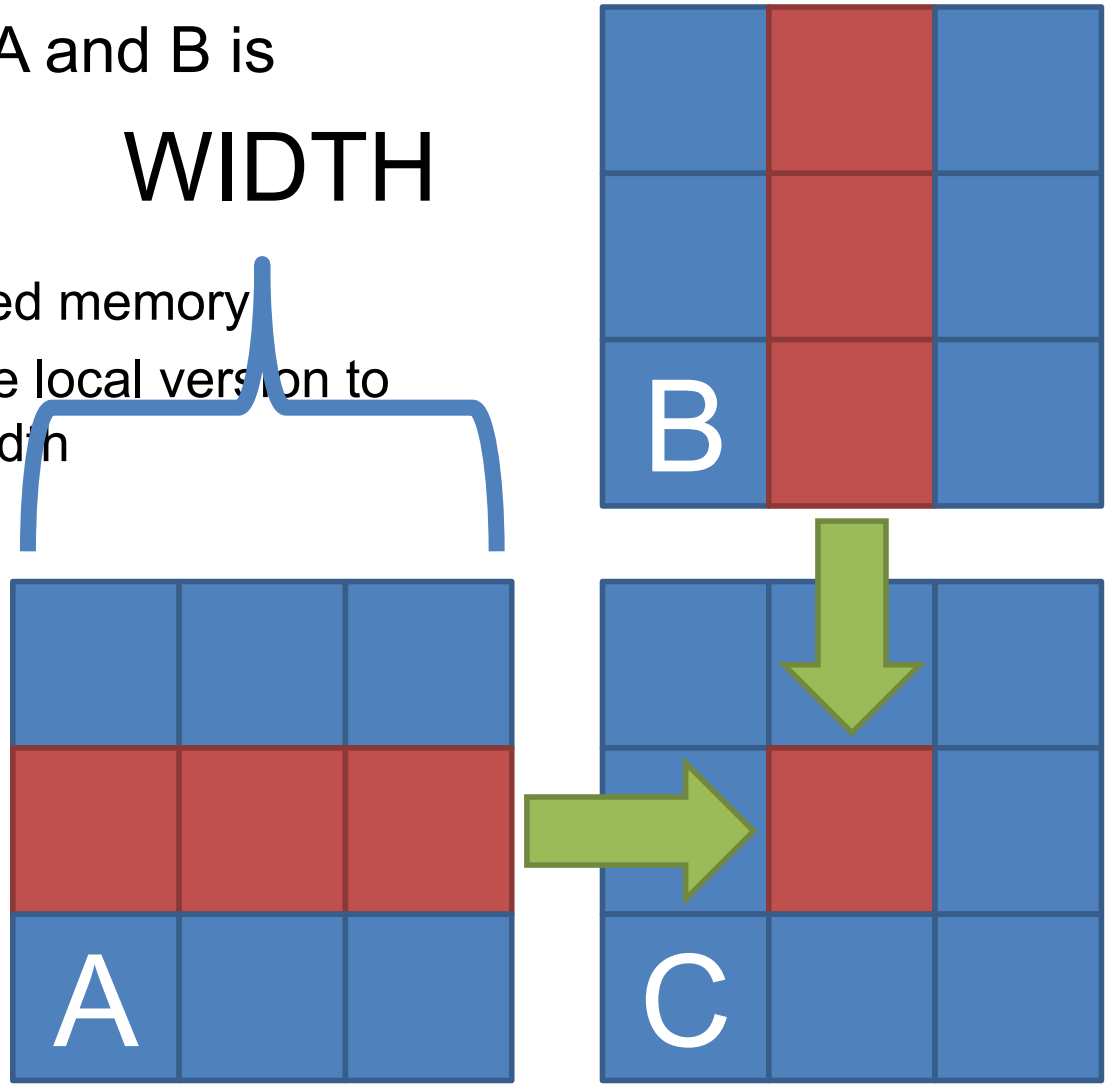


Matrix multiplication performance

Loads per dot product term	2 (a and b) = 8 bytes
FLOPS	2 (multiply and add)
AI	2 / 8 = 0.25
Performance GTX 580	1581 GFLOPs
Memory bandwidth GTX 580	192 GB/s
Attainable performance	192 * 0.25 = 48 GFLOPS
Maximum efficiency	3.0 % of theoretical peak

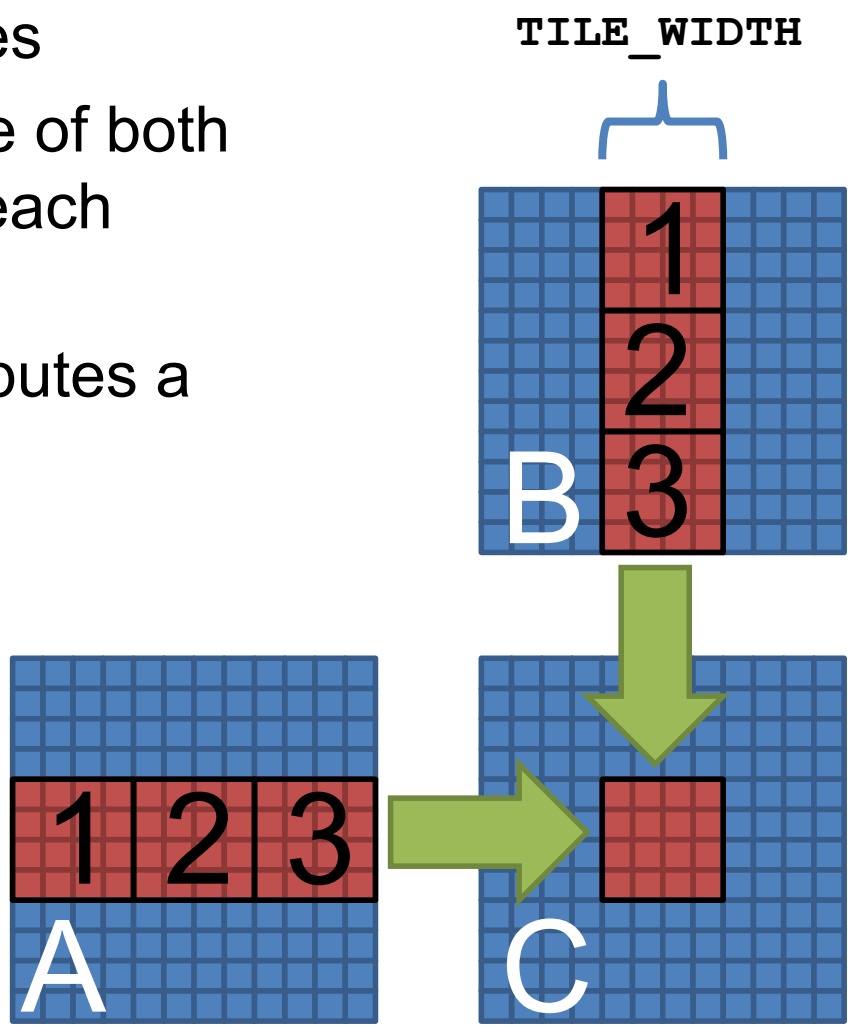
Data reuse

- Each input element in A and B is read WIDTH times
- IDEA:
  - Load elements into shared memory
  - Have several threads use local version to improve memory bandwidth



Using shared memory

- Partition kernel loop into phases
- In each thread block, load a tile of both matrices into shared memory each phase
- Each phase, each thread computes a partial result

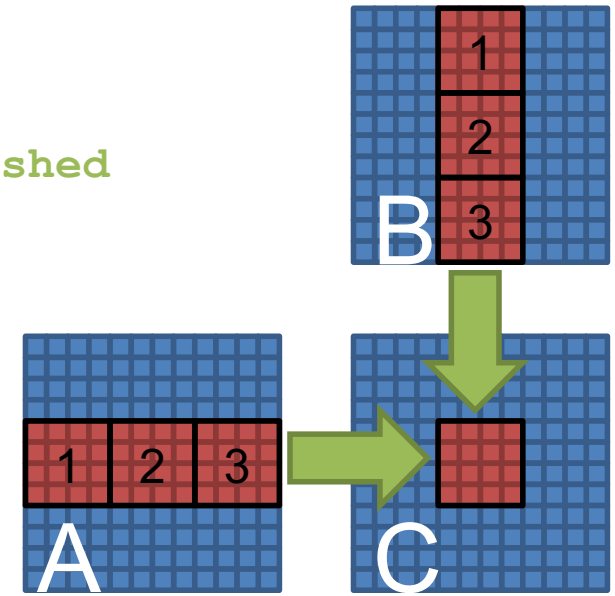


Matrix multiply with shared memory

```
__global__ void mat_mul(float *a, float *b,
                        float *c, int width) {
  // shorthand
  int tx = threadIdx.x, ty = threadIdx.y;
  int bx = blockIdx.x,  by = blockIdx.y;

  // allocate tiles in shared memory
  __shared__ float s_a[TILE_WIDTH][TILE_WIDTH];
  __shared__ float s_b[TILE_WIDTH][TILE_WIDTH];

  // calculate the row & column index from A,B
  int row = by*blockDim.y + ty;
  int col = bx*blockDim.x + tx;

 float result = 0;
```

Matrix multiply with shared memory

```
  // loop over input tiles in phases, p = crt. phase
  for(int p = 0; p < width/TILE_WIDTH; p++) {
    // collaboratively load tiles into shared memory
    s_a[ty][tx] = a[row*width + (p*TILE_WIDTH + tx)];
    s_b[ty][tx] = b[(p*TILE_WIDTH + ty)*width + col];
// barrier: ALL writes to shared memory finished
    __syncthreads();

    // dot product between row of s_a and col of s_b
    for(int k = 0; k < TILE_WIDTH; k++) {
      result += s_a[ty][k] * s_b[k][tx];
    }
// barrier: ALL reads of shared memory finished
    __syncthreads();
  }

  c[row*width+col] = result;
}
```



Use of Barriers in mat_mul

- Two barriers per phase:
  - __syncthreads after all data is loaded into shared memory
  - __syncthreads after all data is read from shared memory
    - Second __syncthreads in phase p guards the load in phase p+1

- Formally, __synchthreads is a barrier for shared memory for a block of threads:

"void __syncthreads();

waits until all threads in the thread block have reached this point **and** all global and shared memory accesses made by these threads prior to __syncthreads() are visible to all threads in the block."

Matrix multiplication performance

	Original	shared memory
Global loads	$2N^3$ * 4 bytes	($2N^3$ / TILE_WIDTH) * 4 bytes
Total ops	$2N^3$	$2N^3$
AI	0.25	0.25 * TILE_WIDTH

Performance GTX 580	1581 GFLOPs
Memory bandwidth GTX 580	192 GB/s
AI needed for peak	1581 / 192 = **8.23**
TILE_WIDTH required to achieve peak	0.25 * TILE_WIDTH = **8.23**, TILE_WIDTH = 32.9

CUDA: MANY OTHER FEATURES

Other interesting topics

- Unified memory
- Tensor cores
- Independent thread scheduling
- nvLink
- HBM2
- Shared GPU for multiple “jobs”
- Task graphs
- Libraries and tools
  - Nsight systems and/or nvprof
  - Auto-tuning
- Other programming models
  - SyCL (follow-up from OpenCL)
  - OpenMP / OpenACC
- Other GPUs
  - Intel (OneAPI / DPC++), AMD (Rocm, HIP), Arm

SUMMARY

Take home message

- GPUs are massively parallel architectures with limited flexibility, but very high throughput
- Pro's:
  - Much higher compute capabilities
  - Higher bandwidth
- Con's
  - Limited on-card memory
  - Low-bandwidth communication with host
- Debate-able
  - Programmability & productivity

Open research questions

- Shall we port all applications on GPUs?
  - If yes – can we automate the process?
  - If not – can we decide how to select?
- Shall we use GPUs in large-scale systems?
- Shall we use heterogeneous CPU+GPU systems?
- Can we improve the GPU design …
  - For HPC?
  - For other application domains?

Questions? Comments? Suggestions?

- A.L.Varbanescu@utwente.nl

… also if you want to work on GPU-related projects OR in a team that works on heterogeneous computing.

… All you have to do is ask ☺

Back-up slides