PRAGMA-BASED GPU COMPUTING

Introduction to GPUs

Introduction to OpenACC

Ana Lucia Varbanescu (UvA)
A.L.Varbanescu@uva.nl

Why GPUs?

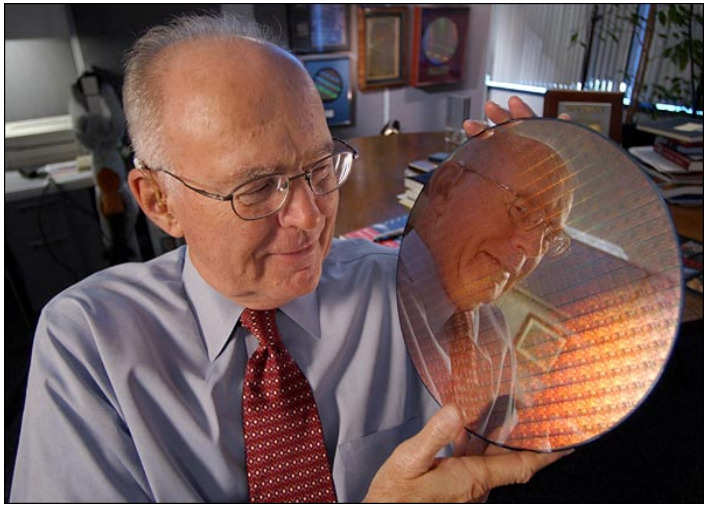
- Promise of performance beyond most other architectures
  - CPUs
  - Multi-core CPUs
  - FPGAs
  - …
- They are power efficient
  - 2-5x better than a CPU
- They are **already paid for**!!!
  - GPGPUs are gaming platforms which we (ab)use for HPC

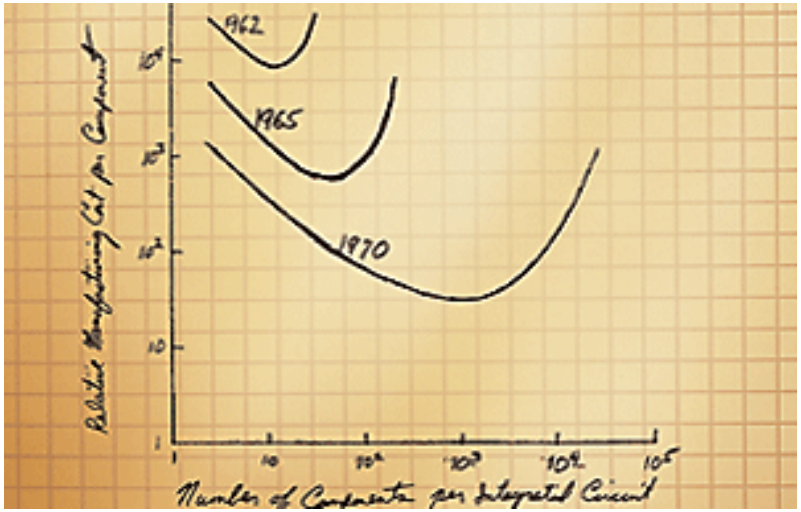
… it's just that programming them is not (yet) a walk in the park…

MULTI-/MANY-CORES

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.

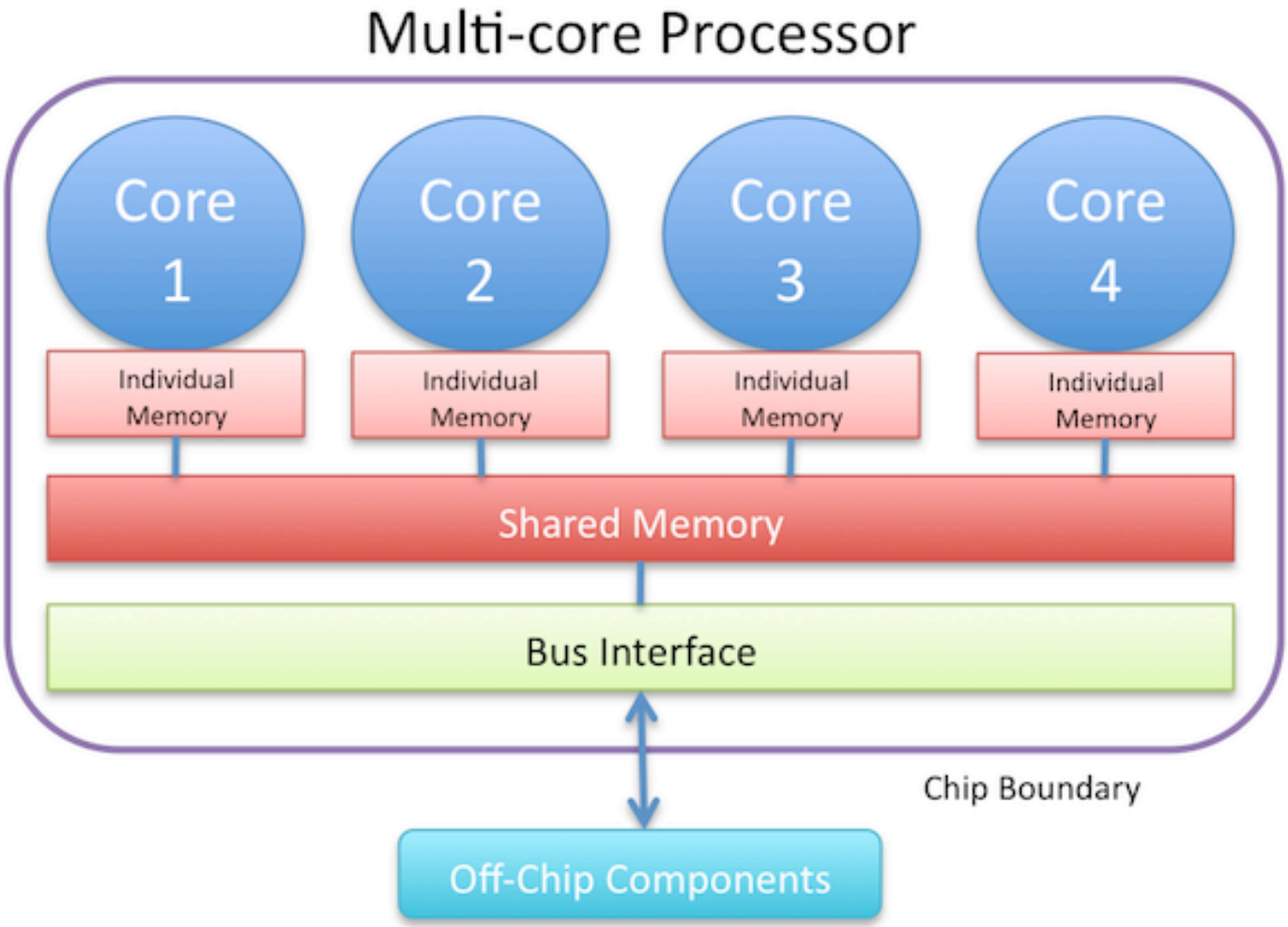
Not anymore!
We no longer gain performance by “growing” sequential processors …

New ways to use transistors

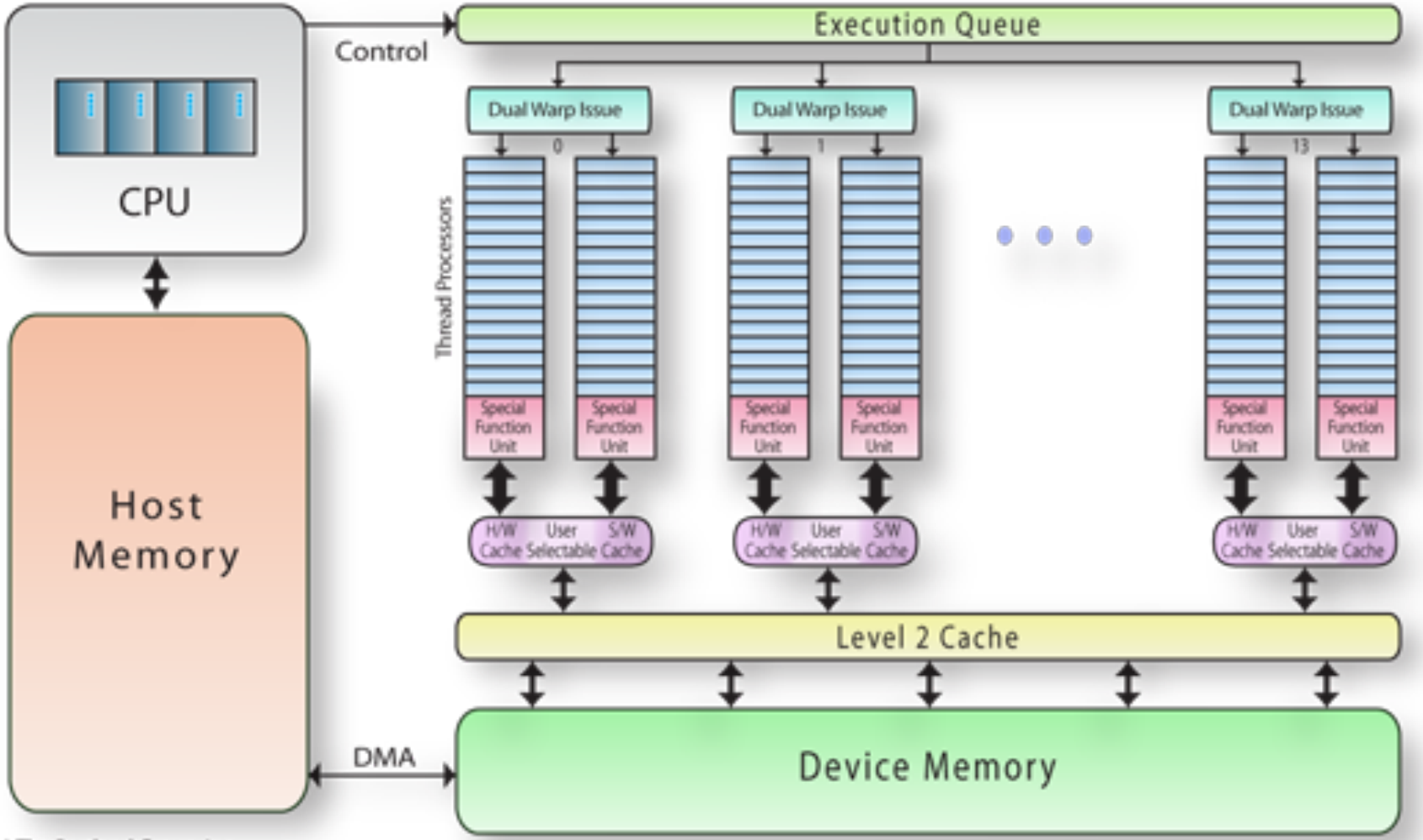
Improve PERFORMANCE by using parallelism on-chip: multi-core (CPUs) and many-core processors (GPUs).

Multiple cores: CPU vs. GPU

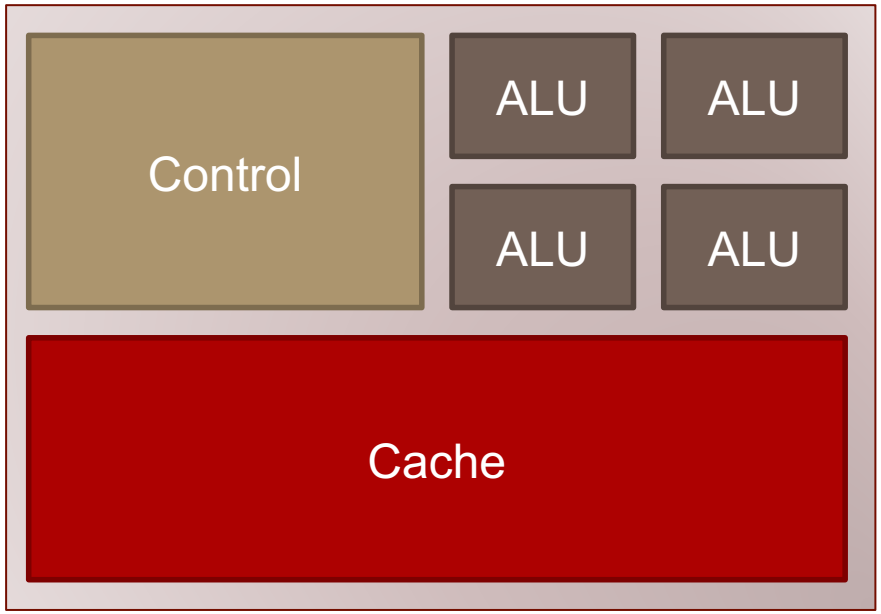
Generic multi-core: a CPU



Generic many-core: a GPU



CPU vs. GPU

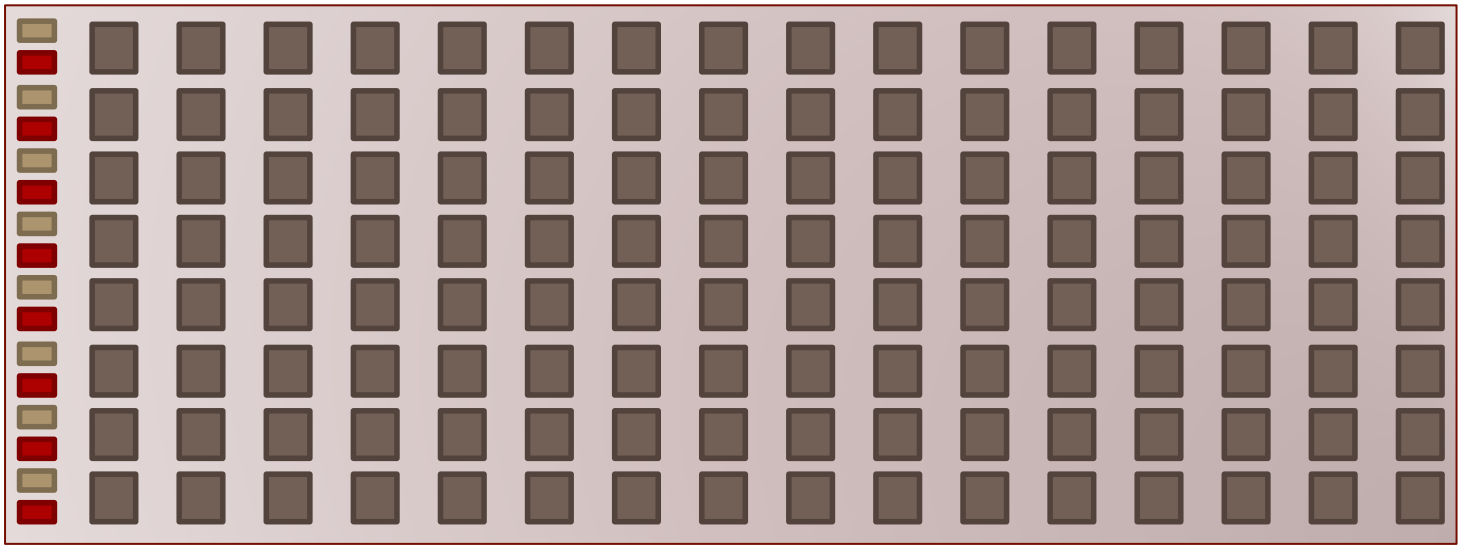


CPU

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

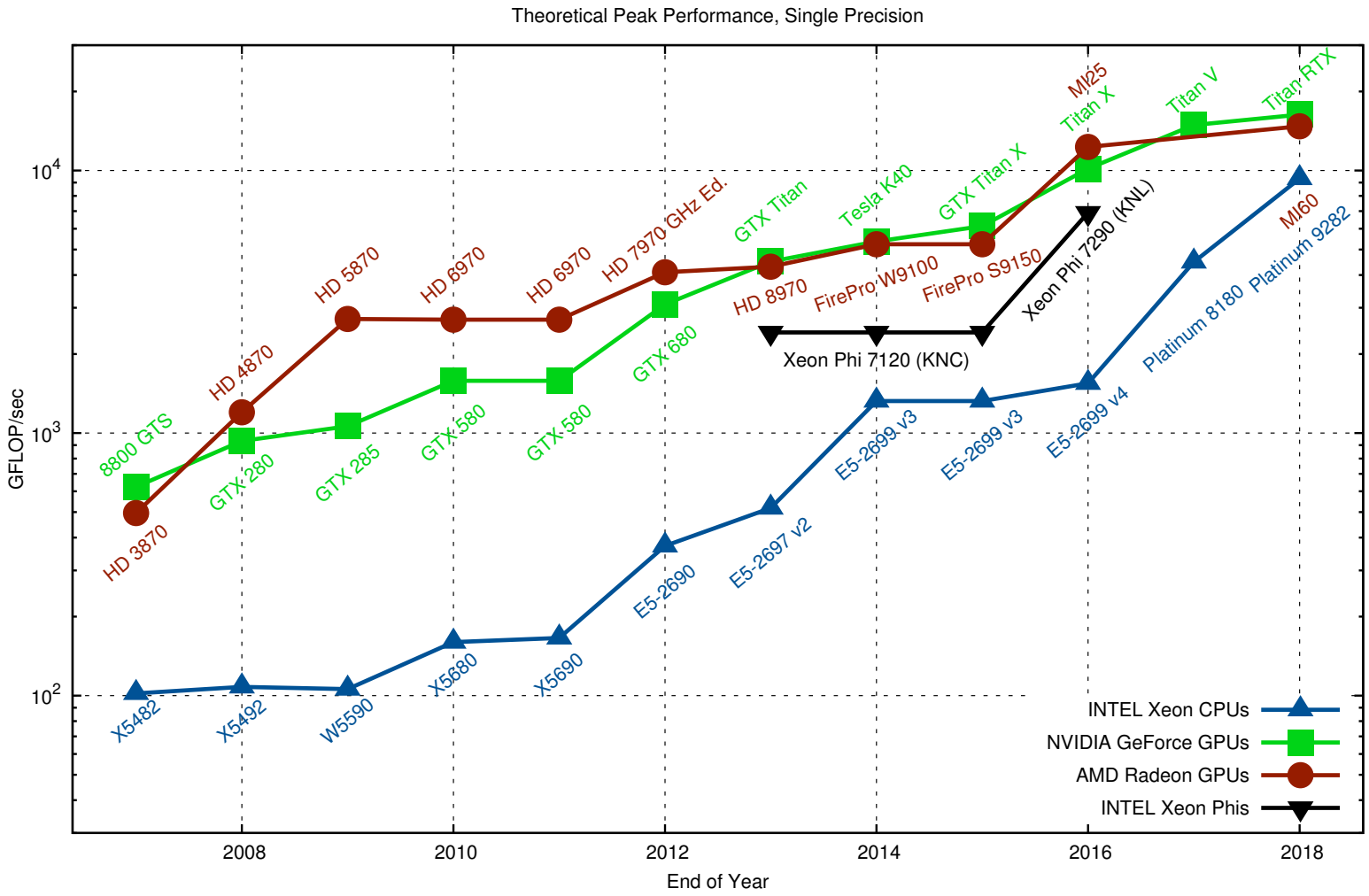
How about performance ?

Hardware Performance metrics

- Clock frequency [GHz] = absolute hardware speed
  - Memories, CPUs, interconnects
- **Operational speed [GFLOPs]**
  - Operations per second
  - **single** AND **double** precision
- **Memory bandwidth [GB/s]**
  - Memory operations per second
    - Can differ for read and write operations !
  - Differs a lot between different memories on chip
- Power [Watt]
  - The rate of consumption of energy
- Derived metrics
  - FLOP/Byte, FLOP/Watt

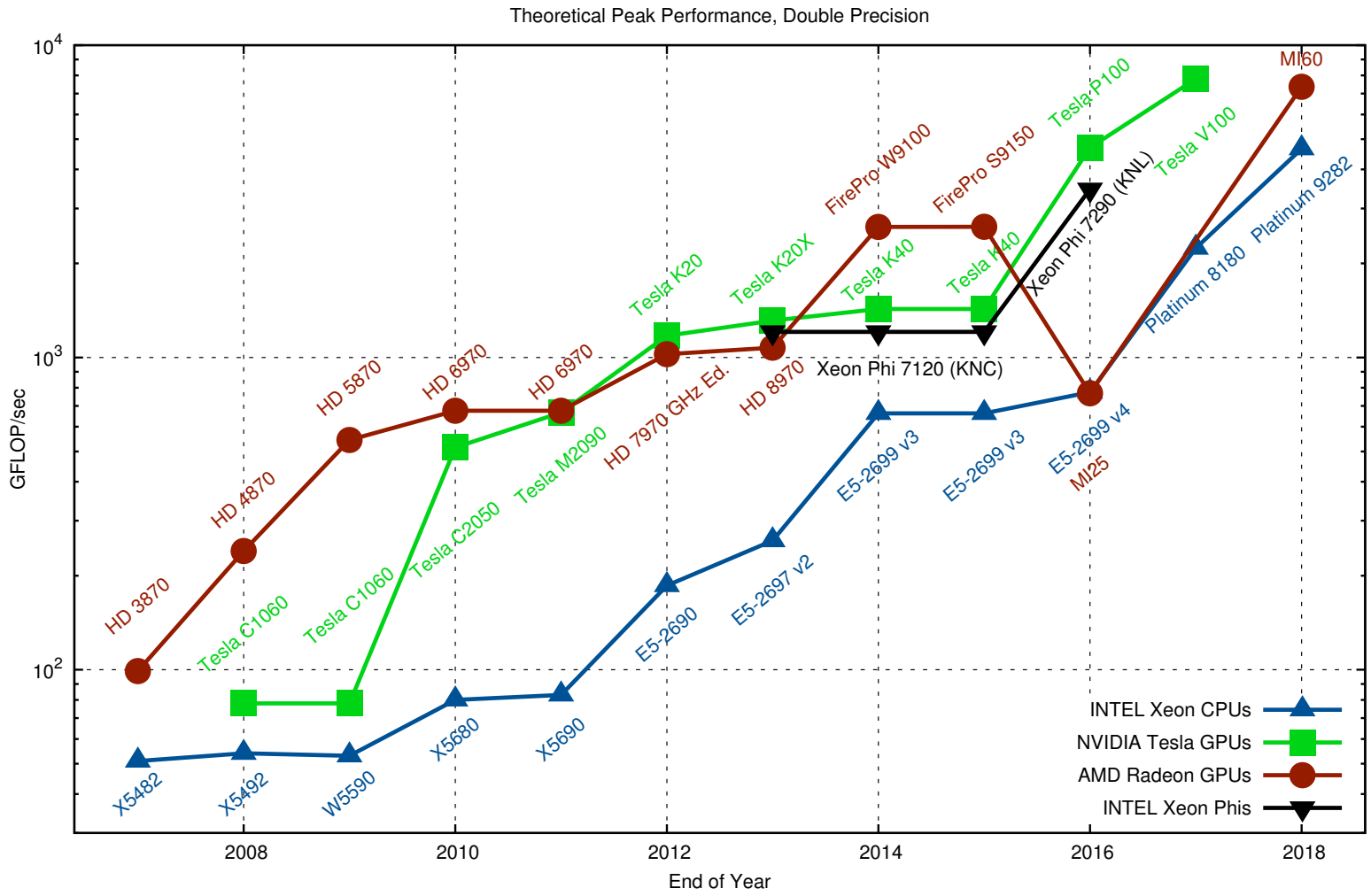
Name	FLOPS
yottaFLOPS	10^{24}
zettaFLOPS	10^{21}
exaFLOPS	10^{18}
petaFLOPS	10^{15}
teraFLOPS	10^{12}
gigaFLOPS	10^{9}
megaFLOPS	10^{6}
kiloFLOPS	10^{3}

multi vs *many* cores (SP-FLOPs)



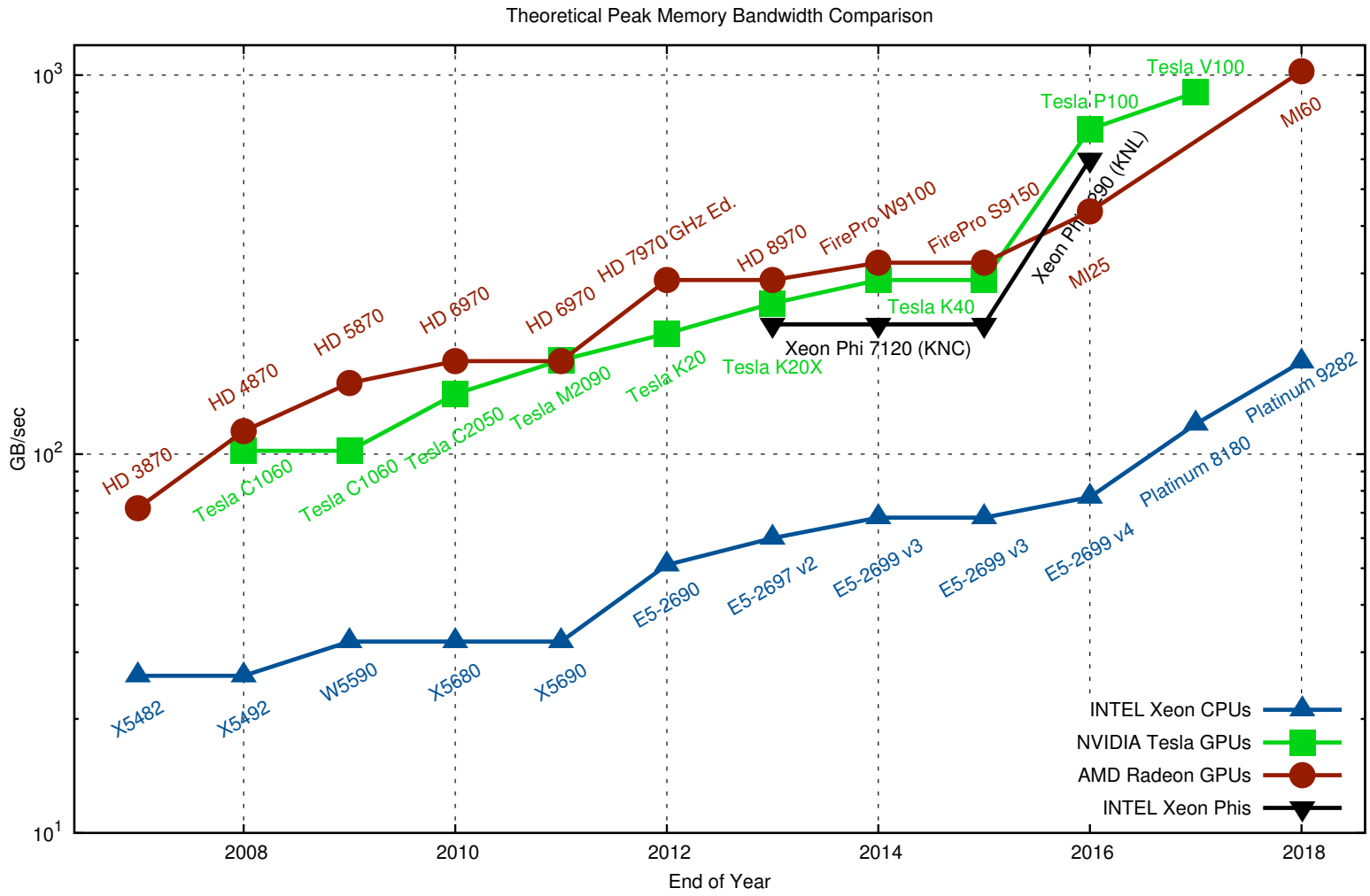
https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

multi vs *many* cores (DP-FLOPs)



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

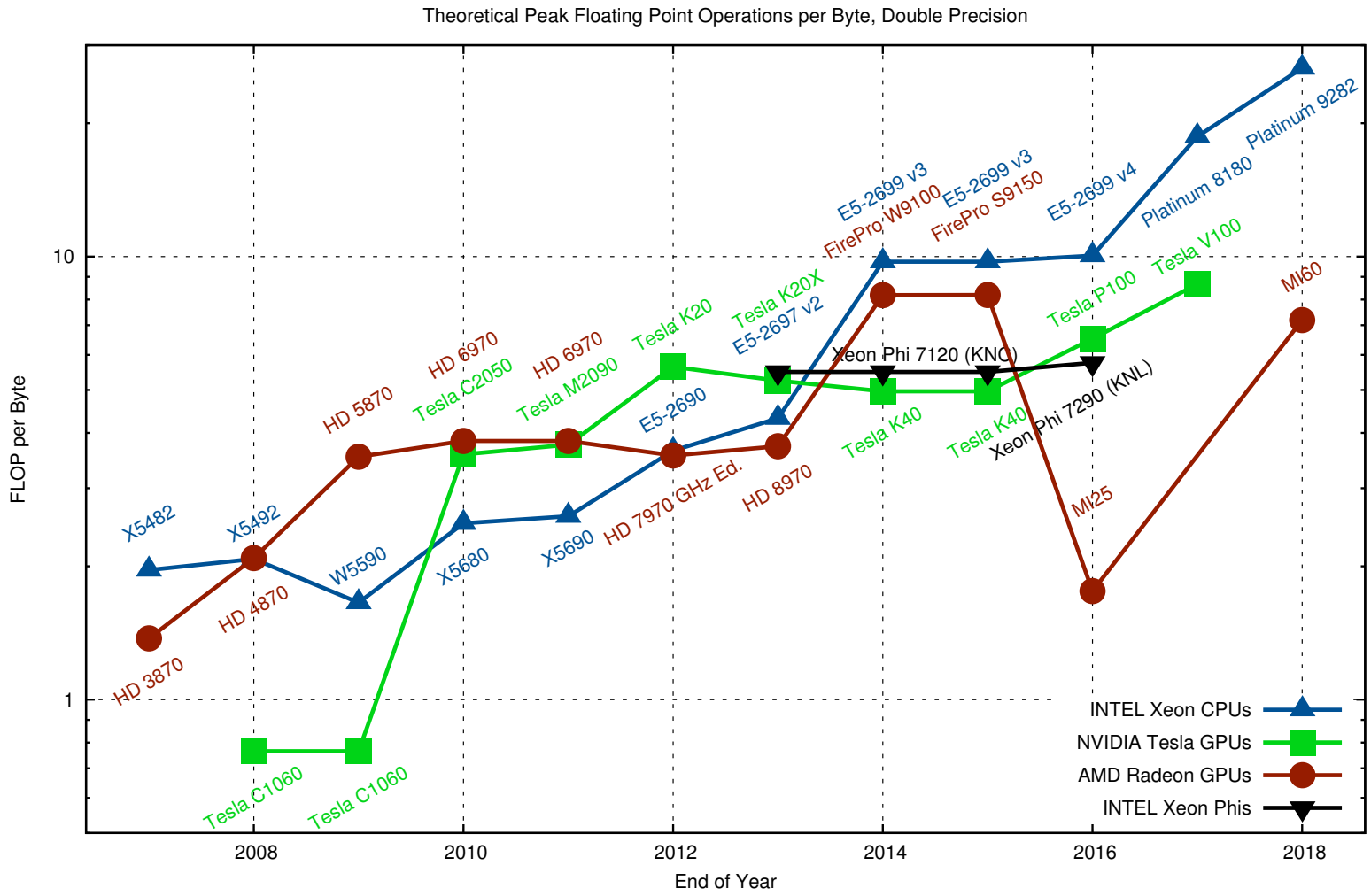
multi vs *many* cores (GB/s)



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

Balance ?

FLOPs/Byte (DP) !

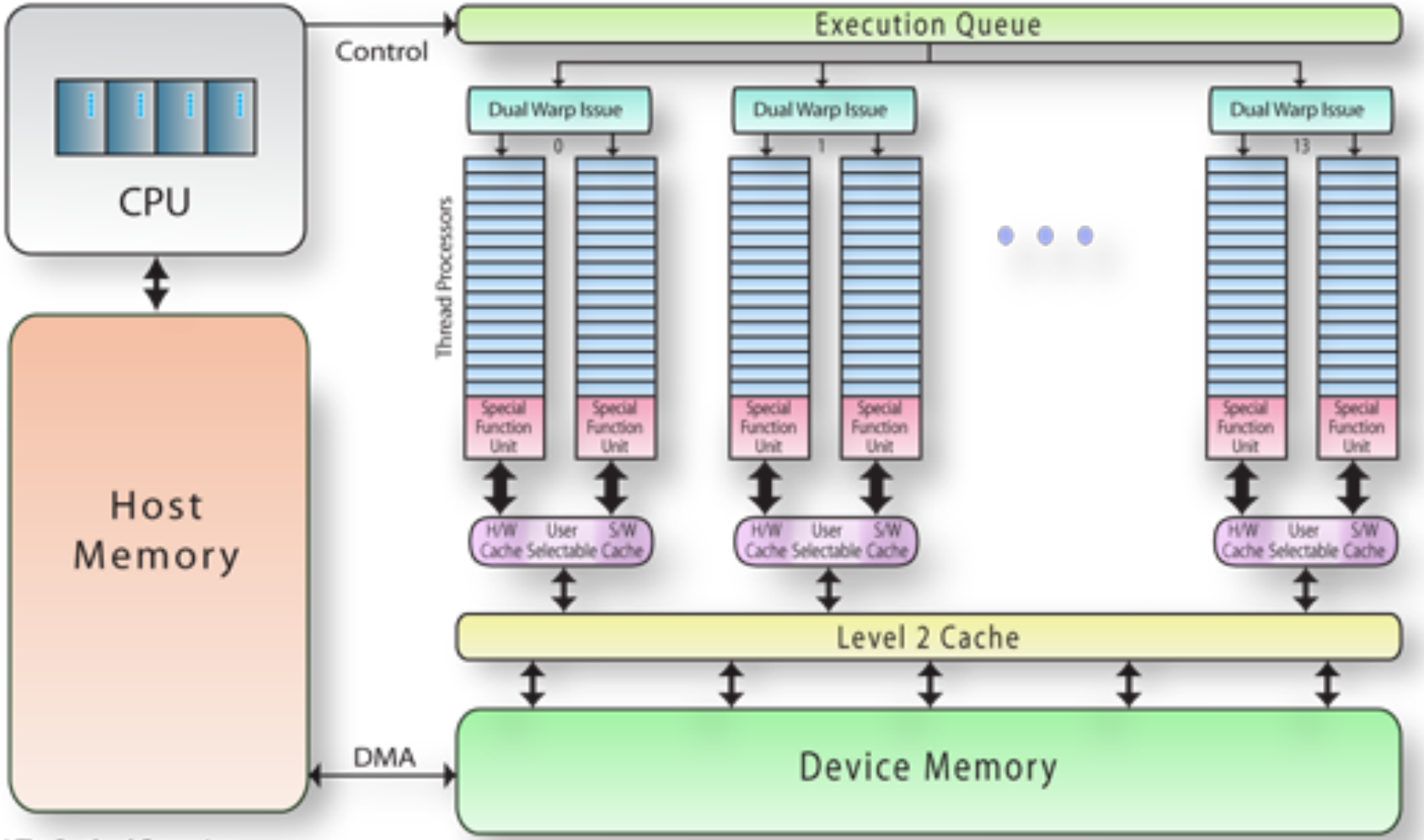


https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

GPU ARCHITECTURE

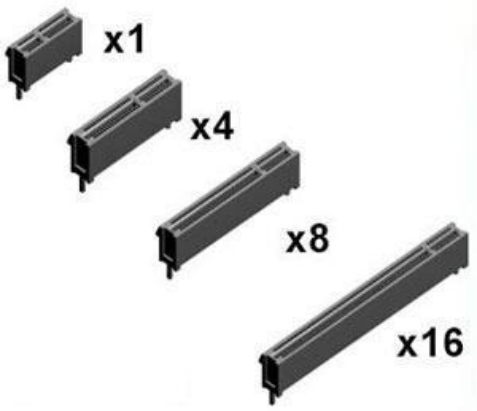
Massive parallelism => massive performance ???

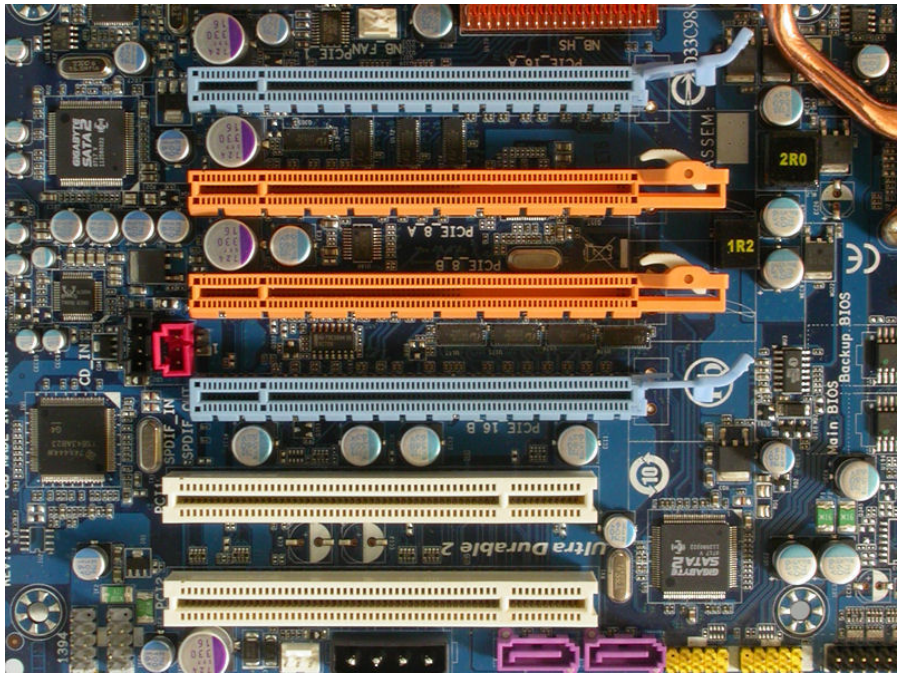
A GPU Architecture



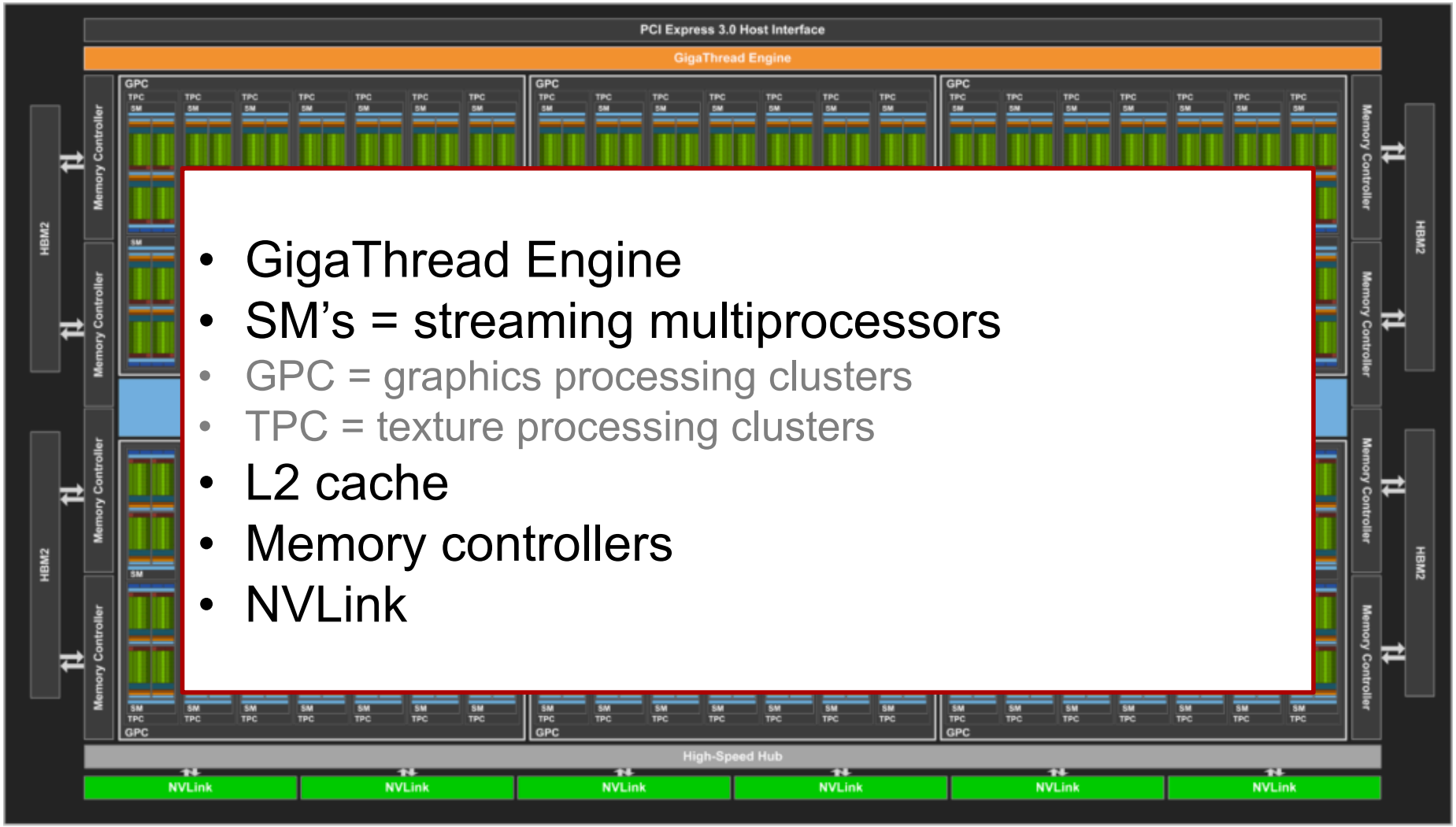
GPU Integration into the host system

- Typically PCI Express 3.0
- Theoretical speed 8 GB/s
  - Effective ≤ 6 GB/s
  - In reality: 4 – 6 GB/s





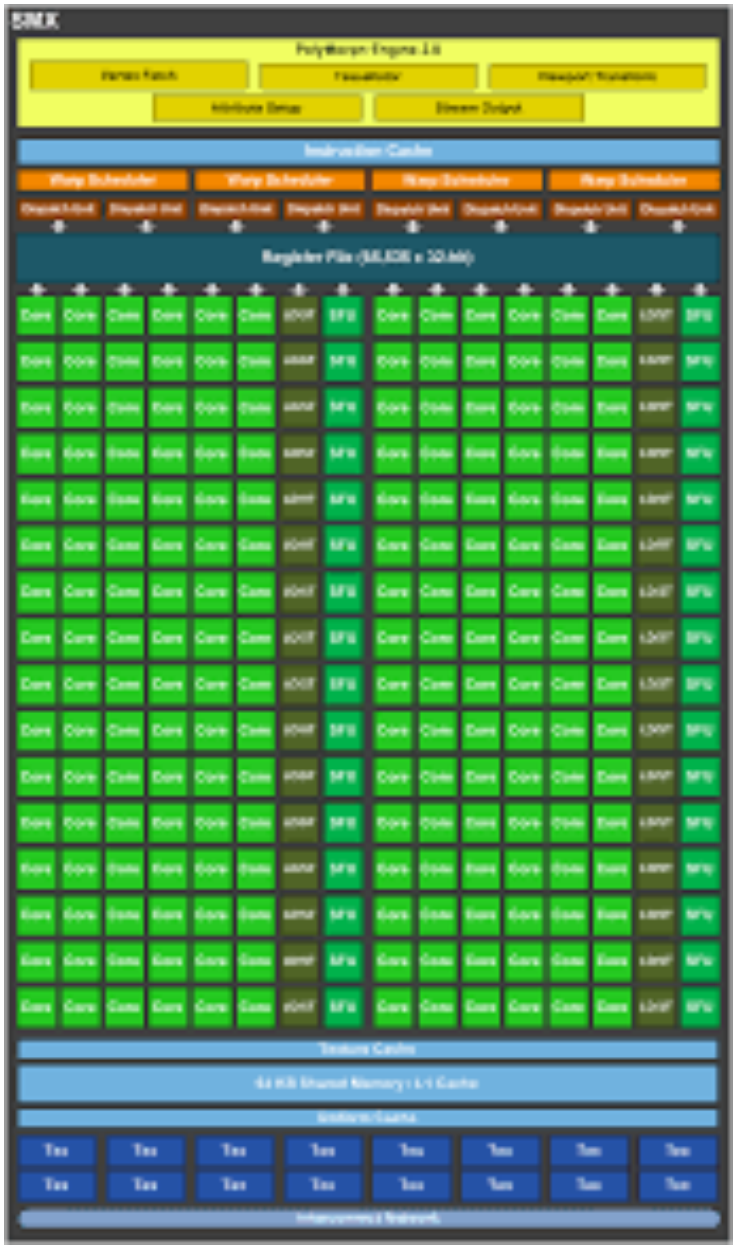
Inside an NVIDIA GPU architecture



- 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



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

DP100 (Pascal)

Inside an SM

- Volta:
  - Tensor cores (8 / SM)
  - Enhanced L1 cache
  - L0, L1 instruction cache

Tesla V100 (Volta)

NVIDIA GPUs (8+ years)

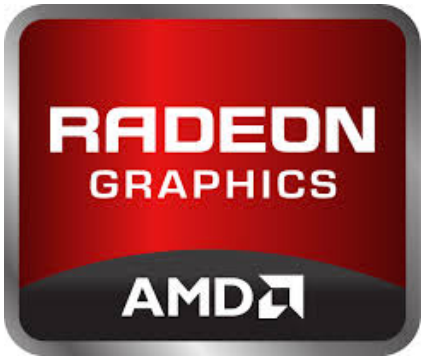
	Fermi	Kepler	Maxwell	Pascal	Volta
GPU	GTX480	GK180	GM200	GP100	GV100
Compute capability (CC)	2.x	3.5	5.2	6.0	7.0
SMs					80
TPC					40
FP3					64
FP6					32
Cloc					30
Peak [TFLOPs]					15.7
Peak FP64 [TFLOPs]	0.168	1.68	.21	5.3	7.8

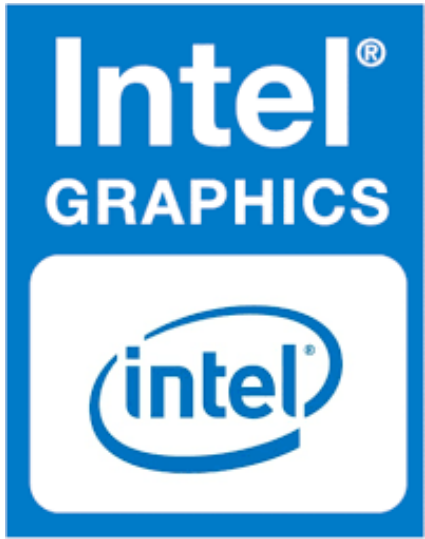
Tesla K40m uses GK110B, which has:

- 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



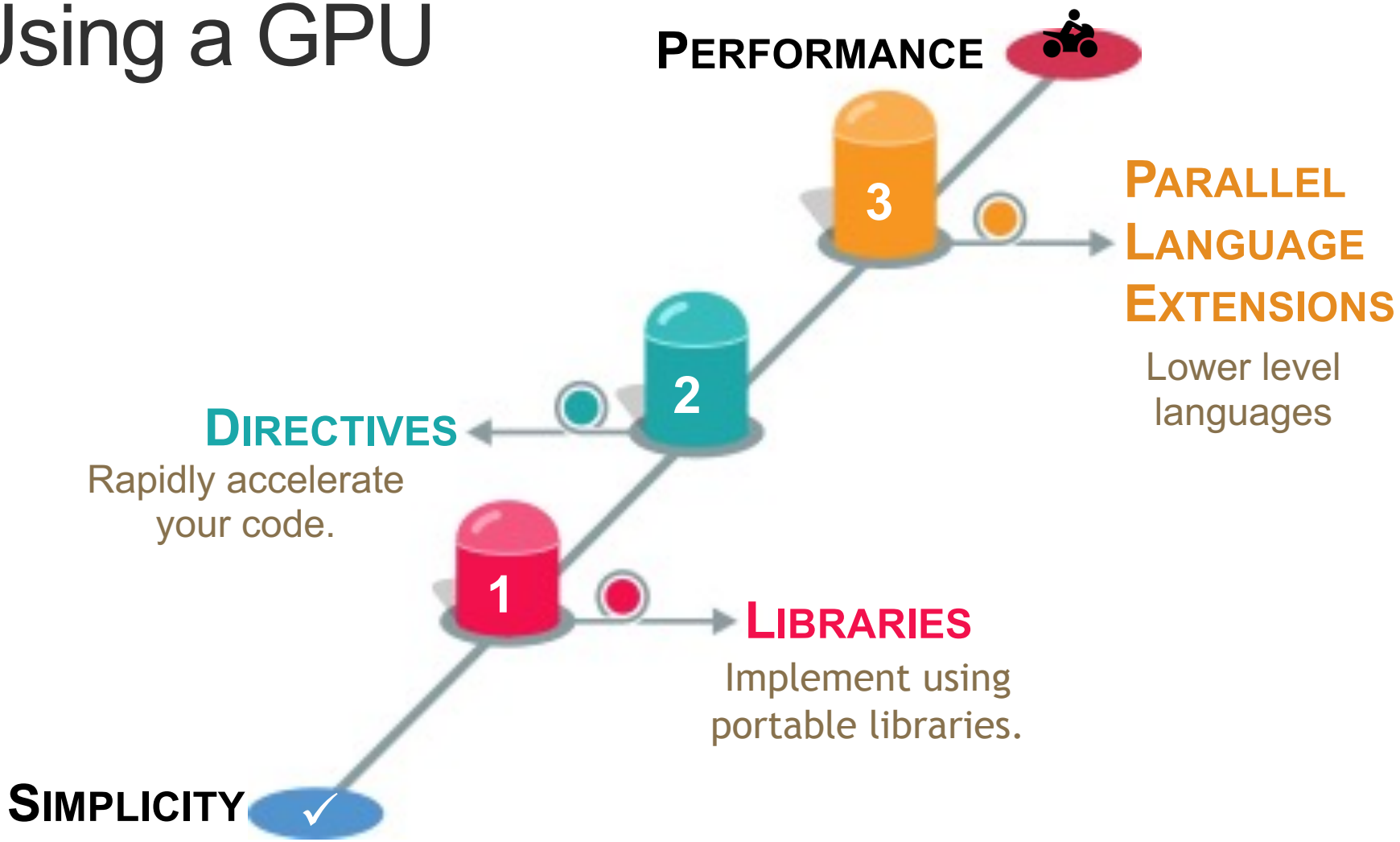


All GPUs …

- 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

Using a GPU



Slide courtesy of Ana da Cunha @ Surf

Using a GPU

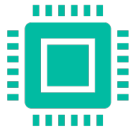
PORT APPLICATIONS

LIBRARIES

- ✓ Small code changes
- ✓ High performance
- ✓ Limited

DIRECTIVES

- ✓ Existing languages
- ✓ Simple
- ✓ Less performance control



LANGUAGES

- ✓ Max. Performance
- ✓ low-level
- ✓ Time consuming

Slide courtesy of Ana da Cunha @ Surf

The principles

GPU Parallelism

- Data parallelism (fine-grain)
- **SIMT** (Single Instruction **Multiple Thread**) execution
  - Many threads execute concurrently
    - Same instruction
    - Different data elements
    - HW automatically handles divergence
  - Not same as SIMD because of multiple register sets, addresses, and flow paths*
- Hardware multithreading
  - HW resource allocation & thread scheduling
    - Excess of threads to hide latency
    - Context switching is (basically) free

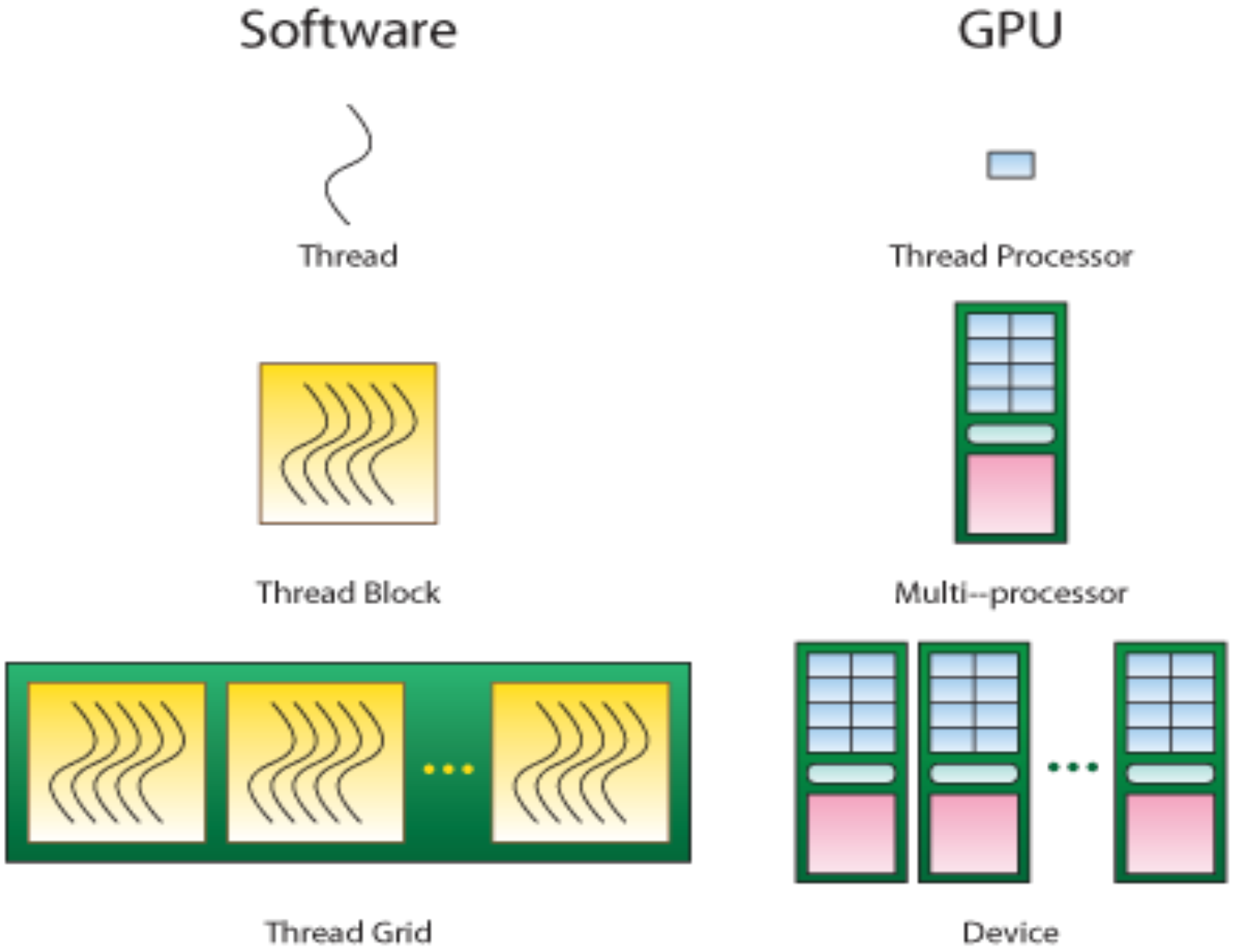
*http://yosefk.com/blog/simd-simt-smt-parallelism-in-nvidia-gpus.html

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

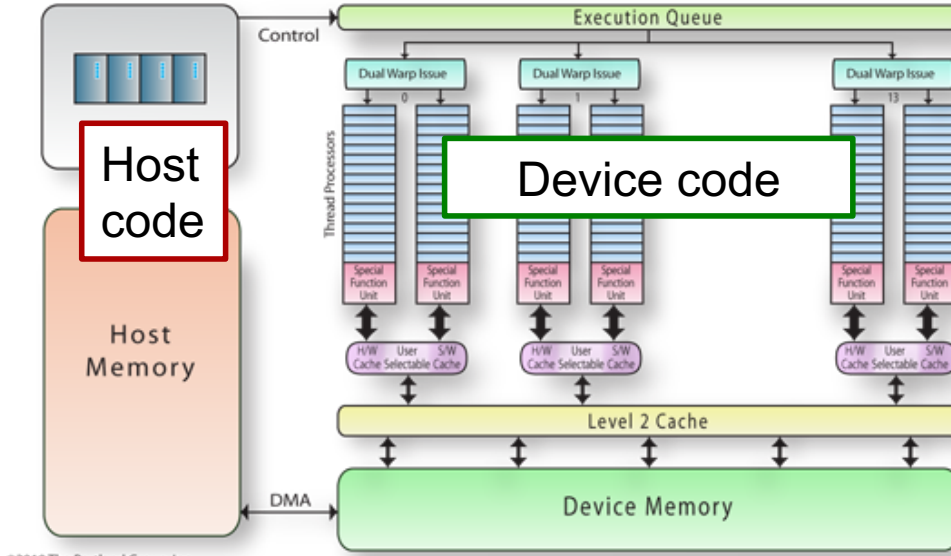
Model of Parallelism



The practicalities

GPU program organization

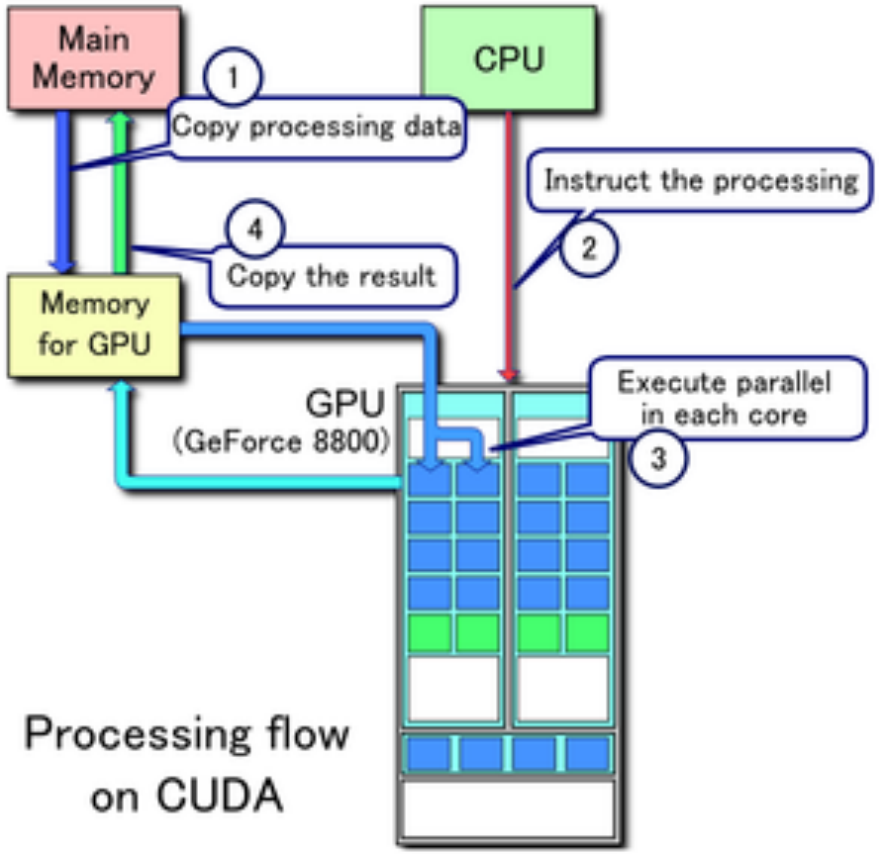
- Two types of code:
  - **Device code** = GPU code = kernel(s)
    - Sequential program
    - Write for 1 thread, execute for all
  - **Host code** = CPU code
    - Instantiate grid + run the kernel
    - Memory allocation, management, deallocation
    - C/C++/Java/Python/…
- Host-device communication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



Execution flow

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

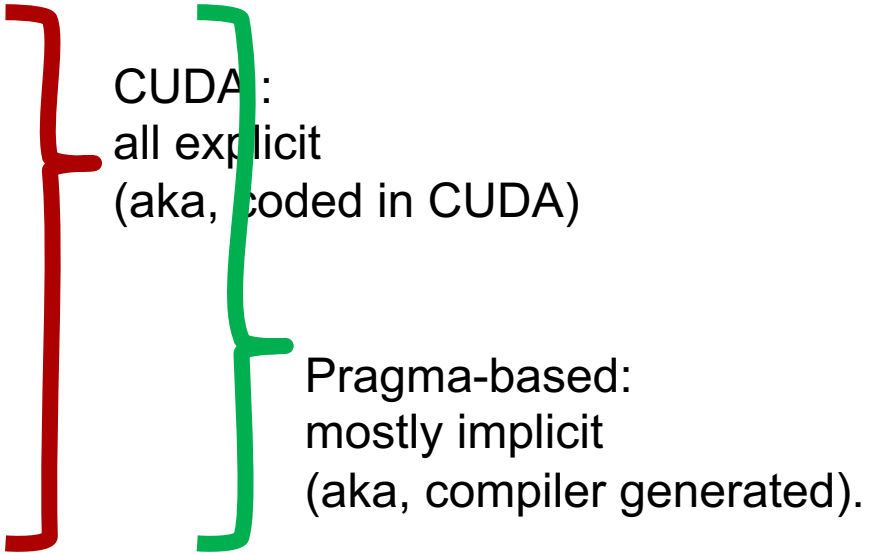
- GPU memory allocation
- Transfer data CPU→GPU
- CPU calls GPU kernel
- GPU kernel executes
- Transfer data GPU→CPU
- GPU memory release
- [Repeat]



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]



EXAMPLE: VECTOR-ADD

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];
    }
}
```

Vector add : CUDA : Kernel

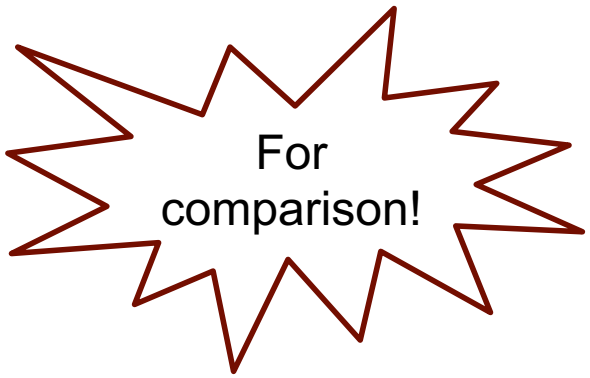
```
// compute vector sum c = a + b
// each thread performs one pair-wise addition
__global__ void vector_add(float* A, float* B, float* C) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

Vector add: OpenACC

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

Vector add : CUDA : Host [1/3]

```
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);
```

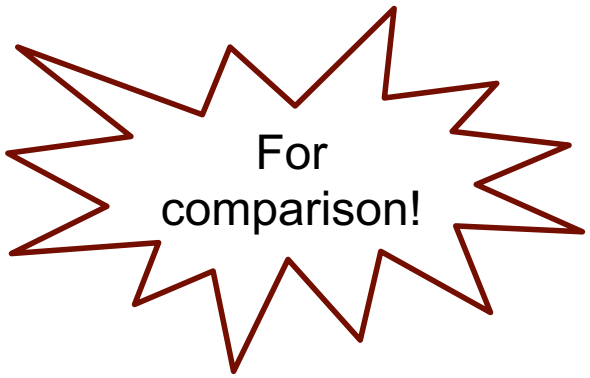


Vector add : CUDA : Host [2/3]

```
 // 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);
}
```

Vector add : CUDA : Host [3/3]

```
// free device memory
 cudaFree(deviceA);
 cudaFree(deviceB);
 cudaFree(deviceC);

// free host memory
 free(A);
 free(B);
 free(C);

}
```



The very practical practicalities

What we need for OpenACC? [1]

- A compiler with OpenACC supoprt and offloading support
  - gcc : https://www.openacc.org/blog/evaluating-performance-openacc-gcc
  - Clang, via Clacc and OpenMP: https://csmd.ornl.gov/project/clacc
  - PGI Compiler: now NVHPC
- Sequential code
  - …that we want to parallelize

What we need for OpenACC? [2]

- **Increasing knowledge** about how to communicate with the compiler
  - In the code: pragma's
  - During compilation: flags

Hands on

- On Cartesius:
  - Login ☺
  - Load the necessary modules

    module load 2020

    module load NVHPC/20.7

  - Get the vector-add code
  - Compile and test how it works (demo)
  - We will try different options for parallelization!

To do – hands-on #1 [1]

- Get the vector-add application
  - Sequential C

- Use the **`pragma acc kernels`**

- Compilation example:

```
pgcc -acc -ta=<fill in> -Minfo=accel -O3 -
mcmodel=medium vecAdd.c -o vecAdd_tesla
```

- Try different compilation options for “-ta” :
  - Target: **host** => sequential execution
  - Target: **multicore** => parallel execution on the multi-core CPU host
  - Target: **tesla:cc35** => offloading to the GPU of compute capability 3.5

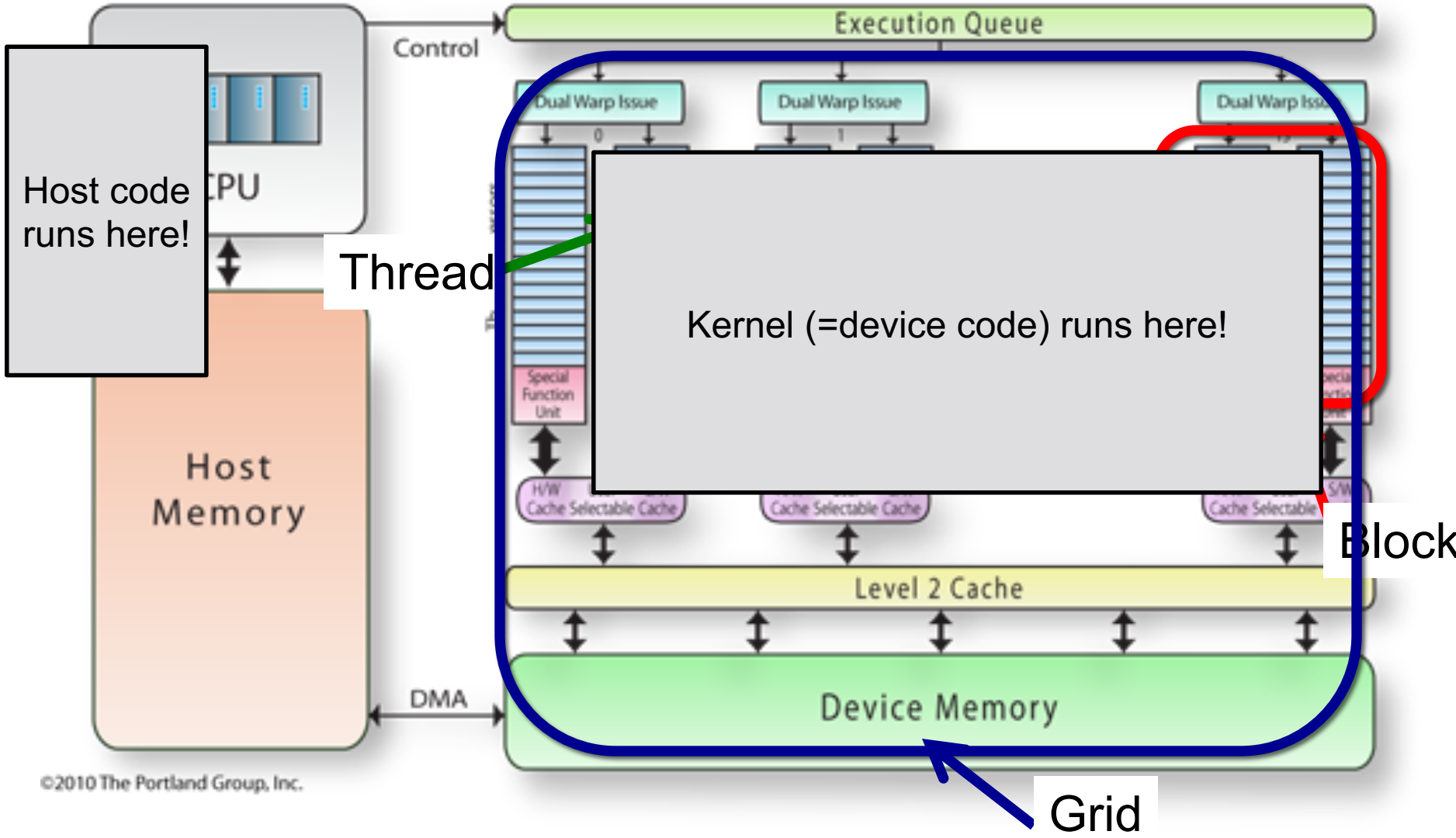
To do – hands-on #1 [2]

- Check what **`nvprof`** reports for the execution on the GPU and CPU versions

- Measure performance
  - Using the nvidia profiler
  - Using regular timing-checkpoints
  - Compare the results for the three versions
    - Host (sequential)
    - Multicore (CPU, parallel)
    - Tesla (GPU, offloading)

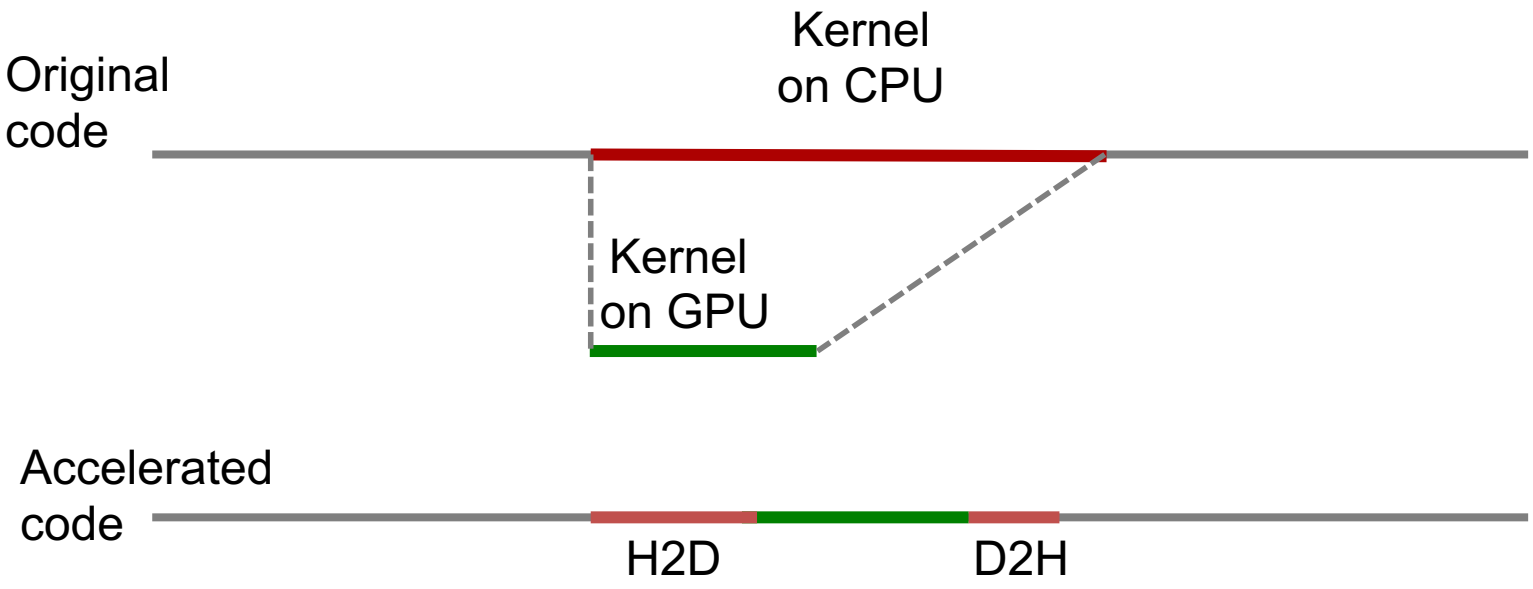
PERFORMANCE

Architecture



How to reason about performance?

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



Speed-up?

Three relevant speed-ups

- Kernel - only
  - S_K = T (kernel_on_CPU) / T (kernel_on_GPU)
- “Computation task”
  - S_C = T (kernel_on_CPU) / T (kernel_on_GPU + H2D + D2H)
    - D2H = device to host data transfer
    - H2D = host to device data transfer
- Application level
  - S_A = T (full application) / T (application_with_offloading)
- Expectation : $S_K >> S_C > S_A$

PRAGMA-BASED GPU PROGRAMMING

OpenACC

Why OpenACC?

- CUDA:
  - Restricted to NVidia GPUs
  - Very low-level, thus unproductive
- OpenCL:
  - Open Compute Language
  - Pure library solution
  - Functionally portable to various GPUs and CPUs
  - As low-level as CUDA
  - Performance often not portable, too low-level
- Problem:
  - Accelerators promise high performance at low cost
  - But programming accelerators is
    - tedious, error-prone, unproductive

Why OpenACC?

- OpenACC:
  - Directives in the style of OpenMP
  - Compiler support
  - Let programmer decide about opportunities
  - Let programmers define the grand lines of the application
  - Let compiler do the leg work
  - **Increase programming productivity !!**

- Supported by: consortium of academic and industrial partners
  - More at www.openacc.org

In a nutshell

- OpenACC as a programming interface:
  - Compiler directives
  - Library functions
  - Environment variables

- An example:

```
#pragma acc name [clause]*
 structured block
```

The minimalistic approach

- The **`pragma acc kernels`**
  - *descriptive* directive.
    - Informs the compiler that there is candidate code for parallelizing/offloading
    - The compiler chooses how to act

- Compiler steps:
  - Analyze the code to try to identify parallelism
  - If found, identify which data must be transferred and when
  - Create a kernel
  - Offload the kernel to the GPU

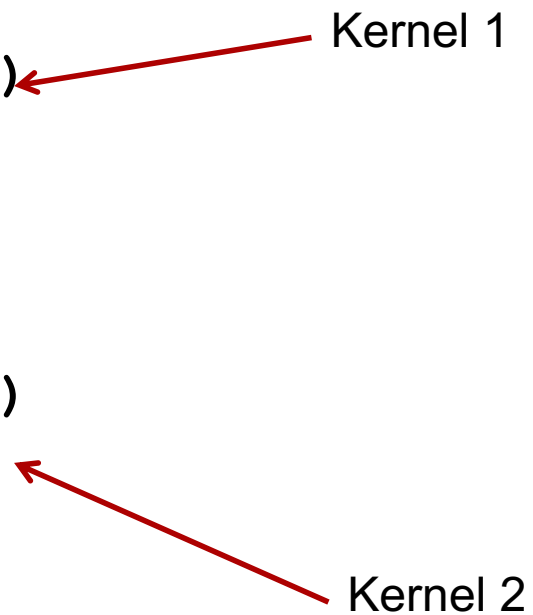
Slide inspired by https://docs.computecanada.ca/wiki/OpenACC_Tutorial_-_Adding_directives#Offloading_to_a_GPU

Leave it all to the compiler

#pragma acc kernels

- Example

```
#pragma acc kernels
{
  for (int i=0; i<N; i++)
  {
    x[i] = 1.0;
    y[i] = 2.0;
  }
  for (int i=0; i<N; i++)
  {
    z[i] = x[i] + y[i];
  }
}
```



Leave it *almost* all to the compiler

`#pragma acc kernels`

- Example

```
#pragma acc kernels
{
  for (int i=0; i<N; i++)
  {
    x[i] = 1.0;
    y[i] = 2.0;
  }
  #pragma acc loop independent
  for (int i=0; i<N; i++)
  {
    z[i] = x[i] + y[i];
  }
}
```

Kernel 1

Overrides compiler asessment => we specifically for this to be a kernel.

Kernel 2

The rest of the OpenACC Directives

- Kernel offloading:
  - Mark **loop nests** for accelerator execution
  - Control **hierarchical** concurrency
    - streaming multiprocessors
    - Cores
- Data management:
  - Move data from host to device
  - Move data from device to host
- Advanced stuff:
  - Multiple devices
  - Orphaned constructs
  - Asynchronous host execution
  - ...

Creating kernels

Loops => kernels

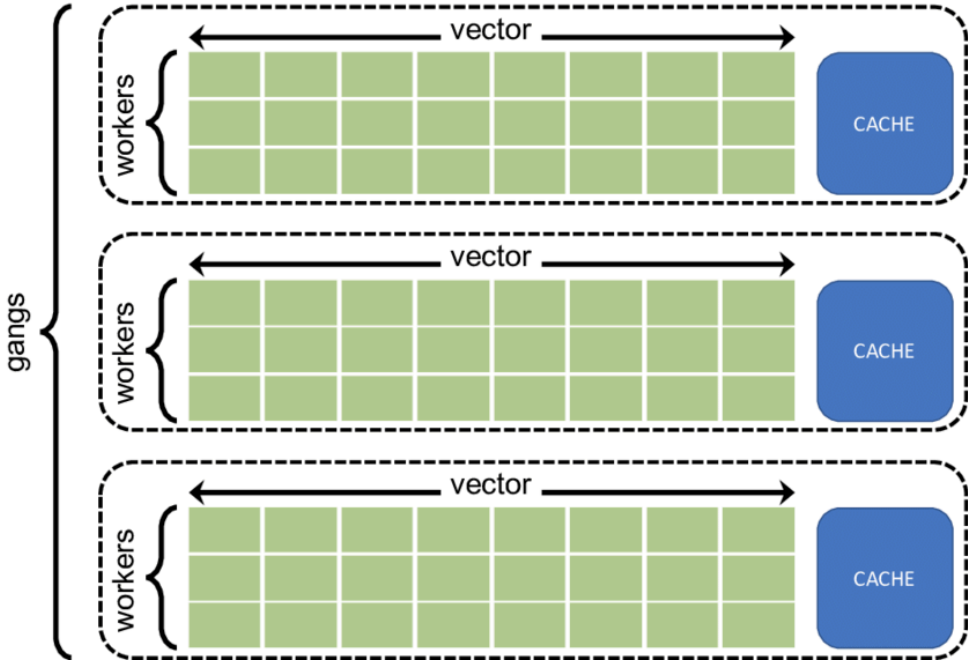
- Use **#pragma acc parallel loop**
  - Rules/management of scope of data must be explicit
  - Additional clauses (e.g., reduction) can be added
  - More control over levels of parallelism

```
#pragma acc parallel loop
  for(int i=0;i<num_rows;i++) {
    double sum=0;
    int row_start, row_end …
#pragma acc loop reduction(+:sum)
    for(int j=row_start;j<row_end;j++)
      sum += …;
    result[i]=sum;
  }
```

```
Compiler output
#pragma acc loop gang /* blockIdx.x */
#pragma acc loop vector(128) /* threadIdx.x */
Sum reduction generated for sum
```

Three levels of parallelism

- **vector** – threads operation
  - threads perform the same operation(s) on multiple data (SIMD) in a lockstep.
- **worker**
  - Computes one or more vectors
- **gang** – block operation
  - One or multiple workers.
  - Multiple gangs run completely independently.



Work sharing examples

- Hierarchical concurrency
  - Gang-redundant code: executed by all gangs
  - Gang-partitioned code: partitioned across gangs
  - Worker-single: single worker per gang is active
  - Worker-partitioned: partitioned across the workers in a gang
  - Vector-single: no vectorization per worker
  - Vector-paritioned: vectorization per worker

- For partitioning a loop
  - Same restrictions to C for-loops apply as in OpenMP.

Work sharing [1] – “gangs”

```
# pragma acc parallel
{
  {code1.1;} // gang - redundant
  # pragma acc loop gang
  for (int i =0; i<N; i++) {
     {code2;} // gang - partitioned , worker - single
  }
  {code1.2;} // gang - redundant
}
```

- Marked loop is *partitioned* among *gangs*.
- Each gang executes one partition of the iteration space.
- Only one worker per gang is active: *worker-single mode*

Work sharing [2] – “gangs + workers”

```
# pragma acc parallel
{
 {code1.1;} // gang-redundant
 #pragma acc loop gang
 for (int i =0; i<N; i++) {
   {code2.1;} //gang-partitioned, worker-single
   #pragma acc loop worker
   for (int j =0; i<M; i++)
      {code3;} // gang-partitioned, worker-partitioned
   {code2.2;} // gang - partitioned , worker – single
 }
 {code1.2;} // gang - redundant
}
```

- Marked loop is partitioned among the workers of each gang.
  - All workers jointly execute the gang's iteration space.

Work sharing [3] : all levels

```
# pragma acc parallel
{
   {code1;} // gang - redundant
   # pragma acc loop gang
   for (int i =0; i<N; i++) {
     {code2;} // gang - partitioned , worker-single
     # pragma acc loop worker
     for (int j =0; i<M; i++) {
       {code3;} // worker - partitioned , vector-single
       # pragma acc loop vector
       for (int k =0; i<P; k++) {
         {code4;} // worker-partitioned, vector-partitioned
} } } }
```

- Marked loop is vectorised.
  - Makes use of SIMD instructions, where available.

Work sharing [4] : merge pragma's

```
#pragma acc parallel
{
  {code1.1;} // gang-redundant
  #pragma acc loop gang worker
  for (int i=0; i<N; i++) {
    {code3;} // gang-partitioned , worker-partitioned
    #pragma acc loop vector
       for (int j=0; i<M; i++)
         {code4;} // vectorised
  }
  {code1.2;} // gang-redundant
}
```

- Outer loop is partitioned among gangs and workers.
- Inner loop is vectorised.

Work sharing [5]: merge all

```
#pragma acc parallel
{
  {code1.1;} // gang-redundant
  #pragma acc loop gang worker vector collapse(2)
  for (int i =0; i<N; i++) {
   for (int j =0; i<M; i++) {
     {code4;}// gang-partitioned,worker-partitioned,vectorised
   }
  }
  {code1.2;} // gang-redundant
}
```

- Through “collapse” both loops are joined into one.
- The joined loop is partitioned among gangs and workers and vectorised.

VectorAdd revisited

```
void vecadd (float *a,float *b,float *c, int len){
#pragma acc parallel num_gangs(10) vector_length(128)
  {
  #pragma acc loop gang vector // work sharing
    for (int i =0; i<len; i ++) {
        c[i] = a[i] + b[i];
       }
} }
```

- Most likely*: 10 blocks, each with 128 threads
  - Numbers remain constant throughout offloaded region
  - Defaults chosen by application depending on
    - code in loaded region
    - properties of the device

*Most likely because the compiler decides the mapping.

Data copying

Explicit data copying

```
void vecadd (float* c,float *a,float *b, int len){

#pragma acc parallel copyin(a[0:len],b[0:len],len)
                     copyout(r[0:len]) {
  #pragma acc loop gang worker
    for (int i =0; i<len; i ++) {
        c[i] = a[i] + b[i];
      }
} }
```

- Allocates memory on device
- Copies data from device to host on exit from region
- De-allocates device memory on exit from region
- Data specification: addr[offset:size]

More clauses …

- copy(...)
  - Allocate + copyin + copyout + deallocate
- create(…)
  - Allocate + deallocate
    - No copy!
- present(…)
  - Data is present on device => look for it.
    - No copy!
- present_or_*(…)
  - If data is NOT present => copy/create/…

Separate data transfer

```
void vecadd(float *c,float *a,float *b, int len){
 #pragma acc parallel loop gang worker
  for (int i =0; i<len; i ++) {
       c[i] = a[i] + b[i];
  }
}


void v3(float* r, float* a, float* b, float* c, int len){
  #pragma acc data
     copyin(a[0:len],b[0:len],c[0:len],len)
     copyout(r[0:len]) {
     vecadd( r, a, b, len );
     vecadd( r, r, c, len );
  }
}
```

Explicit data updating

```
#pragma acc data copy (a[0:m+n])
{
  for (t=0; t<T; t++) {
  // host computes 0.. m
  // device computes m..m+n
      # pragma acc update device(a[0:m])
      # pragma acc update host(a[m:n])
} }
```

- Executable pragma:
  - (Partially) updates data on device with data from host
  - (Partially) updates data on host with data from device
- Can have multiple clauses
- Data transfers are executed in order of appearance
- *Must be run on host*

OpenMP 4.0/4.5/5.0

How about OpenMP?

- Offload first introduced in v4.0 of the standard
- Updated and enriched in v4.5 of the standard

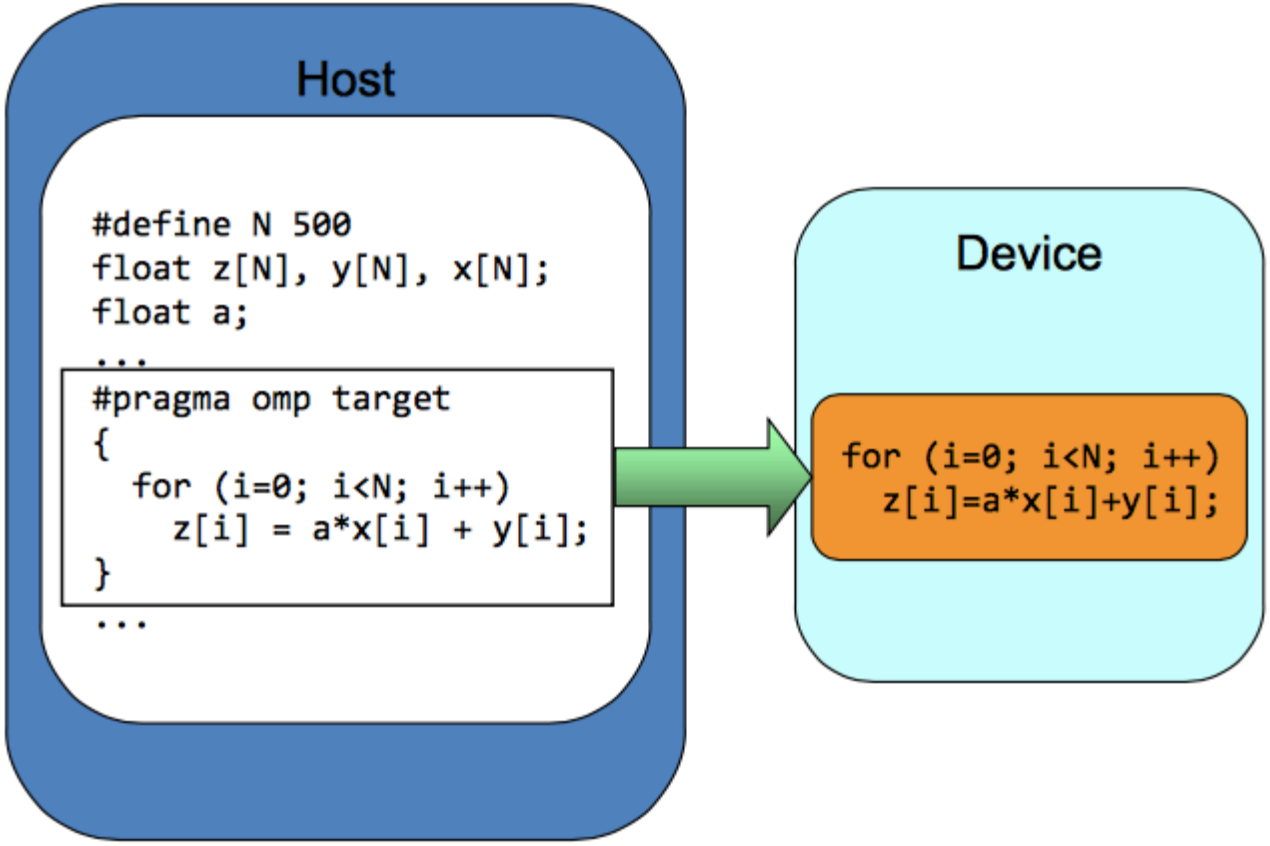


Image from https://openpowerfoundation.org/openmp-accelerator-support-for-gpu/

OpenMP: hosts and devices

- Host = where the OpenMP program starts
- Devices = where part of the code can be offloaded
  - Host-centric model: the host offloads code !
- Offloading = separate task + offloading to target + handle data movement
  - If target does not exist => execution on host !
- Various functions to enquire the existence of devices and where the code is running …

OpenMP accelerator support [1]

- **`#pragma omp target`**
  - Block of code to be offloaded to the device
  - Can include mapping of the data
    - If not, mapping is implicit
  - Can include if clauses

```
float p[N], v1[N], v2[N];

#pragma omp target if (N>1000)
                   map(to: v1, v2) map(from: p)
#pragma omp parallel for
for (i=0; i<N; i++) p[i] = v1[i] * v2[i];
```

OpenMP accelerator support [2]

- pragma omp teams
  - Creates “a league” of teams
  - Only the master thread in each team executes the code region
- pragma omp distribute
  - Work is distributed among (master threads of) teams
- pragma omp parallel (in this context)
  - When encountered by master thread, parallel work on all threads in team

```
#pragma omp target
#pragma omp teams distribute
for (j=0; j<N; j+=blocksize) {
   #pragma omp parallel
   for (i=0; i<blocksize; i++) p[i] = v1[i] * v2[i];
}
```

OpenMP accelerator support [3]

- Support for schedules for both loops
  - Inner and outer loop can use different schedules
- Support for reductions
- Support for asynchronous execution nowait clauses
  - Or using tasks

```
#pragma omp parallel
{
 #pragma omp master
 #pragma omp target teams distribute parallel for nowait \
      map(to:v1[0:n/2]) map(to:v2[0:n/2]) map(from:p[0:n/2])
   for(i=0; i<n/2; i++){ p[i] = v1[i]*v2[i]; }
 #pragma omp for schedule(dynamic,chunk)
   for(i=n/2; i<n; i++){ p[i] = v1[i]*v2[i]; }
}
```

In summary: OpenMP and OpenACC

Advantages:

- Considerably simpler to use than CUDA or OpenCL.
- More agile development, cheaper maintenance.
- Simplicity facilitates experimentation with alternatives.
- Large applications may be parallelized incrementally.

Disadvantages:

- Insight into impact of certain directives / clauses is reduced.
- False directives may lead to bad behaviour.
- Accelerator architectures can be very different from each other
  - performance portability remains a big issue.
- Performance implications of the simple/pretty syntax

Prescriptive vs. descriptive

- OpenMP directives are by design *prescriptive* in nature.
  - compiler is required to perform the requested parallelization
  - Parallelization will be performed the same way
  - Reproducible results from one compiler to the next.

  BUT: might require different sets of directives for different architectures.

- Most of OpenACC's directives are *descriptive* in nature.
  - compiler is free to compile the code whichever way it thinks is best
    - for the target architecture.
  - Different compilers ⇔ different results

Examples

Mandelbrot calculation: seq

- Double nested loop

```
 const float xmin = -1.5;
 const float ymin = -1.25;
 const float dx = 2.0 / xsize;
 const float dy = 2.5 / ysize;
for (unsigned int i = 0; i < xsize * ysize; i++) {
  unsigned int px = i % xsize;
  unsigned int py = i / xsize;
  float x0 = xmin + px*dx;
  float y0 = ymin + py*dy;
  float x = 0.0;
  float y = 0.0;
  unsigned int j = 0;
  for(j = 0; x*x + y*y < 4.0 && j < MAX_ITERS; j++)  {
    float xtemp = x*x - y*y + x0;
    y = 2*x*y + y0;
    x = xtemp;
  }
  buf[i] = j;
}
```

Mandelbrot calculation: OpenACC

- Use data copying, use the ”loop independent” pragma

```
#pragma acc parallel copy(buf[0:xsize*ysize - 1]) {
 const float xmin = -1.5;
 const float ymin = -1.25;
 const float dx = 2.0 / xsize;
 const float dy = 2.5 / ysize;
#pragma acc loop independent
for (unsigned int i = 0; i < xsize * ysize; i++) {
  unsigned int px = i % xsize;
  unsigned int py = i / xsize;
  float x0 = xmin + px*dx;
  float y0 = ymin + py*dy;
  float x = 0.0;
  float y = 0.0;
  unsigned int j = 0;
  for(j = 0; x*x + y*y < 4.0 && j < MAX_ITERS; j++)  {
    float xtemp = x*x - y*y + x0;
    y = 2*x*y + y0; x = xtemp;
  }
  buf[i] = j; }
```

Matrix multiplication: v1 and v2

- Calculate C = A x B, with A,B = square matrices

```
for (i = 0; i < SIZE; ++i) {
  for (j = 0; j < SIZE; ++j) {
    c[i][j] = 0.0;
    for (k = 0; k < SIZE; ++k) {
      c[i][j] += a[i][k] * b[k][j];
    }
  }
}
```

```
for (i = 0; i < SIZE; ++i) {
  for (j = 0; j < SIZE; ++j) {
    tmp = 0.0;
    for (k = 0; k < SIZE; ++k)
      tmp += a[i][k] * b[k][j];
    c[i][j] = tmp;
  }
}
```

Matrix multiplication [2]: OpenACC

- Use automated loop parallelization and data transfers

```
#pragma acc data copyin(a,b) copy(c)
#pragma acc kernels
#pragma acc loop auto
  for (i = 0; i < SIZE; ++i) {
#pragma acc loop auto
    for (j = 0; j < SIZE; ++j) {
#pragma acc loop auto
      for (k = 0; k < SIZE; ++k) {
        c[i][j] += a[i][k] * b[k][j];
      }
    }
  }
```

What happens if we replace “auto” with “seq”?

Matrix multiplication [3]: more control

- Calculate C = A x B, with A,B = square matrices

```
#pragma acc data copyin(a,b) copy(c)
#pragma acc kernels
#pragma acc loop gang(32)
  for (i = 0; i < SIZE; ++i) {
#pragma acc loop vector(16)
    for (j = 0; j < SIZE; ++j) {
      tmp=0.0f;
#pragma acc loop reduction(+:tmp)
      for (k = 0; k < SIZE; ++k) {
        tmp += a[i][k] * b[k][j];
      }
      c[i][j] = tmp;
    }
  }
```

What happens with and without the reduction?

Matrix multiplication: v3

- Better caching for the sequential version:

```
for (k=0; k<n; k++) {
   for (i=0; i<n; i++) {
      r = a[i][k];
      for (j=0; j<n; j++)
          c[i][j] += r * b[k][j];
   }
}
```

- How do we parallelize this one?

TO DO: Hands-on #2 [1]

- Parallelize the following codes and compare the performance, again, when using the three different compilation targets: sequential, multi-core CPU, GPU
  - Mandelbrot
  - Matrix multiplication
    - V1 – basic
    - V2 – temporary variable for reduction
    - V3 – reorder loops

- Try different configurations, including different data transfers, gang sizes, vector/worker sizes, etc.

TO DO: Hands on #2 [2]

- Applications are available in:
  - **`mandelbrot.c`**
  - **`mmul.c`**
  - **`mmul-v2.c`**

- Compilation reminder:

```
pgcc -acc -ta=<fill in> -Minfo=accel -O3 vecAdd.c -o <exe-name>
```

- Try different compilation options for “-ta” :
  - Target: **host** => sequential execution
  - Target: **multicore** => parallel execution on the multi-core CPU host
  - Target: **tesla:cc35** => offloading to the GPU of compute capability 3.5

In summary…

GPU programming

- Different solutions to program GPUs
  - High-level => depend on compilers and well-written sequential code
    - … plus a good selection of pragma's
  - Low-level => depend on the use of low-level architectural details

- Pragma-based programming
  - Advantages:
    - Easy to use for legacy code
    - Soon-to-be in most compilers
  - Disadvantages:
    - Less control over parallelism
    - Less control over the use of low-level features
    - Performance depends on compiler more than user

(Unrequested) Advice

- Try pragma-based models first
  - low effort
  - Still helpful to understand the application
- Understand bottlenecks
  - Use profiling
- Understand expected gain
  - Use predictive models
- Consider re-writing with CUDA (or SyCL/OpenCL) if …
  - You need to have multiple kernels running in parallel
  - You have specific use for shared memory
  - You have complex synchronization patterns
  - You want to investigate algorithmic changes
  - … anything where detailed features are needed …