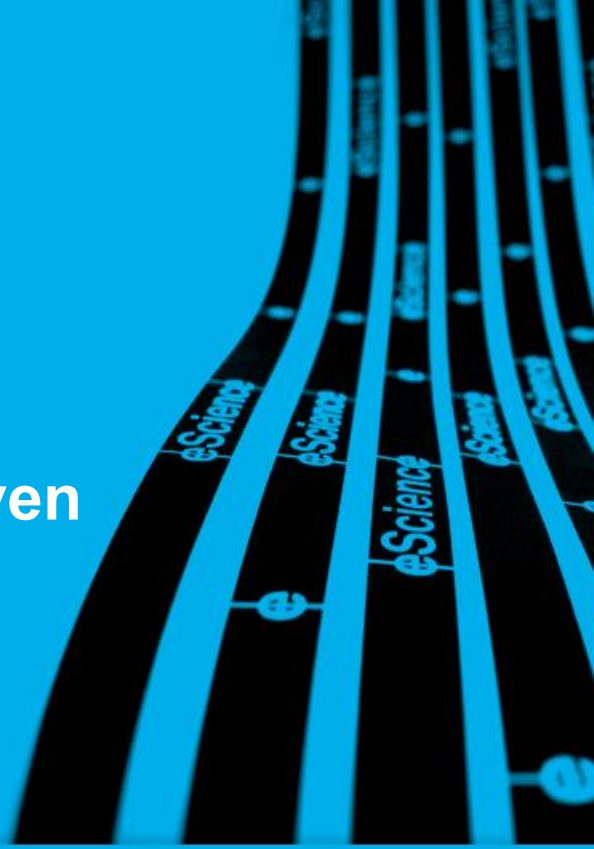
NLeSC GPU Course

March 14 2017

Rob van Nieuwpoort & Ben van Werkhoven



netherlands eScience center

by SURF & NWO

Schedule

- 10:00 – 10:30 Introduction to GPU Computing
- 10:30 – 11:15 High-level intro to CUDA Programming Model
- 11:15 – 11:45 1st Hands-on Session
- 11:45 – 12:00 Solution to first hands-on

- 12:00 – 13:00 Lunch break

- 13:00 – 15:00 CUDA Programming model Part 2 with 2 hands-on sessions
- 15:00 – 15:30 CUDA Program execution
- 15:30 – 16:30 Performance analysis and use case

Download the slides!

- Get your own copy of the slides so you can read along and click on links
  See: https://github.com/benvanwerkhoven/gpu-course/

- Our slides are sometimes very wordy, this is intentional, so they may serve as a reference that you can read again later

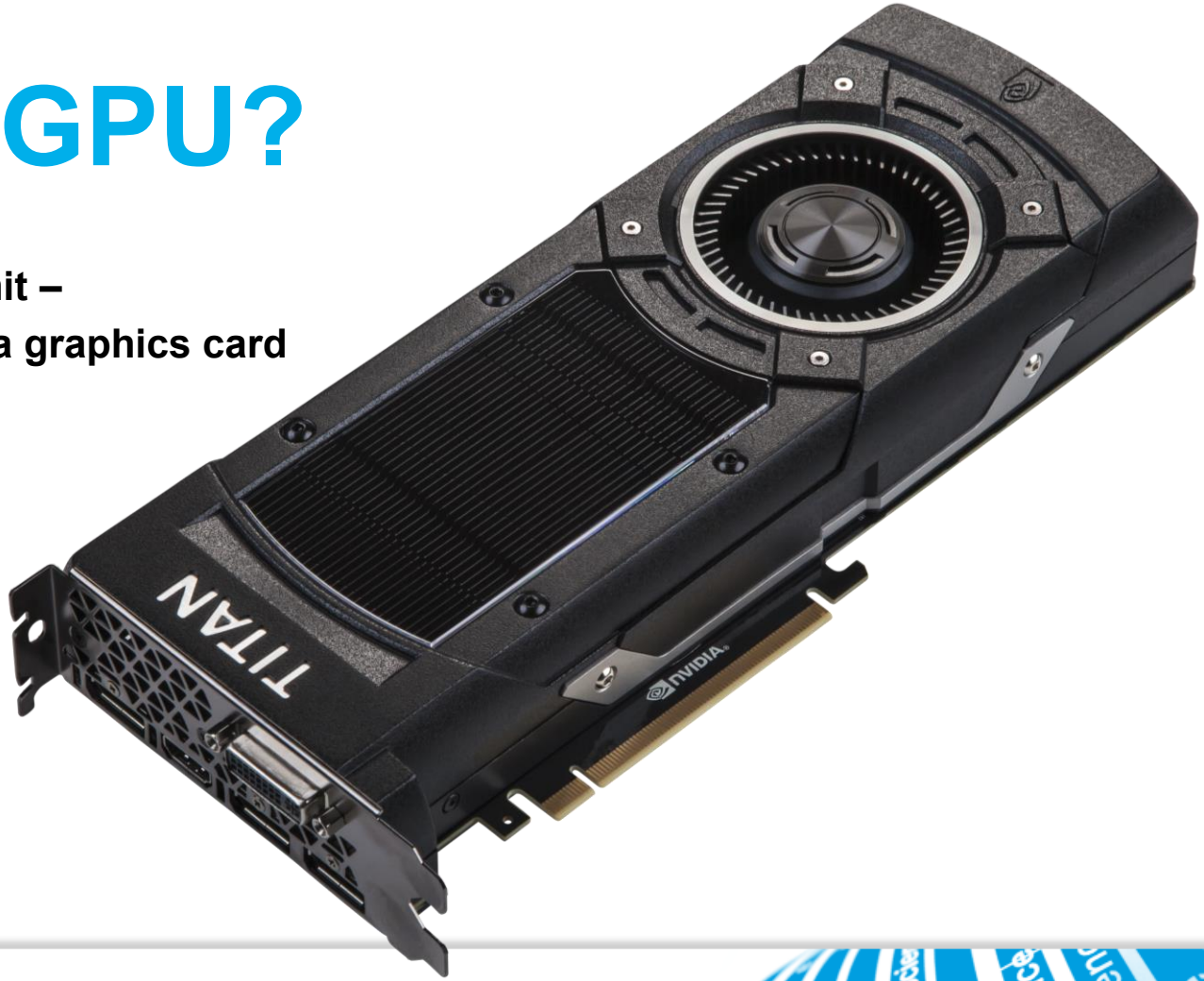
- In code samples on the slides we sometimes leave out ‘{‘ and ‘}’ to save space

Introduction to GPU Computing

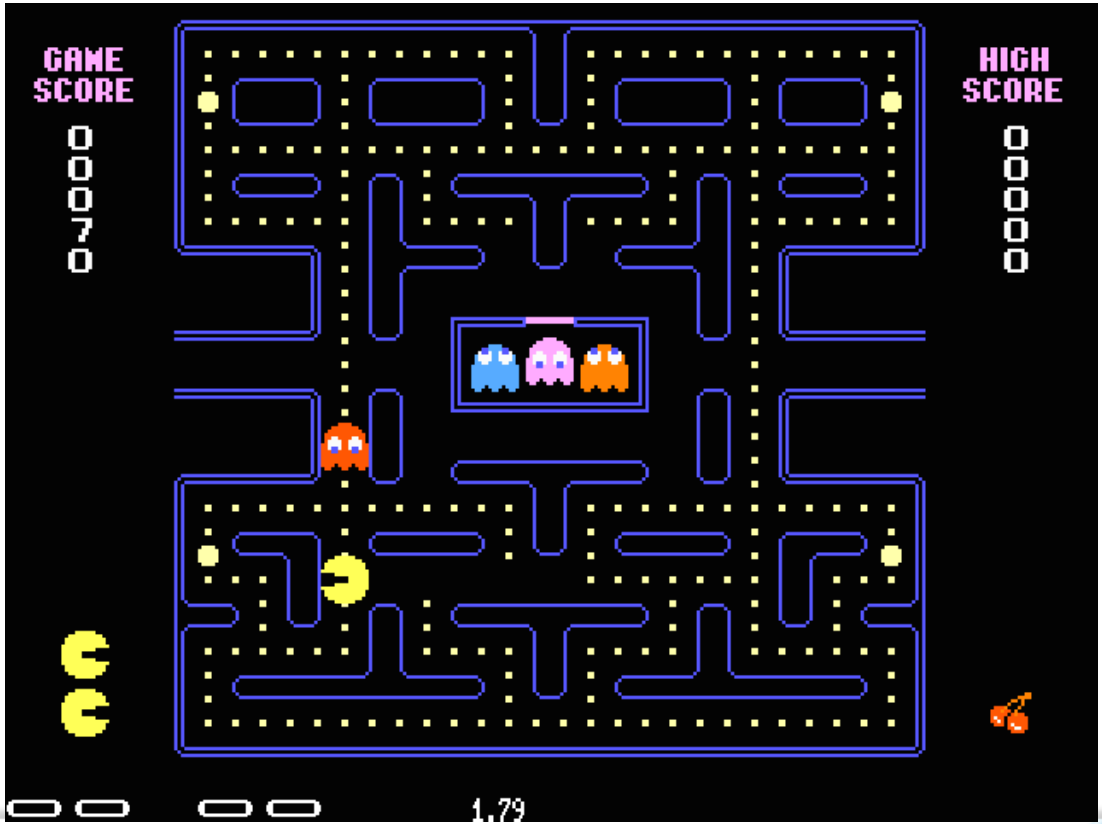
What is a GPU?

- **Graphics Processing Unit –**
  The computer chip on a graphics card

- **GPGPU**



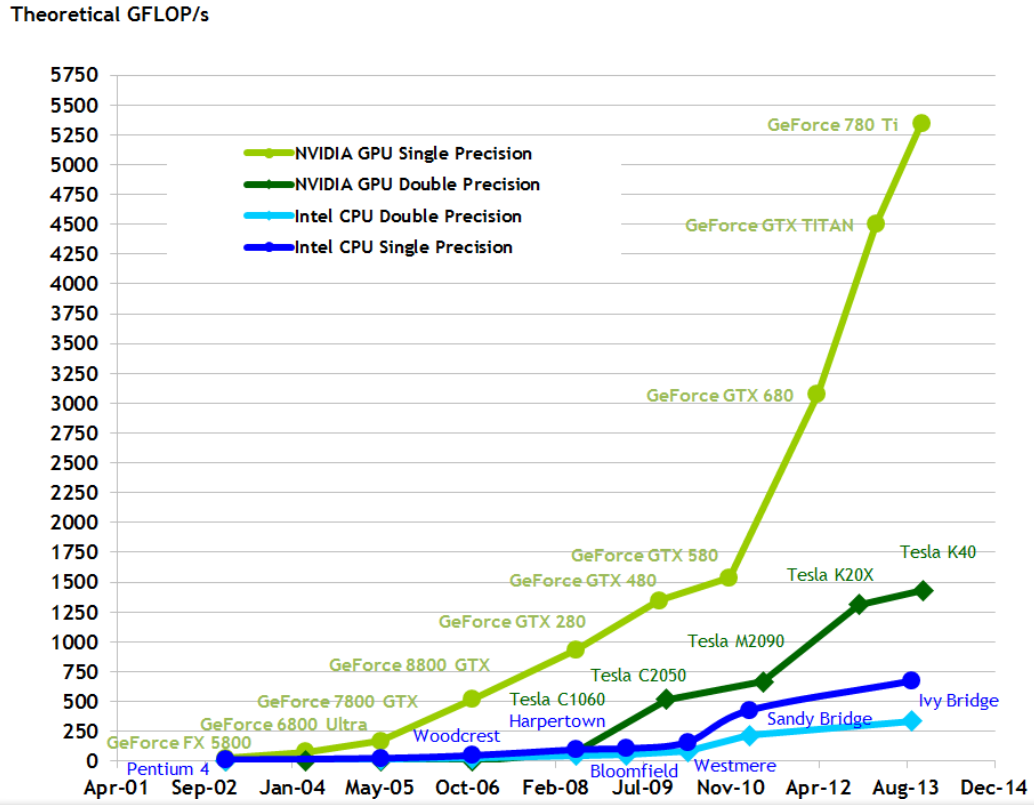
Graphics in 1980



Graphics in 2000

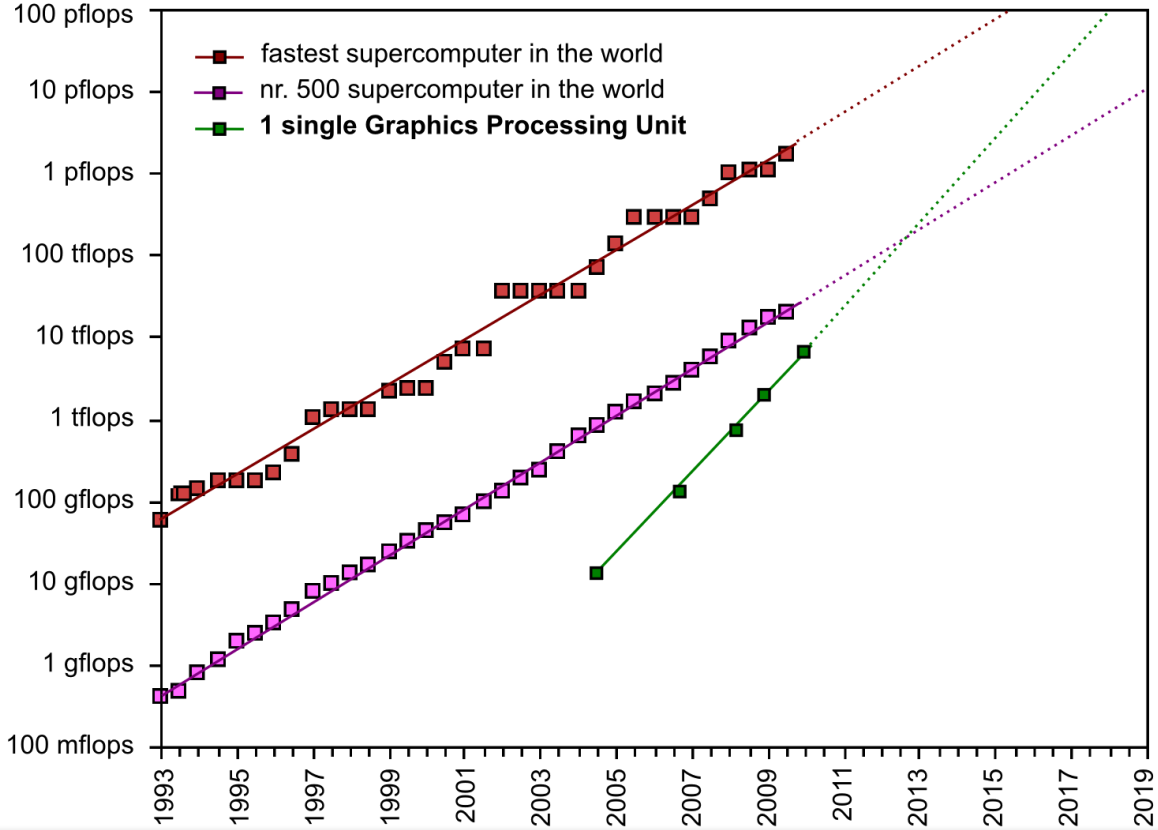
Graphics now

Compute performance

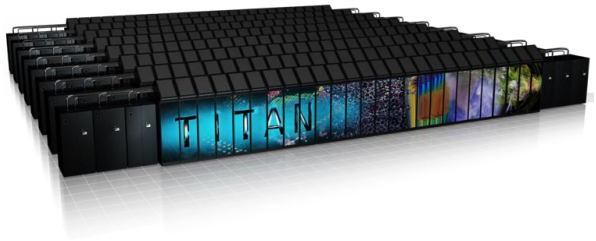


(According to Nvidia)

GPUs vs supercomputers ?



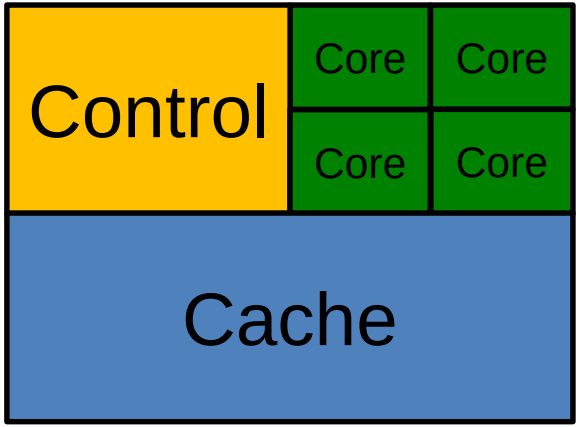
Oak Ridge's Titan

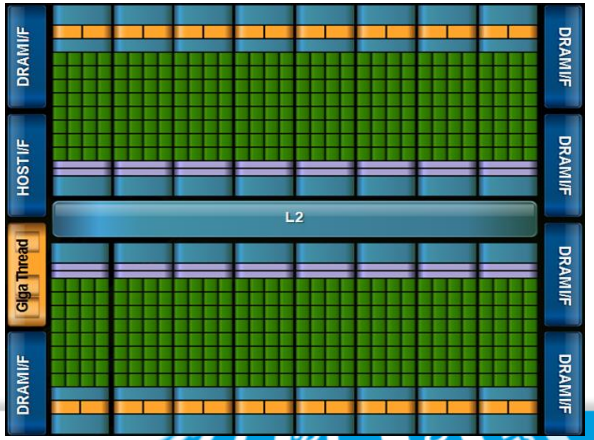


- Number 3 in top500 list: 27.113 pflops peak, 8.2 MW power
- 18.688 AMD Opteron processors x 16 cores = 299.008 cores
- 18.688 Nvidia Tesla K20X GPUs x 2688 cores = 50.233.344 cores

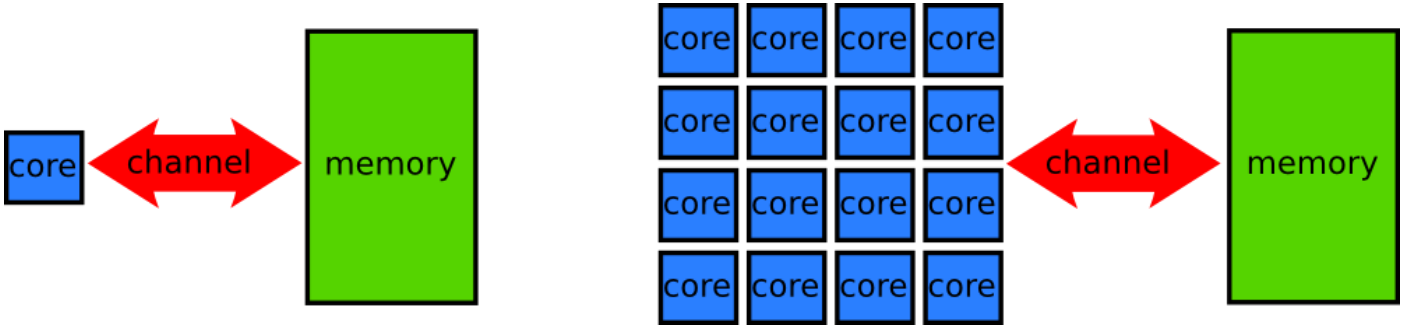
CPU vs GPU Hardware

- **Different goals produce different designs**
  - **GPU assumes work load is highly parallel**
  - **CPU must be good at everything, parallel or not**
- **CPU: minimize latency experienced by 1 thread**
  - **Big on-chip caches**
  - **Sophisticated control logic**
- **GPU: maximize throughput of all threads**
  - **Multithreading can hide latency, so no big caches**
  - **Control logic**
    - **Much simpler**
    - **Less: share control logic across many threads**





It's all about the memory

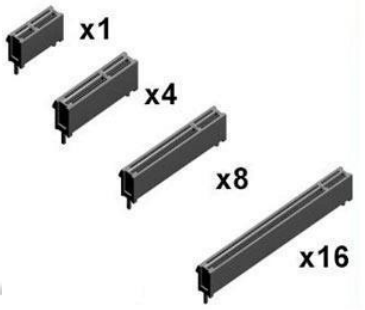


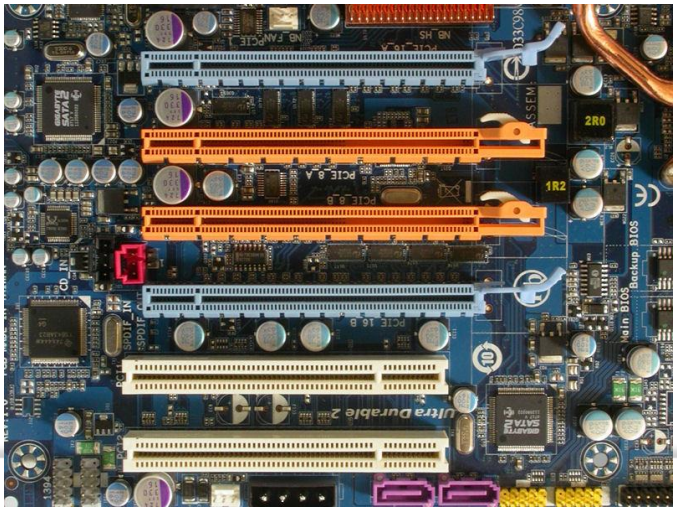
Many-core architectures

From Wikipedia: “A many-core processor is a multi-core processor in which the number of cores is large enough that traditional multi-processor techniques are no longer efficient — largely because of issues with congestion in supplying instructions and data to the many processors.”

Integration into host system

- **PCI-e 3.0 achieves about 16 GB/s**
- **Comparison: GPU device memory bandwidth is 320 GB/s for GTX1080**





Why GPUs?

- **Performance**
  - Large scale parallelism
- **Power Efficiency**
  - Use transistors more efficiently
  - #1 in green 500 uses NVIDIA Tesla P100
- **Price (GPUs)**
  - Huge market, bigger than Hollywood
  - Mass production, economy of scale
  - Gamers pay for our HPC needs!

When use GPU Computing?

- **When:**
  - **Thousands or even millions of elements that can be processed in parallel**

- **Very efficient for algorithms that:**
  - **have high arithmetic intensity (lots of computations per element)**
  - **have regular data access patterns**
  - **do not have a lot of data dependencies between elements**
  - **do the same set of instructions for all elements**

A high-level intro to the CUDA Programming Model

CUDA Programming Model

Before we start:

- **I'm going to explain the CUDA Programming model**
- **I'll try to avoid talking about the hardware for now**
- **For the moment, make no assumptions about the backend or how the program is executed by the hardware**
- **I will be using the term 'thread' a lot, this stands for 'thread of execution' and should be seen as a parallel programming concept. Do not compare them to CPU threads.**

CUDA Programming Model

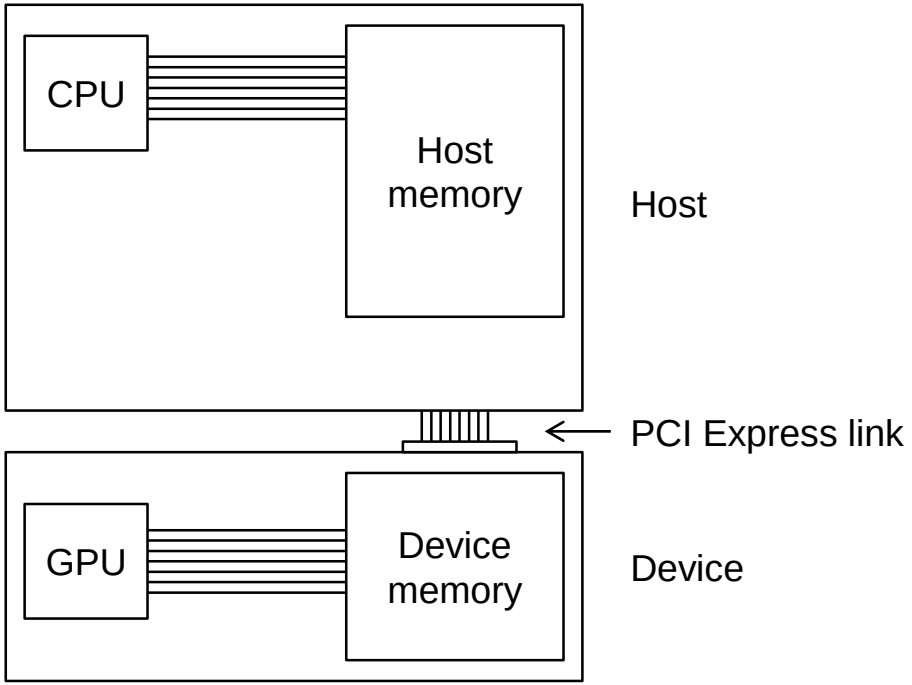
- The CUDA programming model separates a program into a host (CPU) and a device (GPU) part.

- The host part: allocates memory and transfers data between host and device memory, and starts GPU functions

- The device part consists of functions that will execute on the GPU, which are called *kernels*
- Kernels are executed by huge amounts of threads at the same time
- The data-parallel workload is divided among these threads
- The CUDA programming model allows you to code for each thread individually

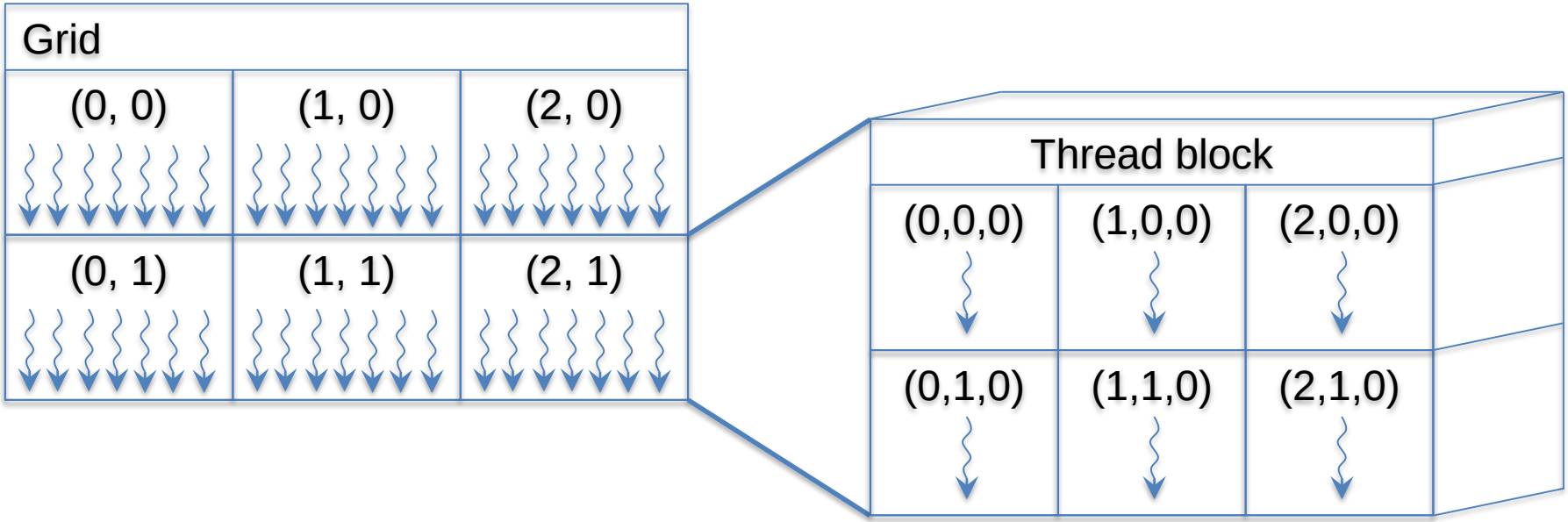
Data management

- **The GPU is located on a separate device**
- **The host program manages the allocation and freeing of GPU memory**
  - C:
    - `cudaMalloc()`
    - `cudaFree()`
  - Python:
    - `mem_alloc()`
- **Host program also copies data between different physical memories**
  - C:
    - `cudaMemcpy()`
  - Python:
    - `memcpy_htod() or memcpy_dtoh()`



Thread Hierarchy

- **Kernels are executed in parallel by possibly millions of threads, so it makes sense to try to organize them in some manner**



Threads

- **In the CUDA programming model a thread is the most fine-grained entity that performs computations**
- **Threads direct themselves to different parts of memory using their built-in variables** `threadIdx.xyz` **(thread index *within* the thread block)**
- **Example:**

```
for (i=0; i<N; i++) {
    c[i] = a[i] + b[i];
}
```

Create a single thread block of N threads:

```
i = threadIdx.x;
c[i] = a[i] + b[i];
```

- **Effectively the loop is 'unrolled' and spread across N threads**

Thread blocks

- **Threads are grouped in thread blocks, allowing you to work on problems larger than the maximum thread block size**
- **Thread blocks are also numbered, using the built-in variable** `blockIdx.xy` **containing the index of each block within the grid.**
- **Total number of threads created is always a multiple of the thread block size, possibly not exactly equal to the problem size**
- **Other built-in variables are used to describe the thread block dimensions** `blockDim.xyz` **and grid dimensions** `gridDim.xy`

Starting a kernel

- **The host program sets the number of threads and thread blocks when it launches the kernel**

- ```
  //create variables to hold grid and thread block dimensions
  dim3 threads(x, y, z)
  dim3 grid(x, y)

  //launch the kernel
  vector_add<<<grid, threads>>>(c, a, b);

  //wait for the kernel to complete
  cudaDeviceSynchronize();
  ```
  ```

Setup hands-on session

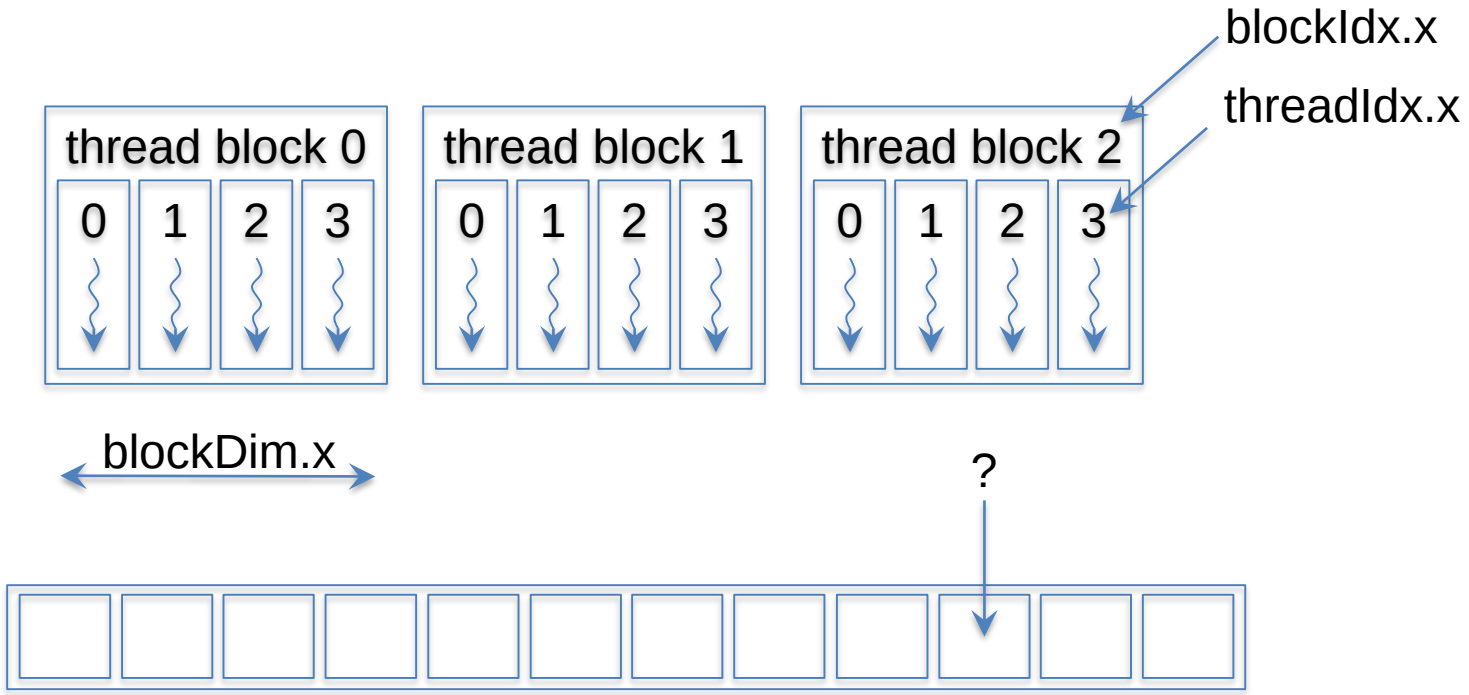
- **Login on DAS-5 (fs0.das5.cs.vu.nl)**
- **Execute (recommended to add to your .bashrc):**
  - `module load cuda80`
  - `alias gpurun="srun -N 1 -C TitanX --gres=gpu:1"`

- **Additional setup if using Python (optional):**
  - `wget https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh`
  - `bash Miniconda3-latest-Linux-x86_64.sh` **(allow to add to .bashrc)**
  - `export PATH="$HOME/miniconda3/bin:$PATH"`
  - `pip install numpy`
  - `pip install pycuda`

First hands-on session

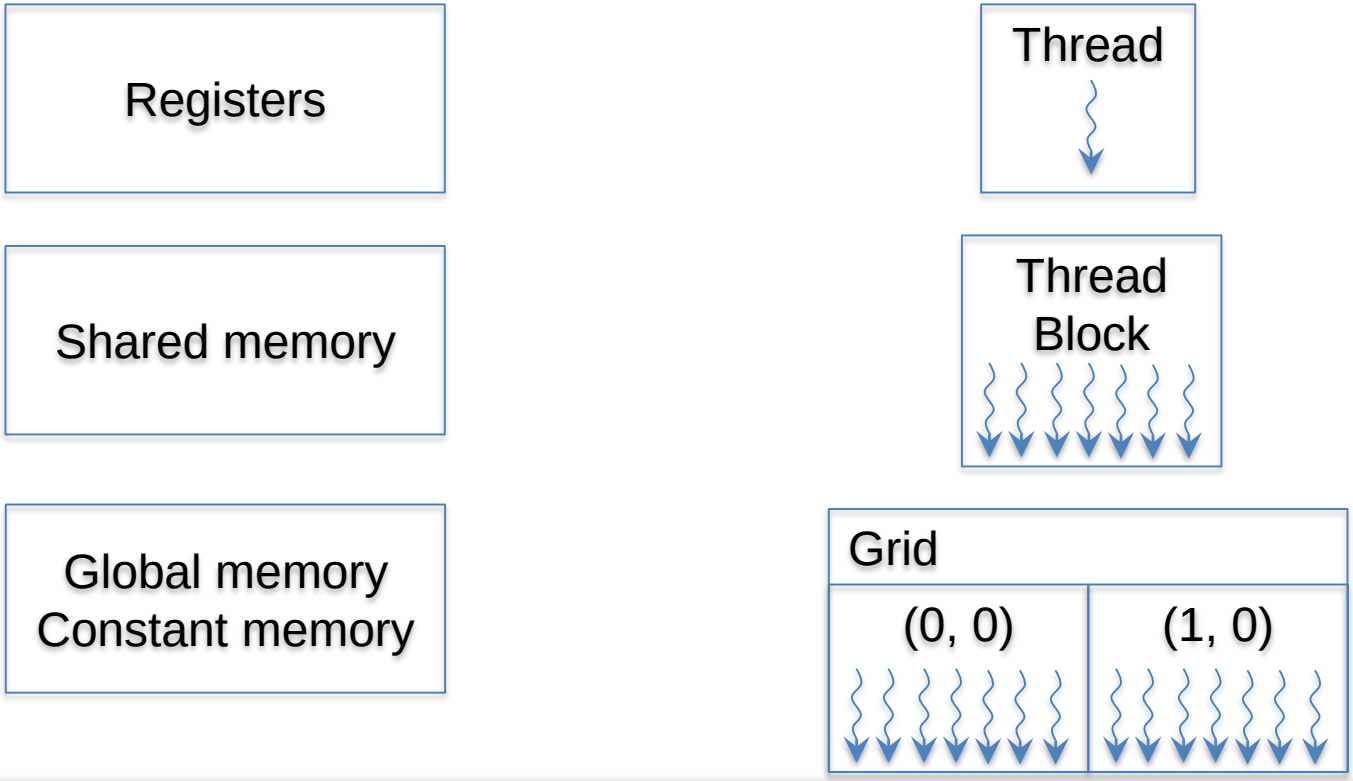
- **Clone the git repository:**
  - `git clone https://github.com/benvanwerkhoven/gpu-course.git`
- **Change to directory vector_add**
- **Using C:**
  - **Compile by typing** `make`**, run by typing** `gpurun vector_add`
- **Using Python:**
  - **Run by typing** `gpurun ./vector_add.py`
- **Make sure you understand everything in the code, and complete the exercise!**
- **Hints:**
  - **Look at how the kernel is launched in the host program**
  - `threadIdx.x` **is the thread index within the thread block**
  - `blockIdx.x` **is the block index within the grid**
  - `blockDim.x` **is the dimension of the thread block**

Hint


blockIdx.x
threadIdx.x
thread block 0
0
1
2
3
thread block 1
0
1
2
3
thread block 2
0
1
2
3
blockDim.x
?

CUDA Programming model
Part 2

CUDA memory hierarchy



Memory space: Registers

- **Example:**

```
__global__ void matmul_kernel(float *C, float *A, float *B) {
    int tx = threadIdx.x;      //local variable in registers
    float local_sum[4];        //small compile-time sized array in registers
```

- **Registers**
  - **Thread-local scalars or small constant size arrays are stored as registers**
  - **Implicit in the programming model**
  - **Behavior is very similar to normal local variables**
  - **Not persistent, after the kernel has finished, values in registers are lost**

Memory space: Global

- **Example:**

```
__global__ void matmul_kernel( float *C,  //C points to global memory
                               float *A,  //A points to global memory
                               float *B)  //B points to global memory
{
```

- **Global memory**
  - **Allocated by the host program using** `cudaMalloc()`
  - **Initialized by the host program using** `cudaMemcpy()` **or previous kernels**
  - **Persistent, the values in global memory remain across kernel invocations**
  - **Not coherent, writes by other threads will not be visible until kernel has finished**

Memory space: Constant

```
__constant__ float filter[filter_width * filter_height]; //initialized by a host function
__global__ void convolution_kernel(float *output, float *input) {
   ...
   for (j = 0; j < filter_height; j++) {
      for (i = 0; i < filter_width; i++) {
         sum += input[y + j][x + i] *
               filter[j * filter_width + i]; //index j and i do not depend on threadIdx (x and y)
      }
   }
```

- **Constant memory**
  - **Statically defined by the host program using `__constant__` qualifier**
  - **Defined as a global variable, visible only within the same translation unit**
  - **Initialized by the host program using `cudaMemcpyToSymbol()`**
  - **Read-only to the GPU, cannot be accessed directly by the host**
  - **Values are cached in a special cache optimized for broadcast access by multiple threads simultaneously, access should not depend on `threadIdx`**

2nd Hands-on

- Go to directory pnpoly, look at the source files
- Store the vertices in constant memory space:
  - Declare a `float2` array of size `VERTICES` as a global variable using the `__constant__` qualifier
  - Make sure the constant memory array is used inside the kernels, instead of the currently used '`vertices`' array in global memory, just leave it unused in the kernel
- In C:
  - Change the `cudaMemcpy()` for filter to `cudaMemcpyToSymbol()` and make sure it copies to the right place
  - See CUDA documentation on `cudaMemcpyToSymbol`
- In Python:
  - Python users can use `memcpy_htod()`, but need to find the symbol to copy to
  - See PyCuda documentation on `get_global`
- Leave the global memory copy of vertices in place, the reference kernel uses it

Memory space: Shared

```
__global__ void histogram(int *output, int *values, int n) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    __shared__ int sh_output[NUM_BINS];                    //declare shared memory array
    if(i < n) {
        int bin = values[i];
        atomicAdd(&sh_output[bin], 1);                     //increment bin in shared memory
        __syncthreads();                                   //wait for all threads
    ...
```

- **Shared memory**
  - **Variables have to be declared using `__shared__` qualifier, size known at compile time**
  - **In the scope of thread block, all threads in a thread block see the same piece of memory**
  - **Not initialized, threads have to fill shared memory with meaningful values**
  - **Not persistent, after the kernel has finished, values in shared memory are lost**
  - **Not coherent, `__syncthreads()` is required to make writes visible to other threads within the thread block**

Shared memory: Example

```
__global__ void transpose(int h, int w, float* output, float* input) {
    int i = threadIdx.y + blockIdx.y * block_size_y;
    int j = threadIdx.x + blockIdx.x * block_size_x;

    __shared__ float sh_mem[block_size_y][block_size_x];      //declare shared memory array

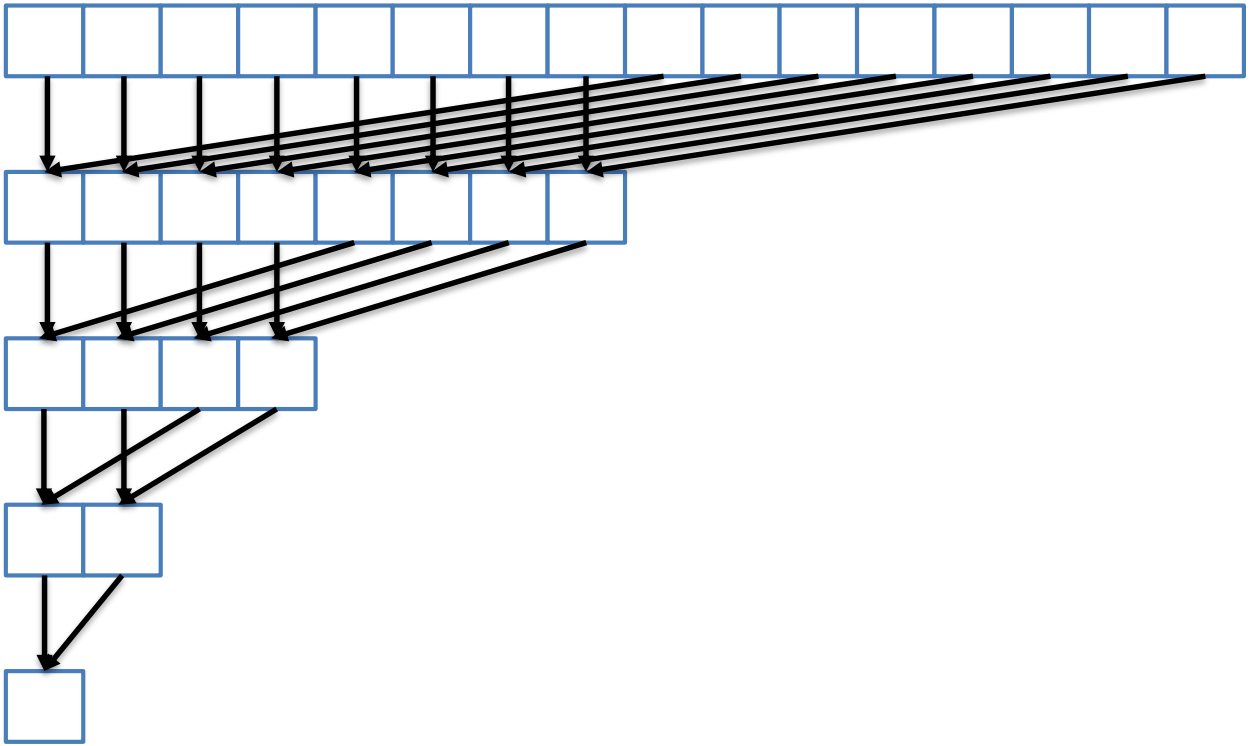
    if (j < w && i < h) {
          sh_mem[threadIdx.y][threadIdx.x] = input[i*w+j];   //fill shared with values from global
    }
    __syncthreads();                                         //wait for all thread in block

    i = threadIdx.x + blockIdx.y * block_size_y;
    j = threadIdx.y + blockIdx.x * block_size_x;
    if (j < w && i < h) {
          output[j*h+i] = sh_mem[threadIdx.x][threadIdx.y]; //store to global using shared memory
    }
}
```

3rd Hands-on

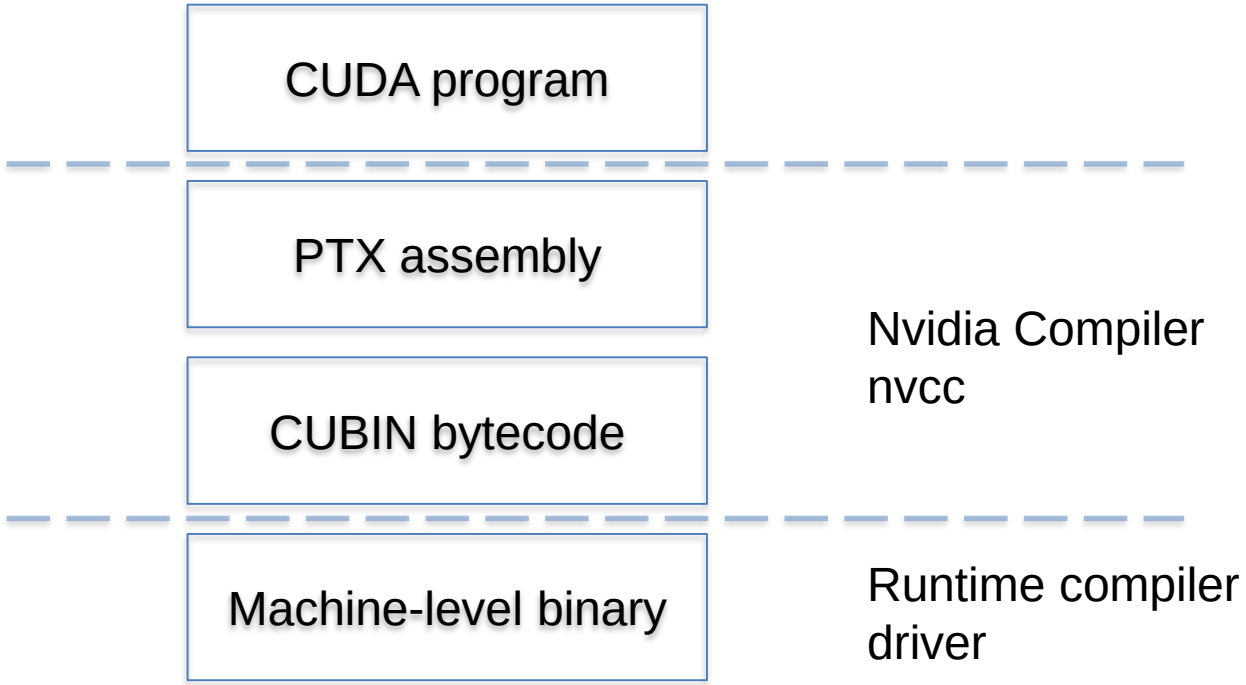
- Go to directory reduction, look at the source files
- Make sure you understand everything in the code
- Task:
  - Implement the kernel such that shared memory is used to sum the per-thread partial sums into a single per-thread block partial sum
- Hints:
  - The number of thread blocks does not depend on n. All threads from all blocks first iterate (collectively) over the problem size (n) to obtain a per-thread partial sum
  - Within the thread block the per-thread partial sums are to be combined to a per-thread block partial sum
  - Each thread block stores its partial sum to `out_array[blockIdx.x]`
  - The kernel is called twice, the second kernel is executed with only one thread block to combine all per-block partial sums to a single sum

Hint – Parallel Summation



CUDA Program execution

Compilation



Translation table

CUDA	OpenCL	OpenACC	OpenMP 4
Grid	NDRange	compute region	parallel region
Thread block	Work group	Gang	Team
Warp	CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE	Worker	SIMD Chunk
Thread	Work item	Vector	Thread or SIMD

- **Note that for the mapping is actually implementation dependent for the open standards and may differ across computing platforms**
- **Not too sure about the OpenMP 4 naming scheme, please correct me if wrong**

How threads are executed

- Remember: all threads in a CUDA kernel execute the exact same program
- Threads are actually executed in groups of (32) threads called *warps*
- Threads within a warp all execute one common instruction simultaneously
- The context of each thread is stored separately, as such the GPU stores the context of all currently active threads
- The GPU can switch between warps even after executing only 1 instruction, effectively hiding the long latency of instructions such as memory loads

Predication

- All threads in a warp execute the exact same *instruction* at the same cycle

```
mad.f32    %f1, %f2, %f3, %f1;          // c += a*b;
```

- The same instruction, but on different data

- What about control flow instructions? (if, else, for, while)
  - All threads in the warp execute all live paths, with some threads predicated

```
if (a > 0.0f)
```

  - This is less efficient, but not always bad.
  - Avoid data-dependent conditional branching if possible

- Thread index-dependent branching is usually harmless, in particular when you respect the warp size

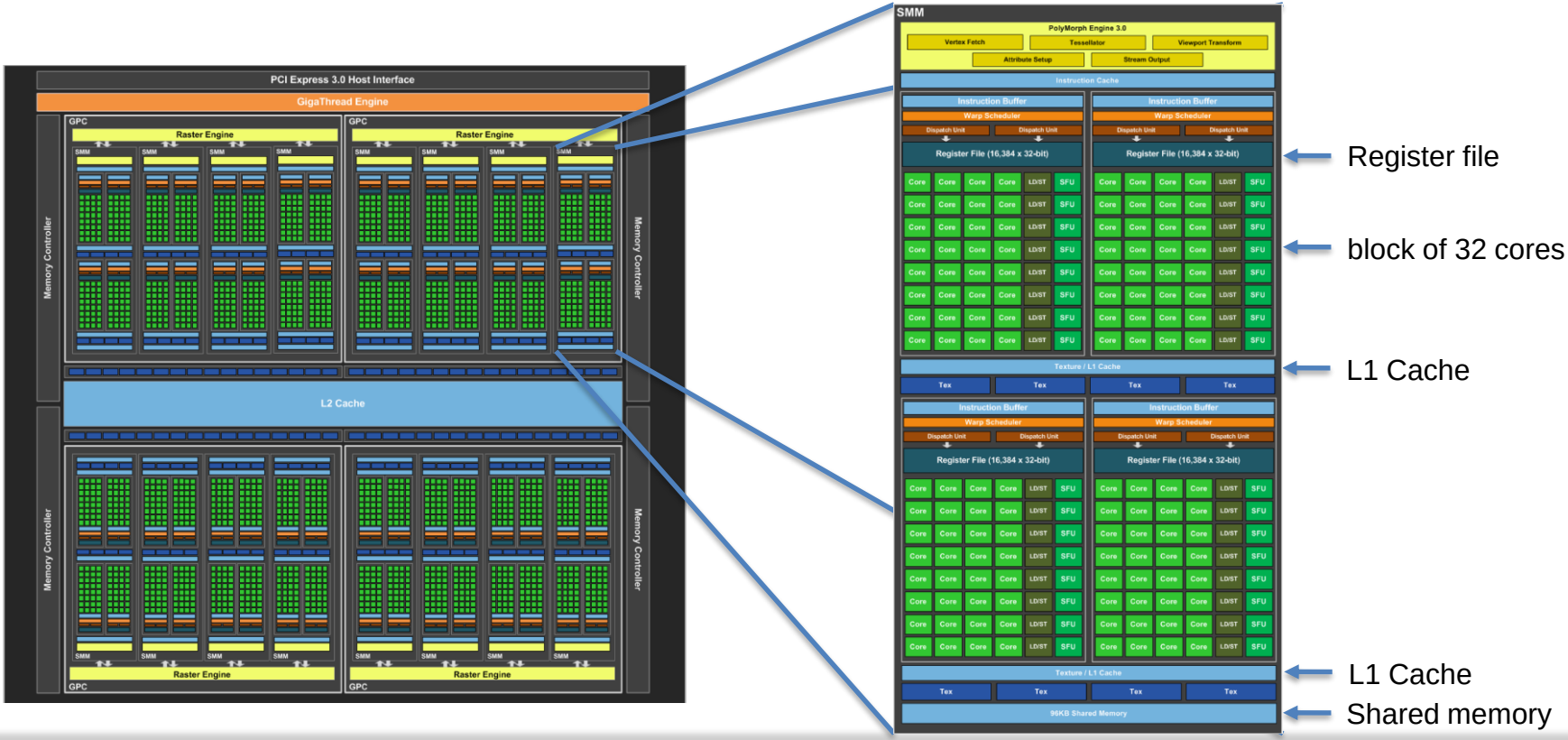
```
if (threadIdx.x < 32)
```

Maxwell Architecture

Streaming multiprocessor (SM)

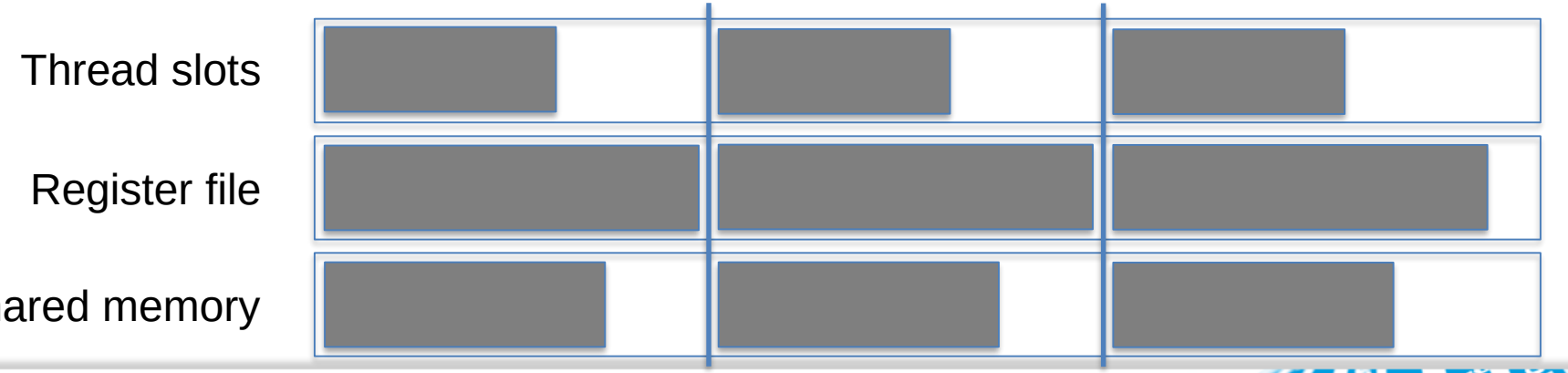
32 core block

Maxwell Architecture



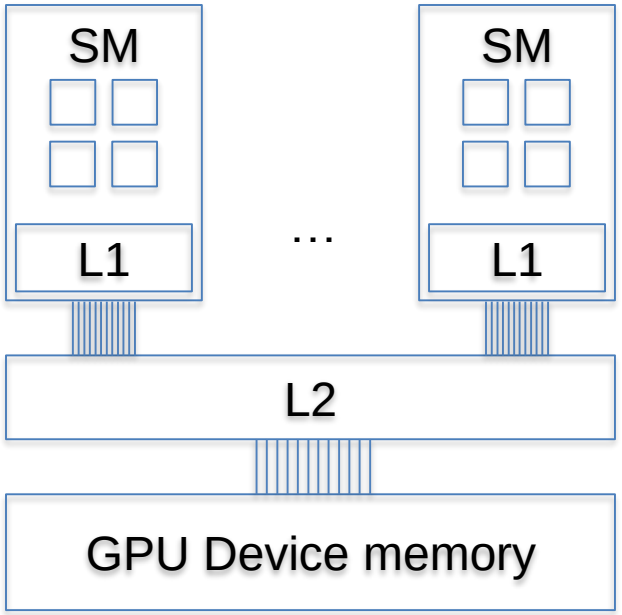
Resource partitioning

- **The GPU consists of several (1 to 56) *streaming multiprocessors* (SMs)**
- **The SMs are fully independent**
- **Each SM contains several resources: Thread and Thread Block slots, Register file, and Shared memory**
- **SM Resources are dynamically partitioned among the thread blocks that execute concurrently on the SM, resulting in a certain *occupancy***



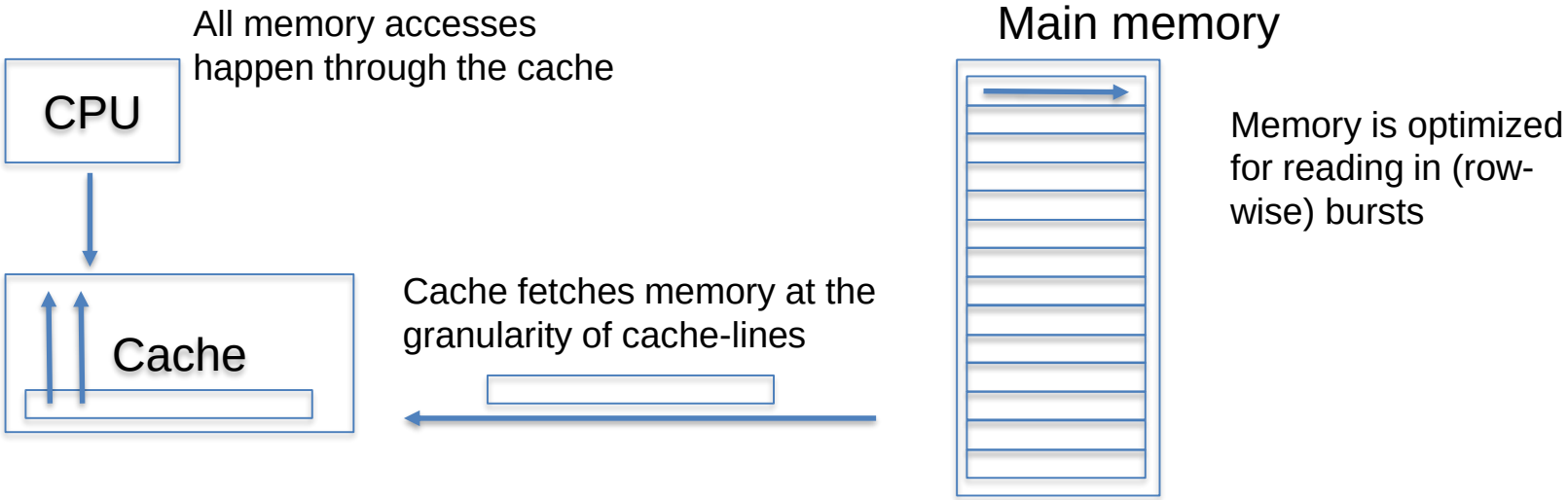
Global Memory access

- Global memory is cached at L2, and for some GPUs also in L1
- When a thread reads a value from global memory, think about:
  - The total number of values that are accessed by the warp that the thread belongs to
  - The cache line length and the number of cache lines that those values will belong to
  - Alignment of the data accesses to that of the cache lines



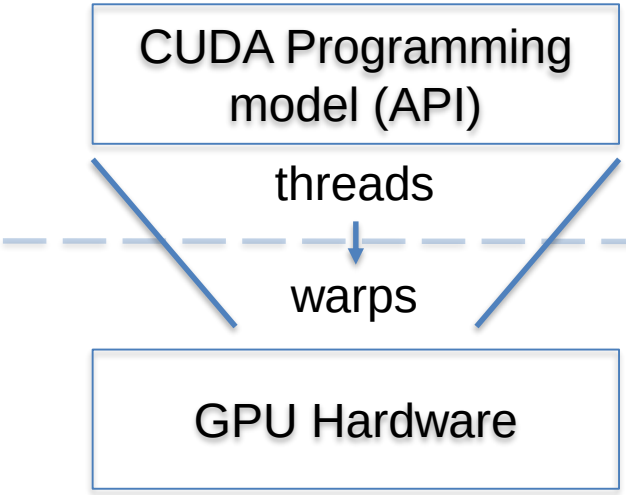
Cached memory access

- **The memory hierarchy is optimized for certain access patterns**



Subsequently accessing values that are adjacent on the same cache line is much faster than when each access requires a new cache line to be fetched

Overview



Think in terms of threads
Reason on program correctness

Think in terms of warps
Reason on program performance

A simple performance model: Roofline

Arithmetic intensity

- **The number of arithmetic (floating point) operations per byte of memory that is accessed**
  - **Is the program compute intensive or data intensive on a particular architecture?**

- **Ignore "overheads"**
  - **Loop counters**
  - **Array index calculations**
  - **Etc.**

RGB to gray

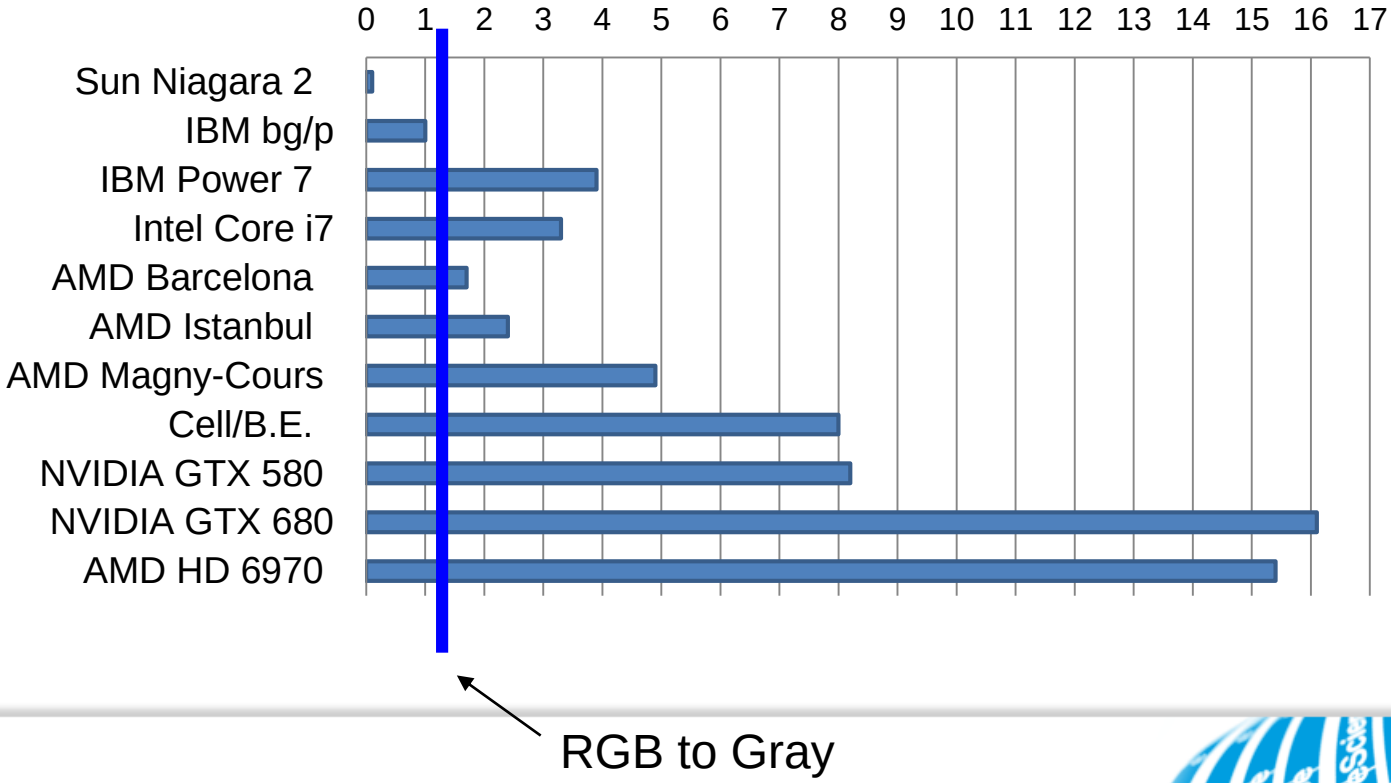
```
for (int y = 0; y < height; y++) {
   for (int x = 0; x < width; x++) {
    Pixel pixel = RGB[y][x];
    gray[y][x] =
          0.30 * pixel.R
        + 0.59 * pixel.G
        + 0.11 * pixel.B;
   }
}
```

RGB to gray

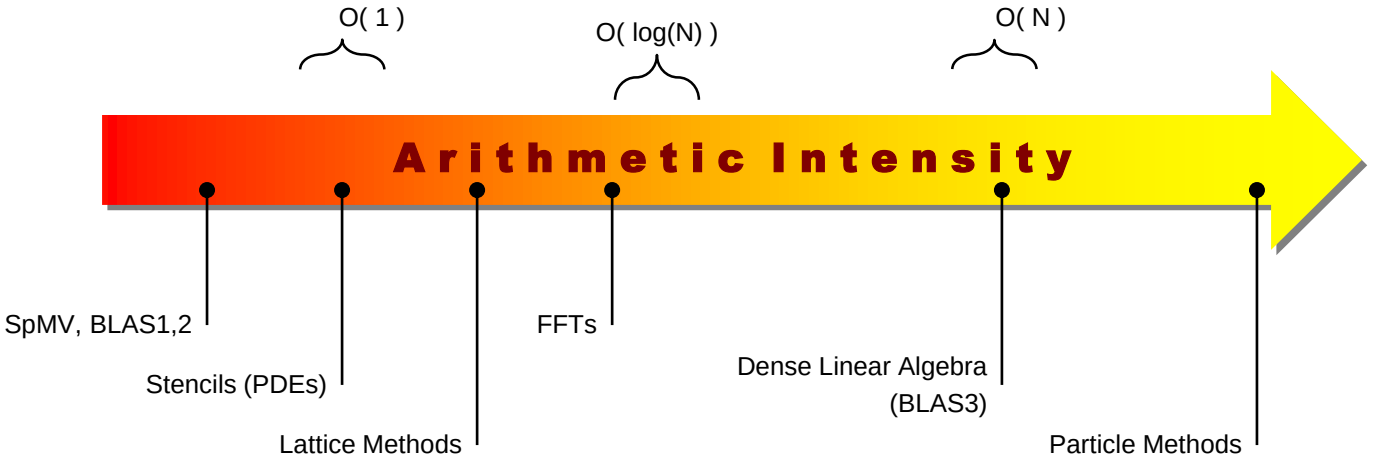
```
for (int y = 0; y < height; y++) {
   for (int x = 0; x < width; x++) {
    Pixel pixel = RGB[y][x];
    gray[y][x] =
          0.30 * pixel.R
        + 0.59 * pixel.G
        + 0.11 * pixel.B;
   }
}
```

- 2 additions, 3 multiplies = 5 operations
- 3 reads, 1 write = 4 memory accesses
- AI = 5/4 = 1.25

Compute or memory intensive?



Applications AI



Operational intensity

- **The number of operations per byte of DRAM traffic**

- **Difference with Arithmetic Intensity**
  - **Operations, not just arithmetic**
  - **Caches**
    - **“After they have been filtered by the cache hierarchy”**
    - **Not between processor and cache**
    - **But between cache and DRAM memory**

Attainable performance

- **Attainable GFlops/sec**

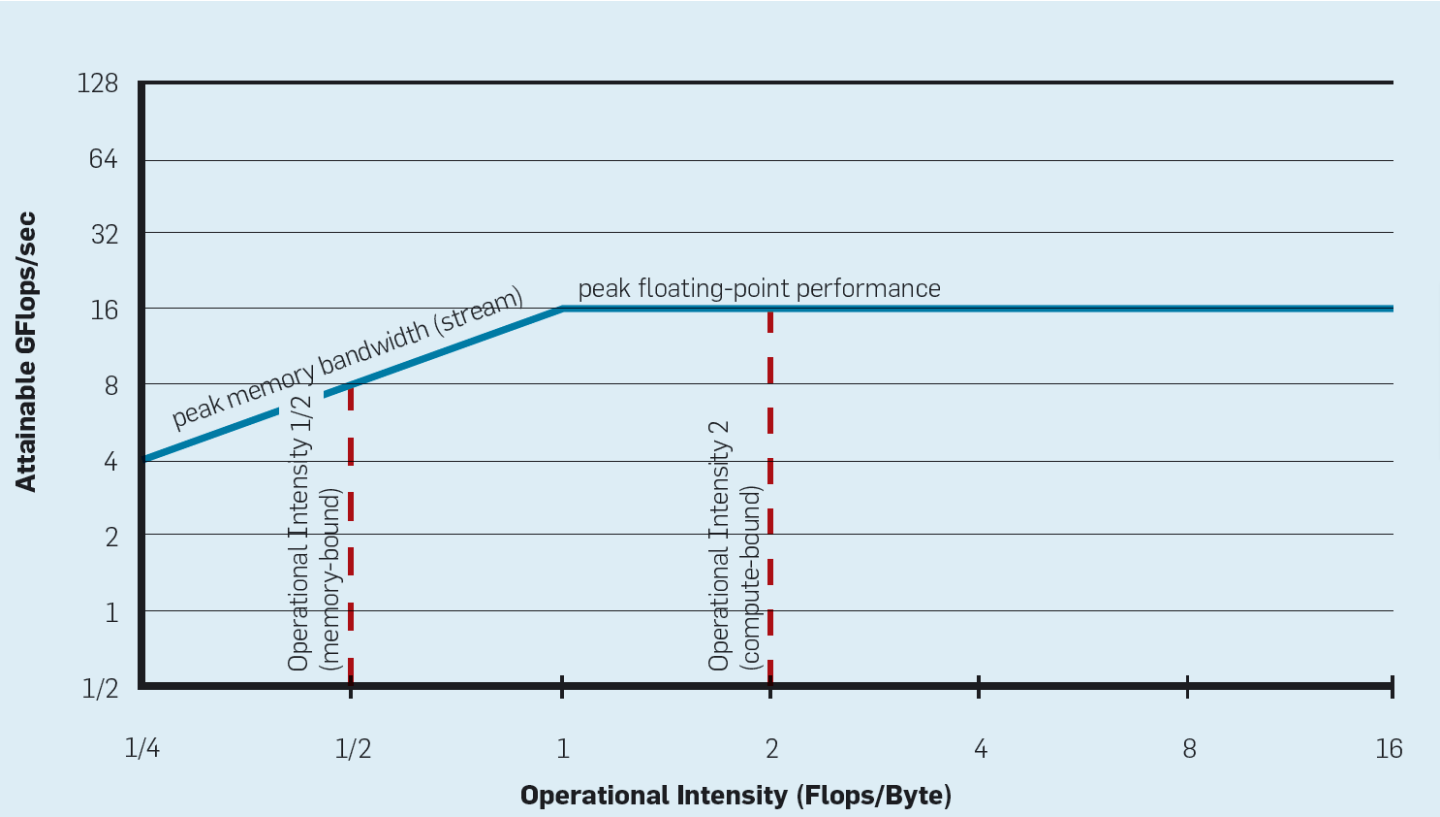
  = *min* (Peak Floating-Point Performance,

  Peak Memory Bandwidth * Operational Intensity)

Samuel Williams, Andrew Waterman, David Patterson

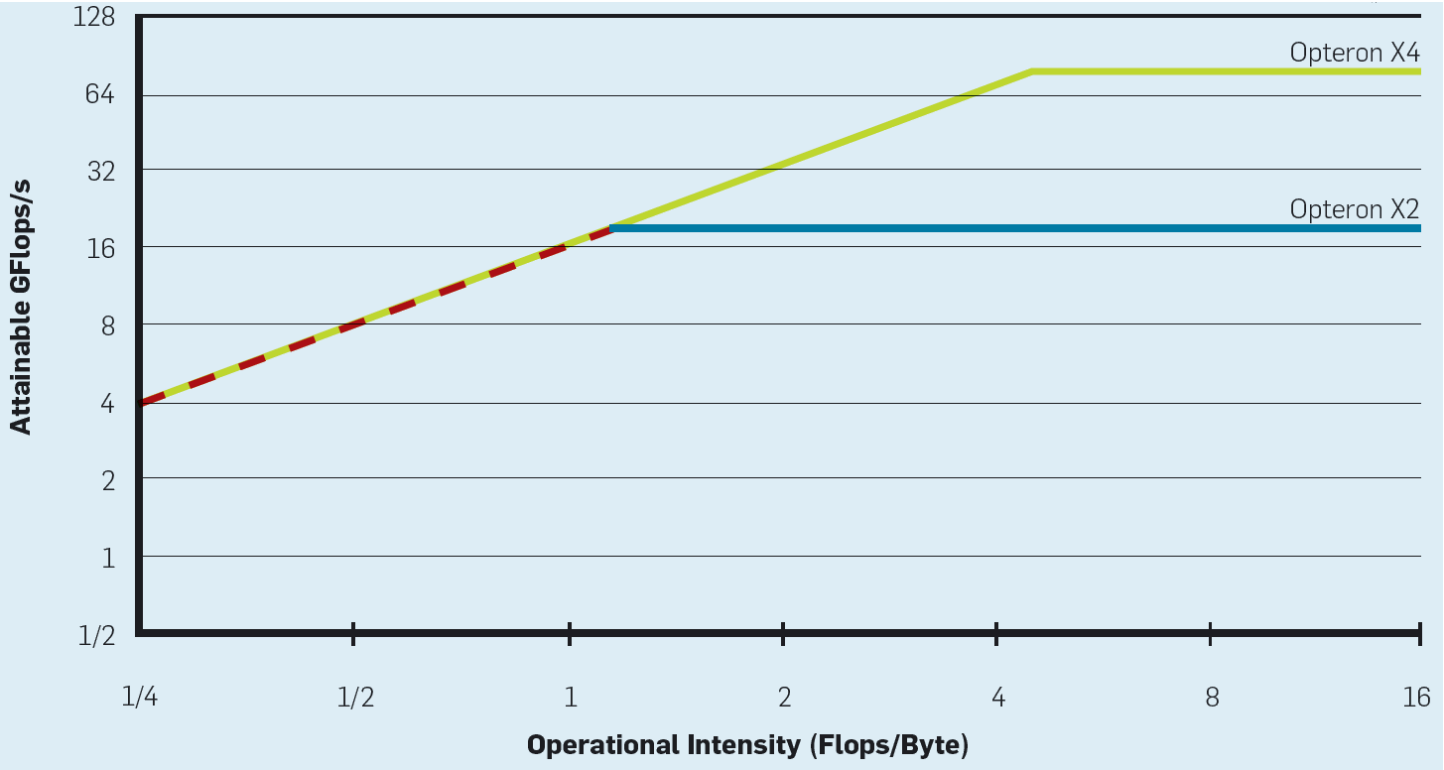
"Roofline: an insightful visual performance model for multicore architectures"

The Roofline model



AMD Opteron X2 (two cores): 17.6 gflops, 15 GB/s, ops/byte = 1.17

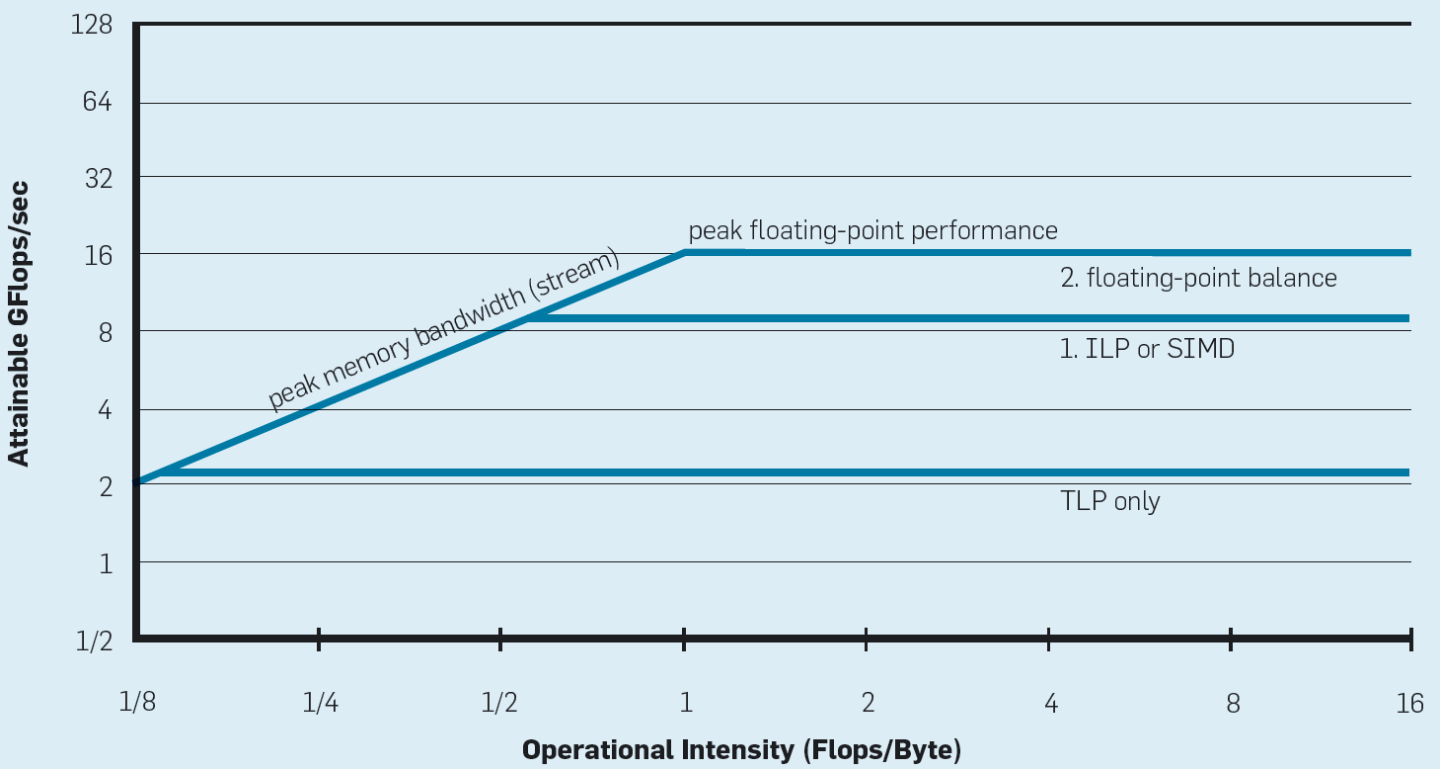
Roofline: comparing architectures



AMD Opteron X2: 17.6 gflops, 15 GB/s, ops/byte = 1.17

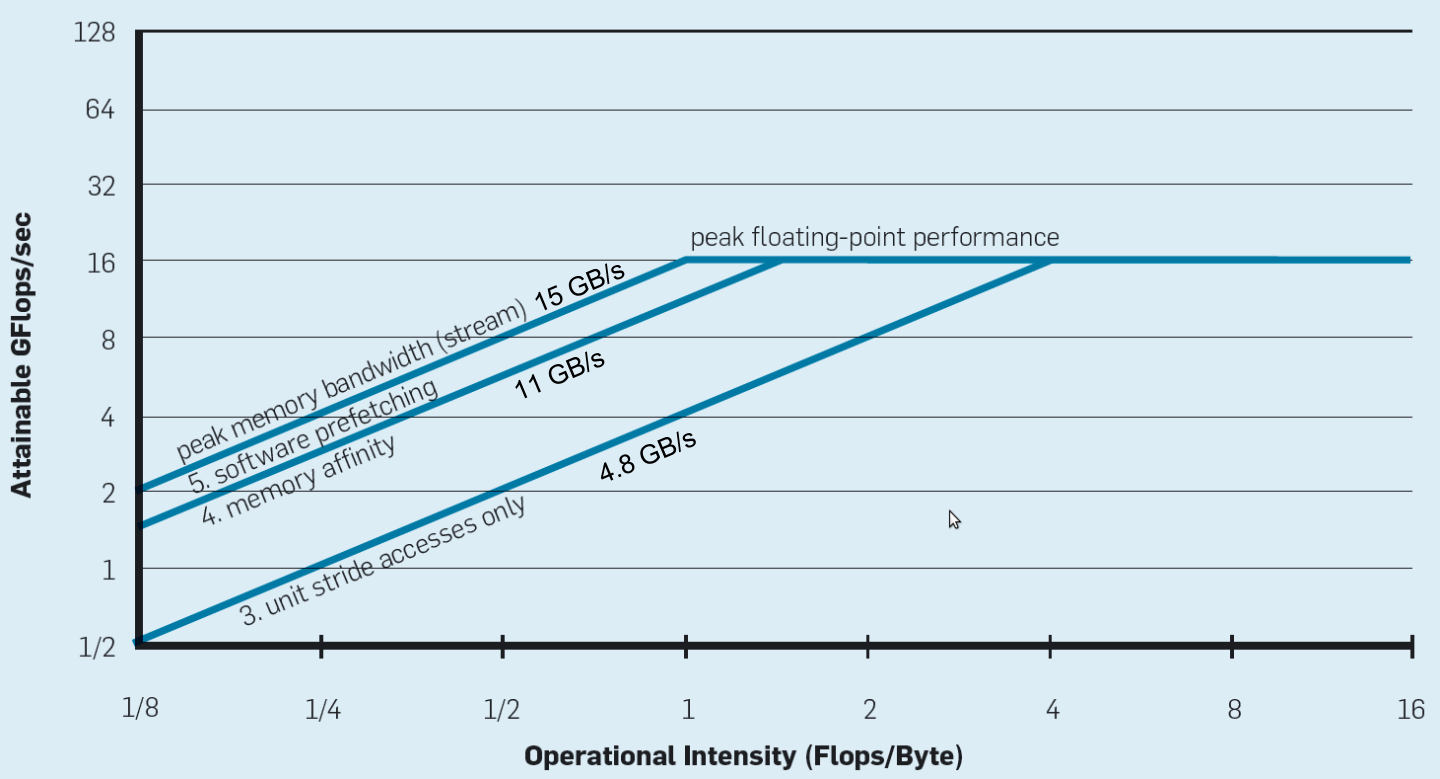
AMD Opteron X4: 73.6 gflops, 15 GB/s, ops/byte = 4.9

Roofline: computational ceilings



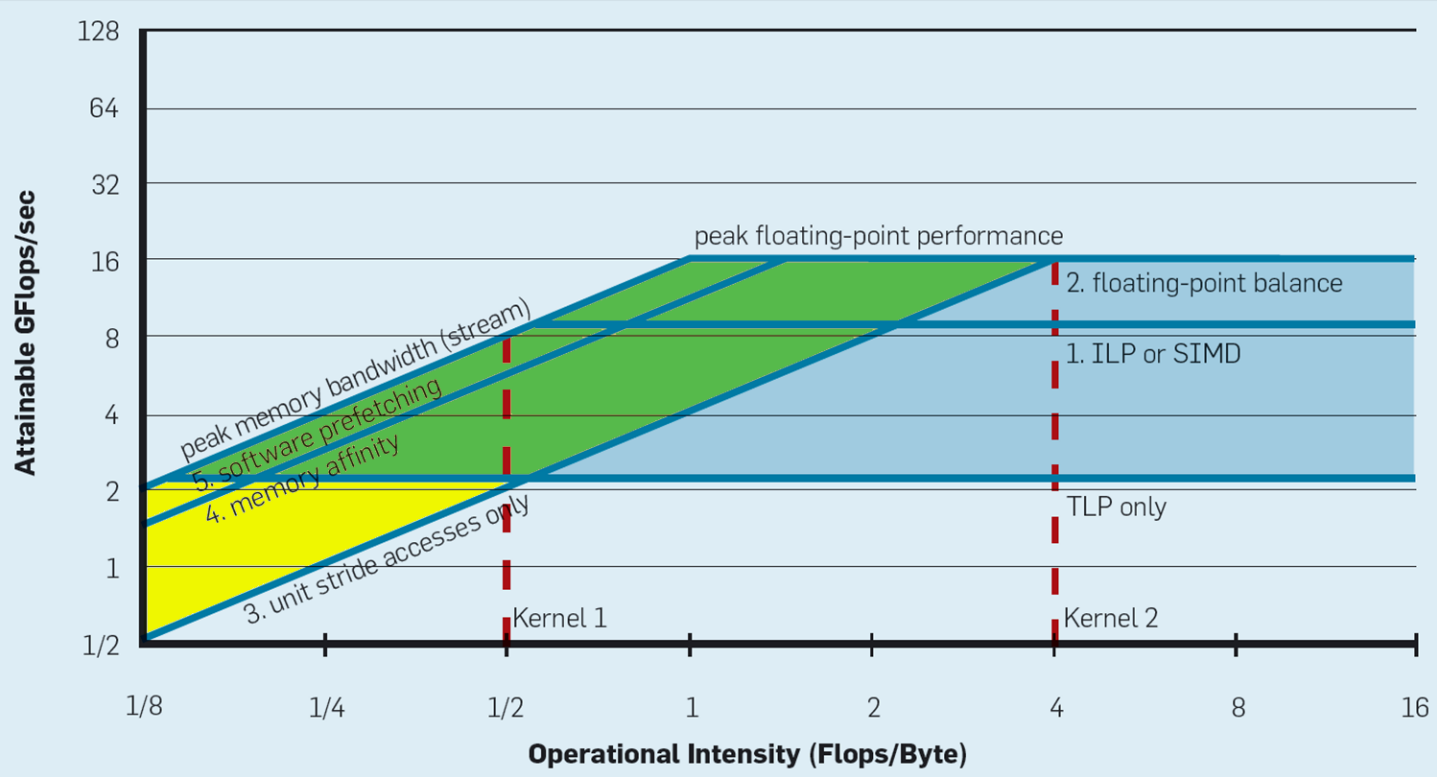
AMD Opteron X2 (two cores): 17.6 gflops, 15 GB/s, ops/byte = 1.17

Roofline: bandwidth ceilings



AMD Opteron X2 (two cores): 17.6 gflops, 15 GB/s, ops/byte = 1.17

Roofline: optimization regions



Use the Roofline model

- **Determine what to do first to gain performance**
  - **Increase memory streaming rate**
  - **Apply in-core optimizations**
  - **Increase arithmetic intensity**

Samuel Williams, Andrew Waterman, David Patterson
"Roofline: an insightful visual performance model for multicore architectures"

LOFAR use case

Rob V. van Nieuwpoort and John W. Romein:
Correlating Radio Astronomy Signals with Many-Core Hardware

John W. Romein:
A Comparison of Accelerator Architectures for Radio-Astronomical Signal-Processing Algorithms

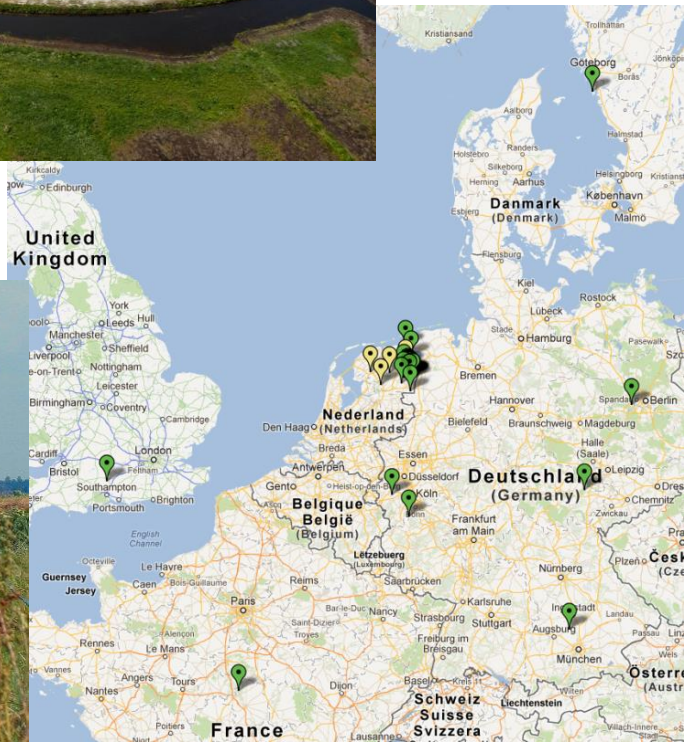
Software radio telescopes

- **Software radio telescopes**
  - **We cannot keep on building larger dishes**
  - **Replace dishes with thousands of small antennas**
  - **Combine signals in software**

LOFAR

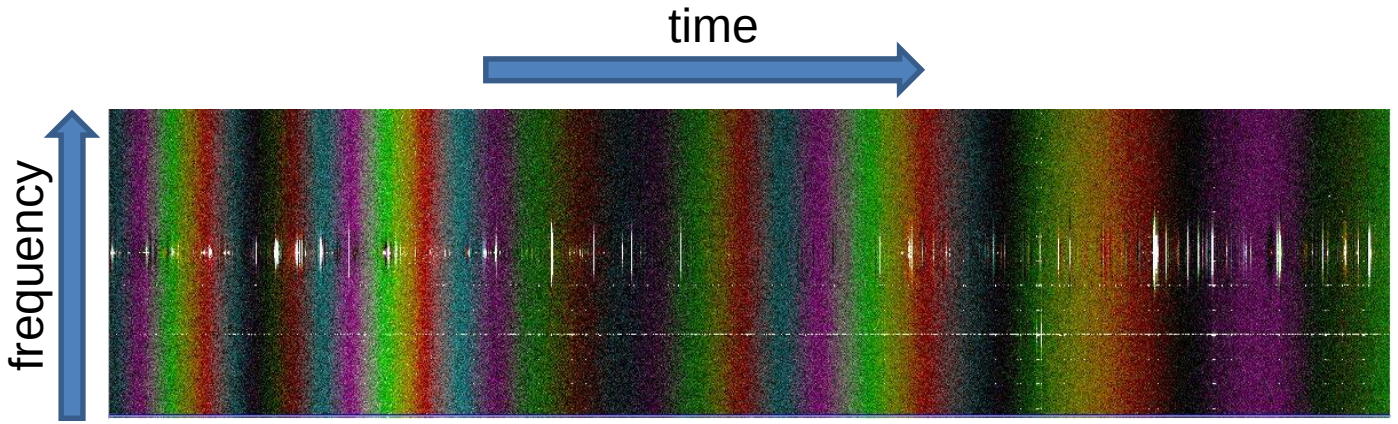
- Largest telescope in the world
- 100.000 omni-directional antennas
- Hundreds of gbit/s
- Hundreds of teraFLOPS
- 100x more sensitive

- SKA will be 1000x larger
- More data than the internet



Correlator

- **Correlator separates signal from noise**



Correlator algorithm

- For all channels (63488)
- For all combinations of two stations (2080)
- For the combinations of polarizations (4)
  - Complex float sum = 0;
  - For the time integration interval (768 samples)
    - Sum += sample1 * sample2 (complex multiplication)
  - Store sum in memory

Correlator optimization

- Overlap data transfers and computations
- Exploit caches / shared memory
- Loop unrolling
- Tiling
- Scheduling
- SIMD operations
- Assembly
- ...

Correlator: Arithmetic Intensity

Correlator inner loop:

```
for (time = 0; time < integrationTime; time++) {
  sum +=  samples[ch][station1][time][pol1] *
          samples[ch][station2][time][pol2];
}
```

- **complex multiply-add: 8 flops**
- **sample: real + complex float (2 * 4 bytes)**

Correlator: Arithmetic Intensity

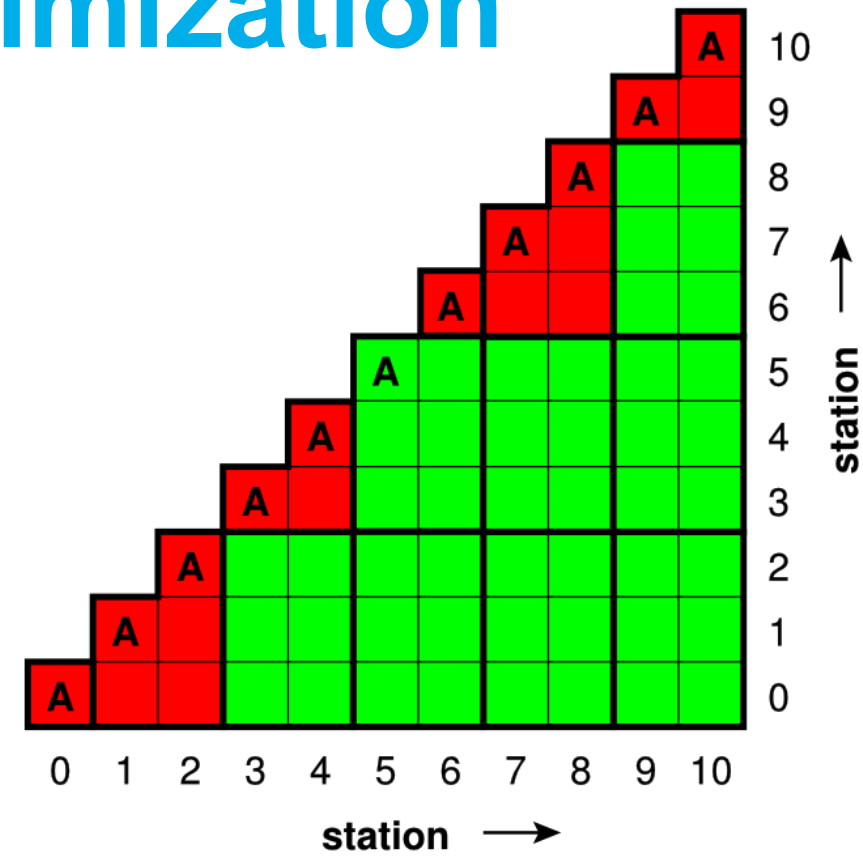
Correlator inner loop:

```
for (time = 0; time < integrationTime; time++) {
  sum +=  samples[ch][station1][time][pol1] *
          samples[ch][station2][time][pol2];
}
```

- **complex multiply-add: 8 flops**
- **sample: real + complex float (2 * 4 bytes)**
- **AI: 8 FLOPS, 2 samples: 8 / 16 = 0.5**

Correlator AI optimization

- Combine polarizations
  - complex multiply-add: 8 flops
    - 2 polarizations: X, Y
    - calculate XX, XY, YX, YY
    - 32 flops per square
  - Complex XY-sample: 16 bytes (x2)
  - 1 flop/byte
- Tiling
  - 1 flop/byte ➔ 2.4 flops/byte
  - but, we need registers
  - 1x1 already needs 16!

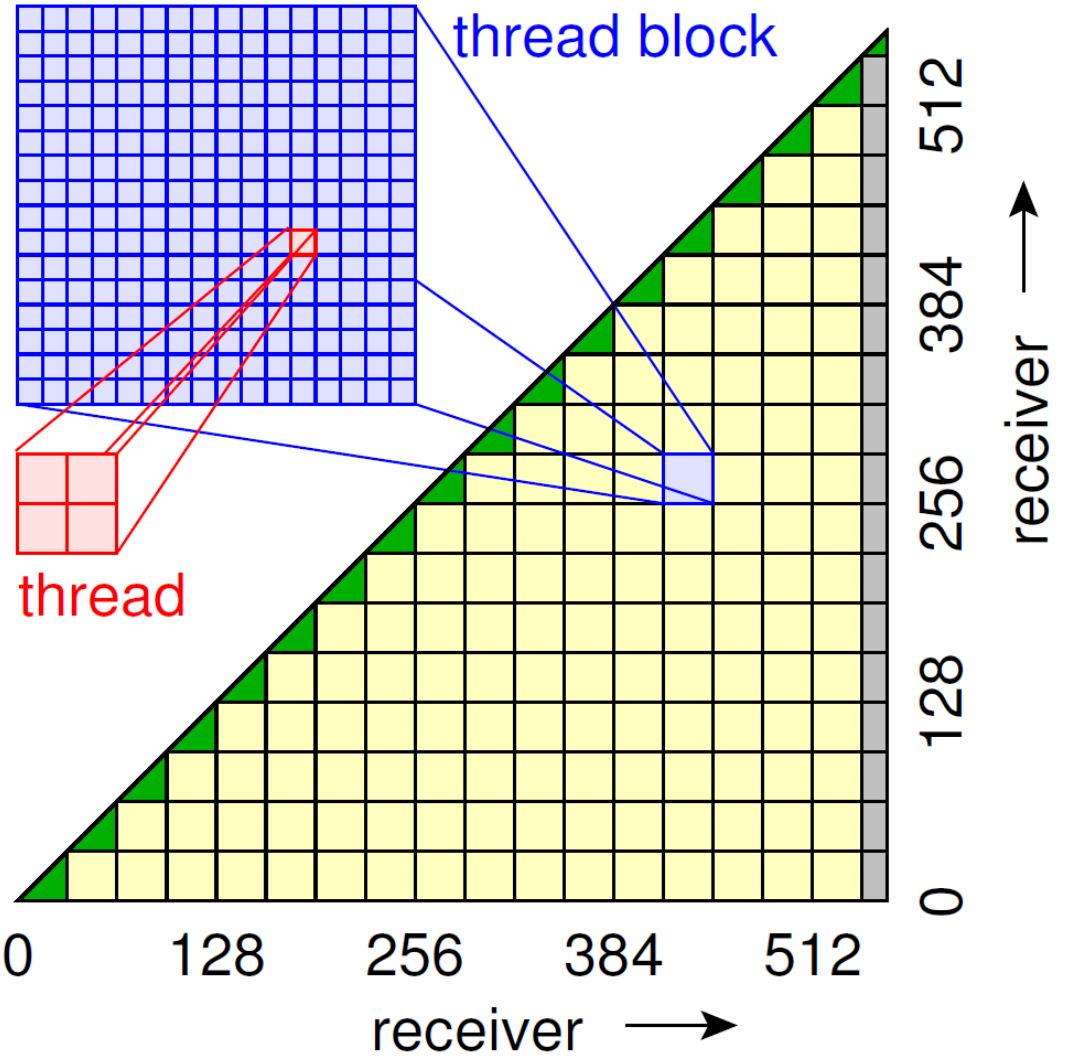


Tuning the tile size

tile size	floating point operations	memory loads (bytes)	arithmetic intensity	Minimum # Registers (floats)
1 x 1	32	32	1.00	16
2 x 1	64	48	1.33	24
2 x 2	128	64	2.00	44
3 x 2	192	80	2.40	60
3 x 3	288	96	3.00	88
4 x 3	384	112	3.43	112
4 x 4	512	128	4.00	148

Using shared memory

- On GPUs:
- Load 32x32 tiles in shared memory
- 2x2 tiles per thread



Picture courtesy John Romein, ASTRON

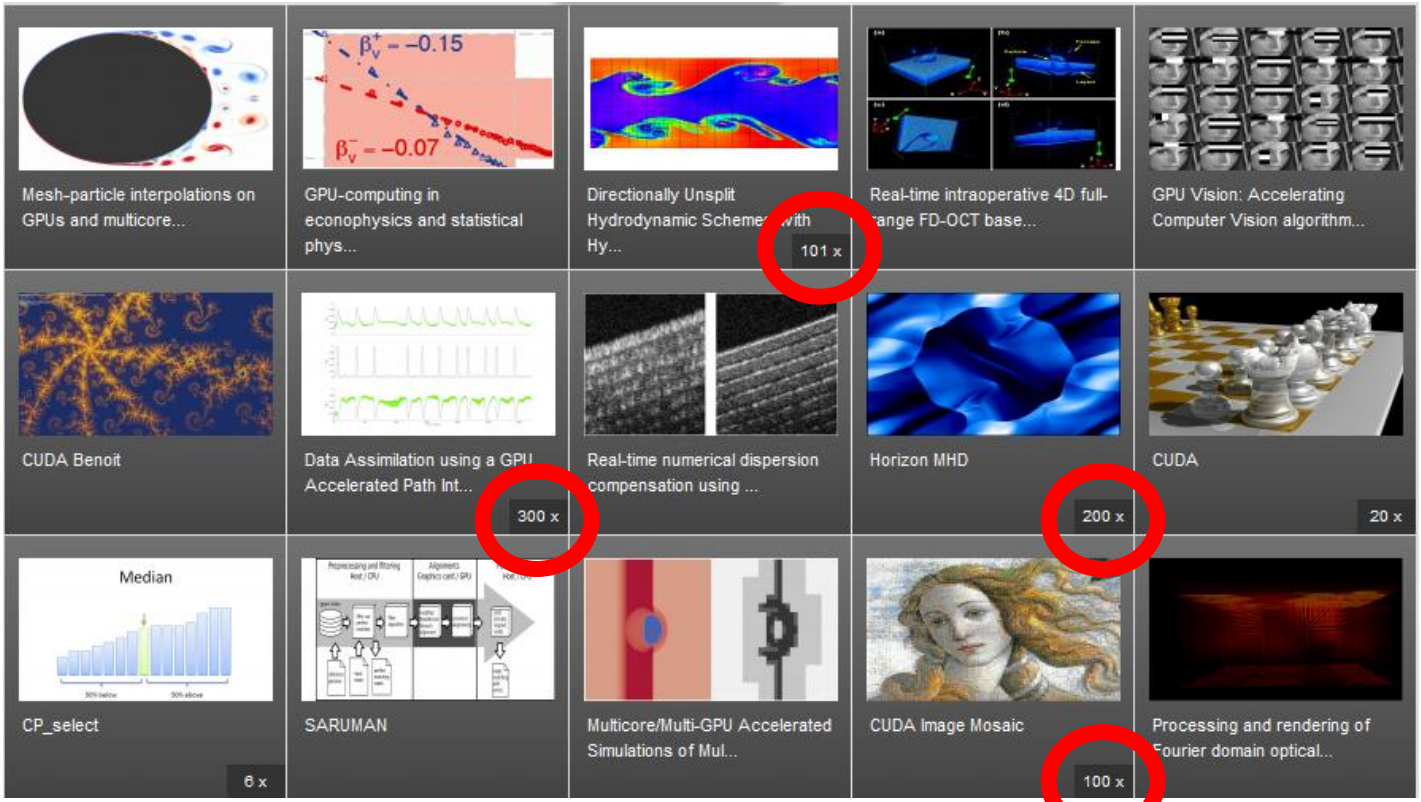
Implementation strategies

- CPU
- Partition frequencies over the cores
  - Independent
  - Multithreading
- Each core computes its own correlation triangle
- Use tiling
- Vectorize with SSE
  - Unroll time loop; compute 4 time steps in parallel

- GPU
- Partition frequencies over the streaming multiprocessors
  - Independent
- Double buffering between GPU and host
- Exploit data reuse as much as possible
  - Each streaming multiprocessor computes a correlation triangle
    - Threads/cores within a SM cooperate on a single triangle
  - Load samples into shared memory
  - Use tiling

LOFAR use case Evaluation

How to cheat with speedups



How can this be?

Core I7 CPU has 154 GFLOPs

NVIDIA GTX 580 GPU has 1581 GFLOPs (10.3 X more)

How to cheat with speedups

- Heavily Optimize GPU version
  - Coalescing, Shared memory
  - Tiling, Loop unrolling
  - …
- Do not optimize CPU version
  - 1 core only
  - No SSE
  - Cache unaware
  - No loop unrolling and tiling, …
- Result: very high speedups!
- Exception: kernels that do interpolations (texturing hardware)
- Solution
  - Optimize CPU version
  - Use efficiencies: % of peak performance, Roofline

Theoretical performance bounds

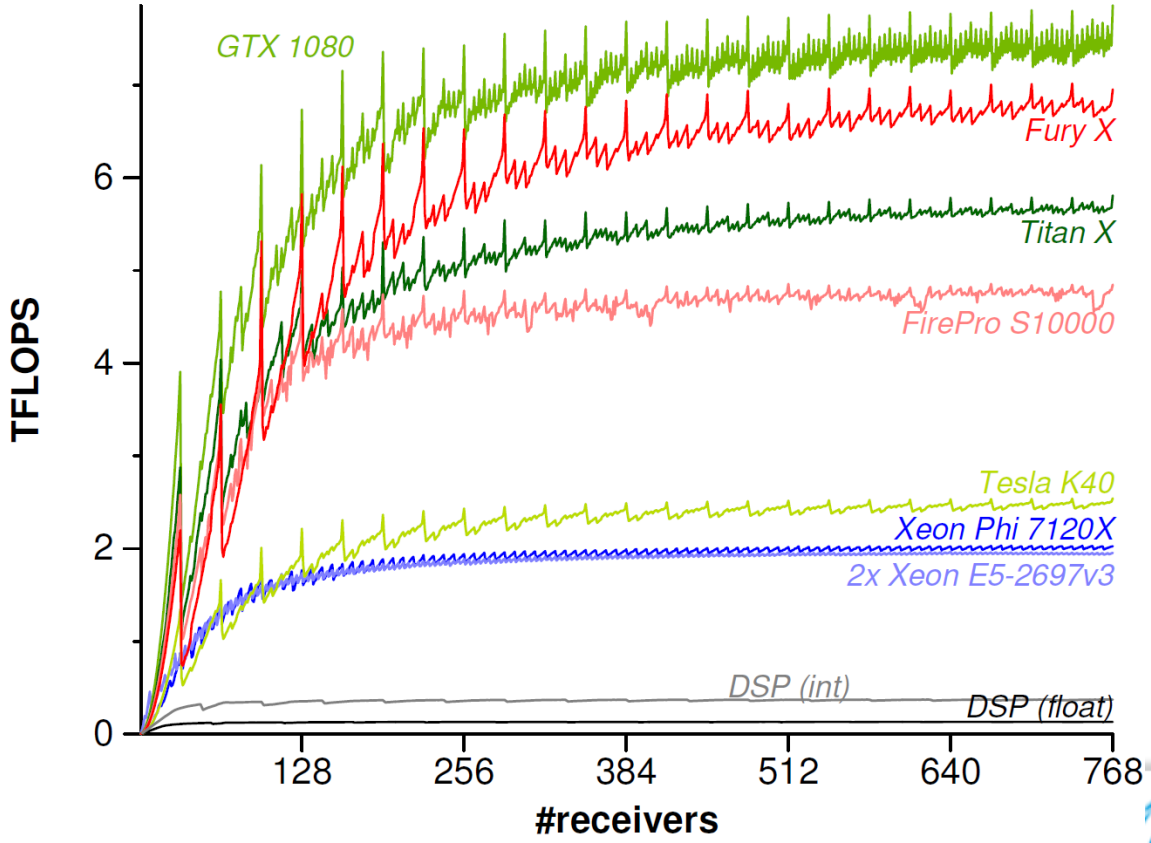
- Distinguish between global and local (host vs device)
  - Local AI = 1 .. 4
  - Depends on tile size, and # registers
- Max performance = AI * memory bandwidth
- NVIDIA GTX 1080
  - Peak performance: 9.22 Tflops
  - Max device memory bandwidth: 320 GB/s
  - Needs AI of 28.8 for full utilization
  - Max performance 2x2 tiles: 2.0 * 320 = 640 gflops
  - Max performance 32x32 in shared memory: 16 * 320 = 5.1 teraflops
- Can we achieve more than this?

Theoretical performance bounds

- Global AI = #stations + 1 (LOFAR: 65)
- Max performance = AI * memory bandwidth
- Use bandwidth of PCI-e 3.0 (16 GB/s)
- Max performance GPUs, with AI global:
  - NVIDIA: 65 * 16 = 1.0 teraflop
    - need 142 GB/s for peak

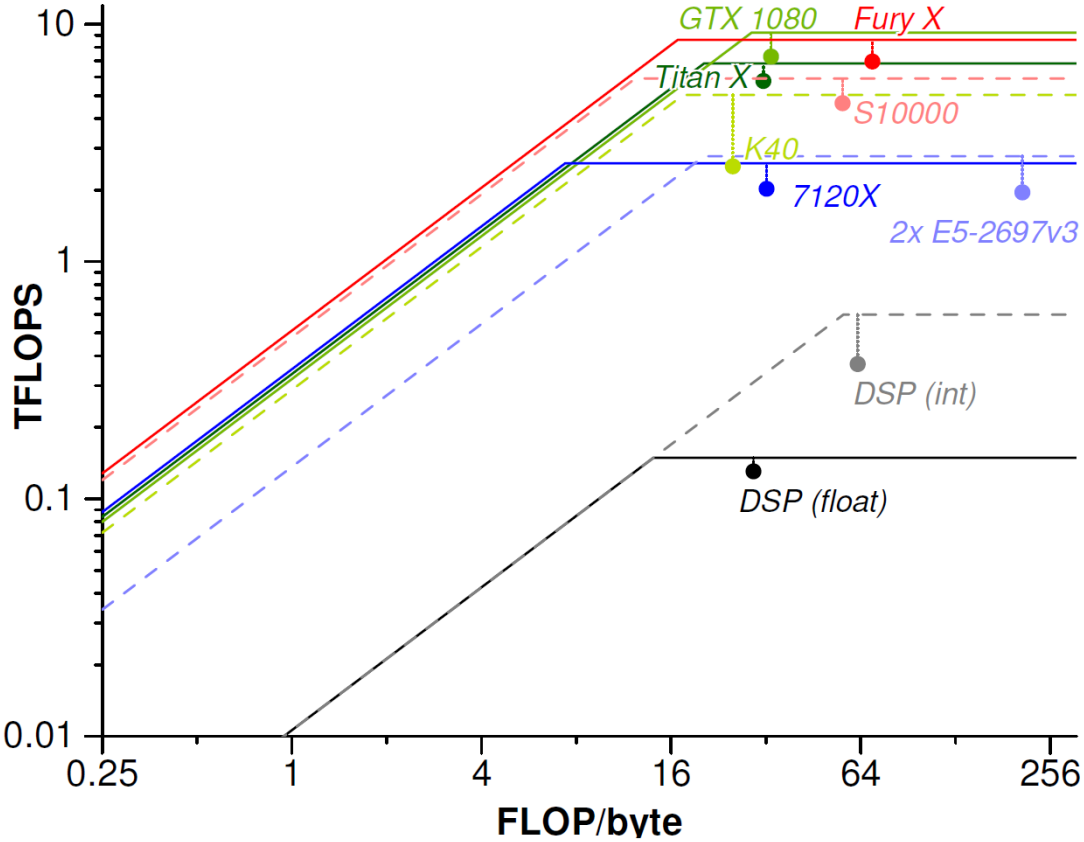
- Can we achieve more than this?

Performance of the correlator



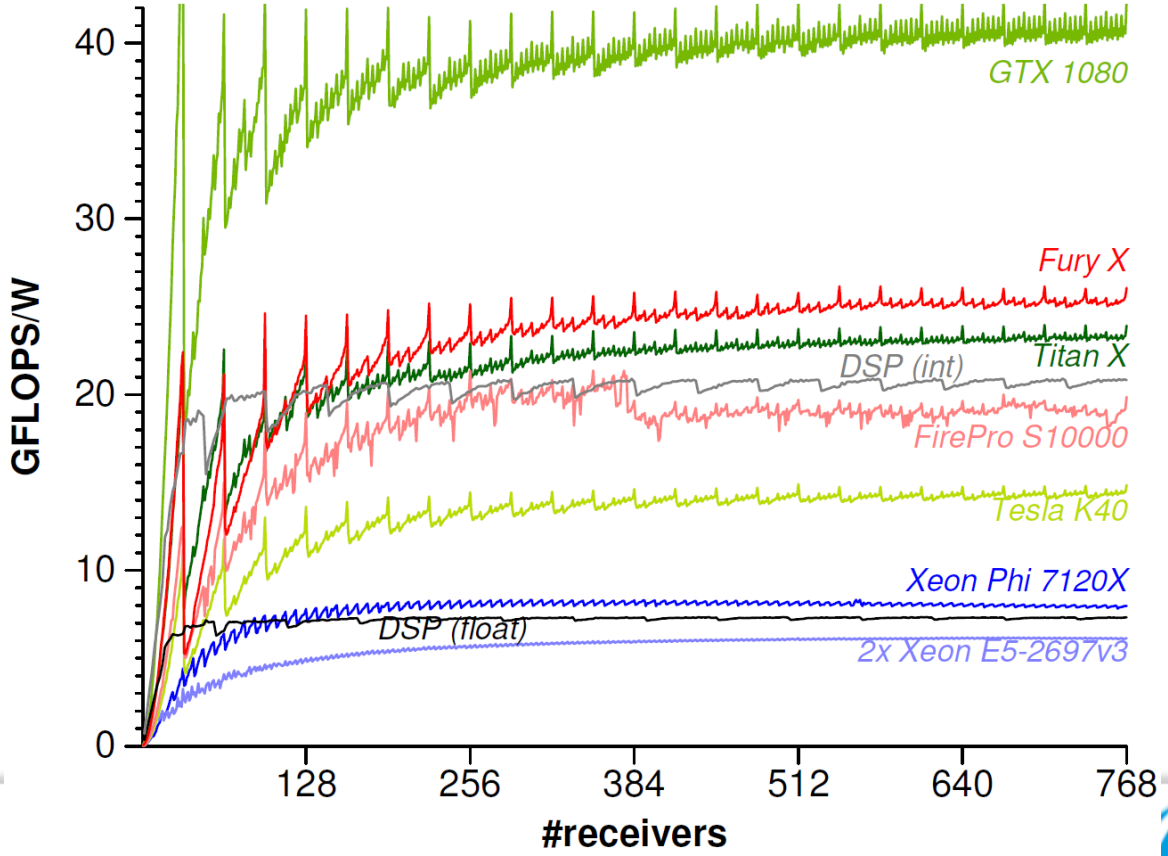
Graph courtesy
John Romein
ASTRON

Roofline of correlator



Graph courtesy
John Romein
ASTRON

Energy efficiency of correlator



Graph courtesy
John Romein
ASTRON

Optimizing Code

- Moving data around is more expensive than computing on it
- Start with a simple algorithm and keep it for readability and correctness checks
- Optimize only when needed
- Focus on the bottlenecks first (use Roofline)
- Auto-tune (automatically explore the parameter space)
  - Different loop orderings
  - Different tile sizes, on multiple levels L3, L2, and L1
  - Different number of threads, thread blocks, vector lengths, etc
  - e.g. using the Kernel Tuner (https://github.com/benvanwerkhoven/kernel_tuner)

backup slides

1D indexing of 2D arrays

- **This is crucial to understand the second hands-on!**

- **Idea is to mimic a 2D data structure on what is actually a 1D array in memory**
- **Row-major accessing all elements in an array:**

```
for (i=0; i<nrows; i++)
    for (j=0; j<ncols; j++)
        matrix[i][j]          //2D array

for (i=0; i<nrows; i++)
    for (j=0; j<ncols; j++)
        matrix[i*ncols+j]     //1D array, 2D access
```