netherlands eScience center

ASCI A24 GPU Programming course

By Alessio Sclocco and Ben van Werkhoven
December 11, 2018

Schedule

- 10:00 – 10:30 Introduction to GPU Computing
- 10:30 – 11:30 High-level intro to CUDA Programming Model (including 1st hands-on)
- 11:30 – 13:00 CUDA memories (includes 2 hands-ons)

- 13:00 – 14:00 Lunch break

- 14:00 – 14:30 Solution to 3rd hands-on
- 14:30 – 15:00 CUDA Program Execution
- 15:00 – 15:30 Experiences with using GPUs for research
- 15:30 – 16:30 GPU kernel optimization techniques

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

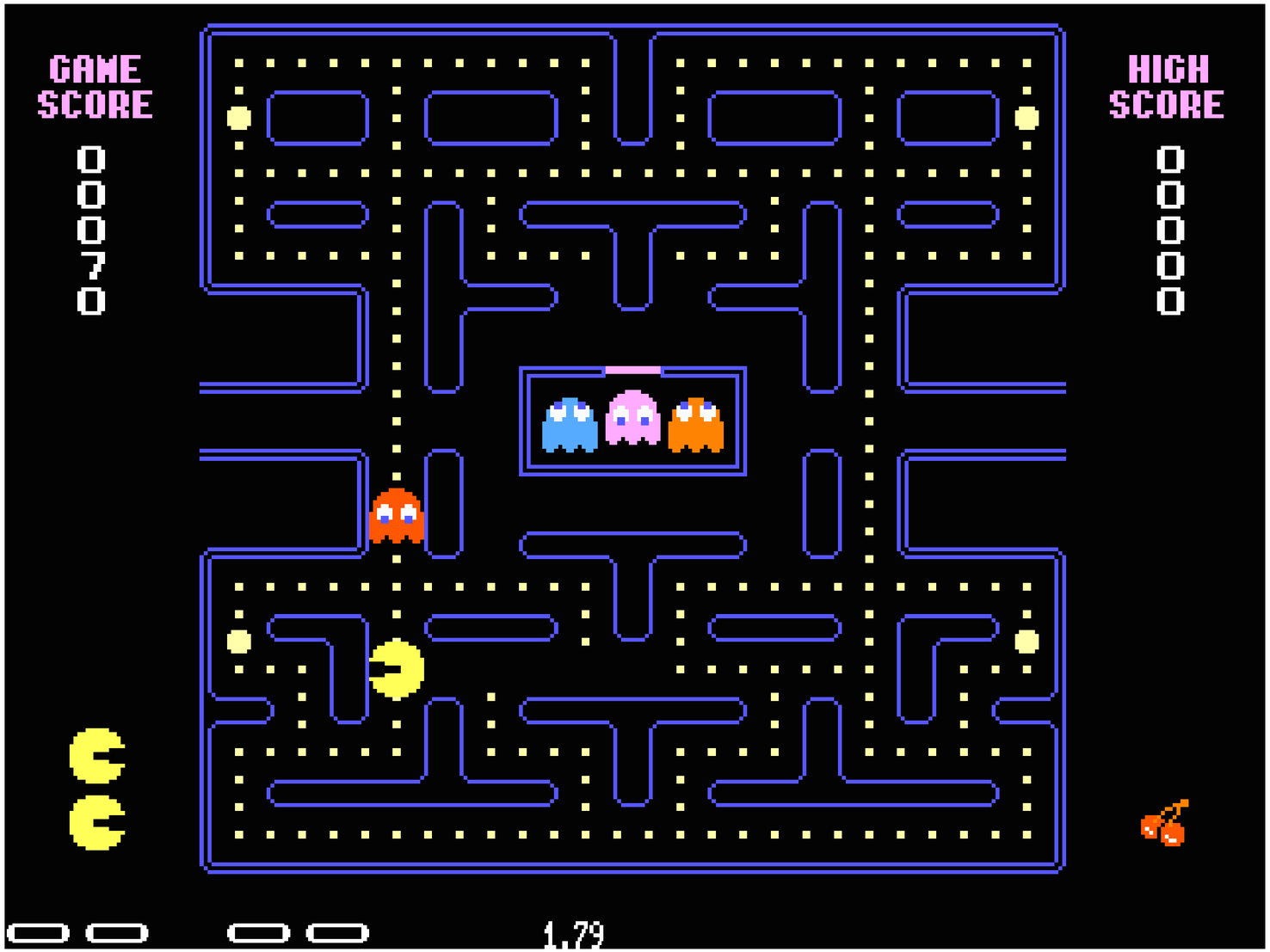
Introduction to GPU Computing

What is a GPU?

- Graphics Processing Unit –
  The computing chip on a graphics card

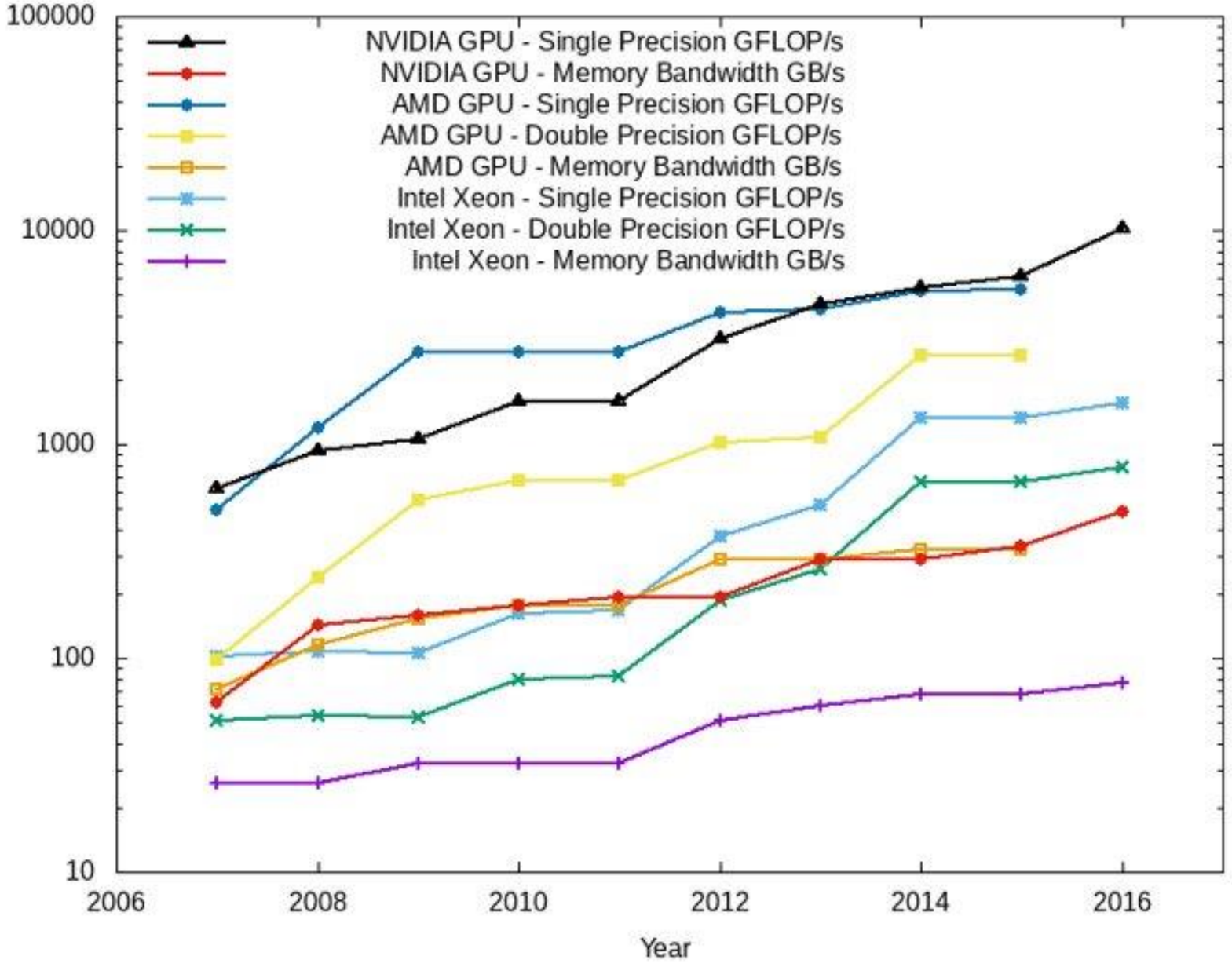
- GPGPU

Graphics in 1980



Graphics in 2000

Performance Comparison: CPUs vs GPUs

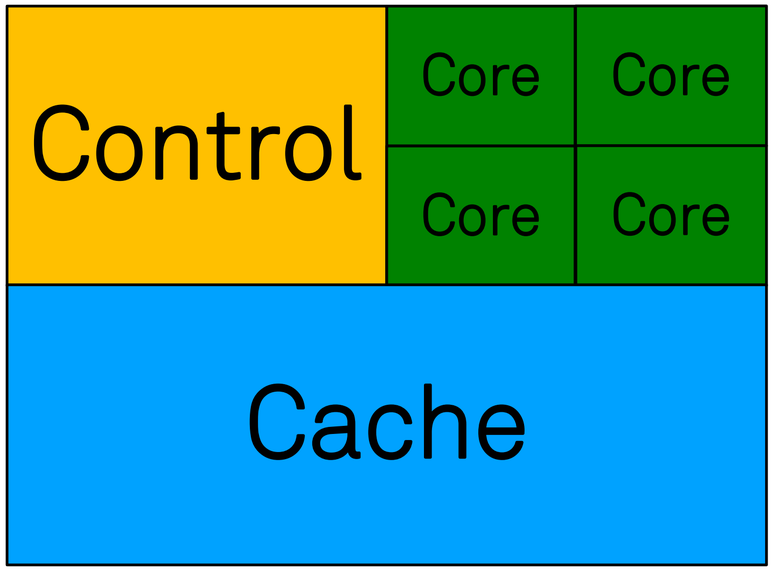


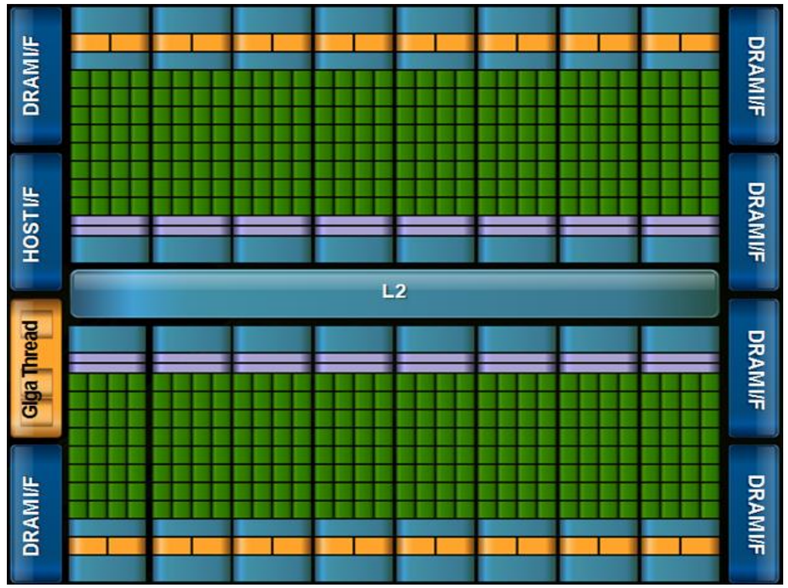
The World's Fastest Supercomputer: Oak Ridge’s Summit

- Number 1 in TOP500 list (November 2018)
  - 200 PFLOP/s peak
  - 4,608 nodes
  - 13 MW power consumption
  - 9,216 CPUs
    - IBM Power 9
    - 2 CPUs per node
  - 27,648 GPUs
    - NVIDIA Volta
    - 6GPUs per node
  - 2,397,824 cores

Design: CPUs vs GPUs

- 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





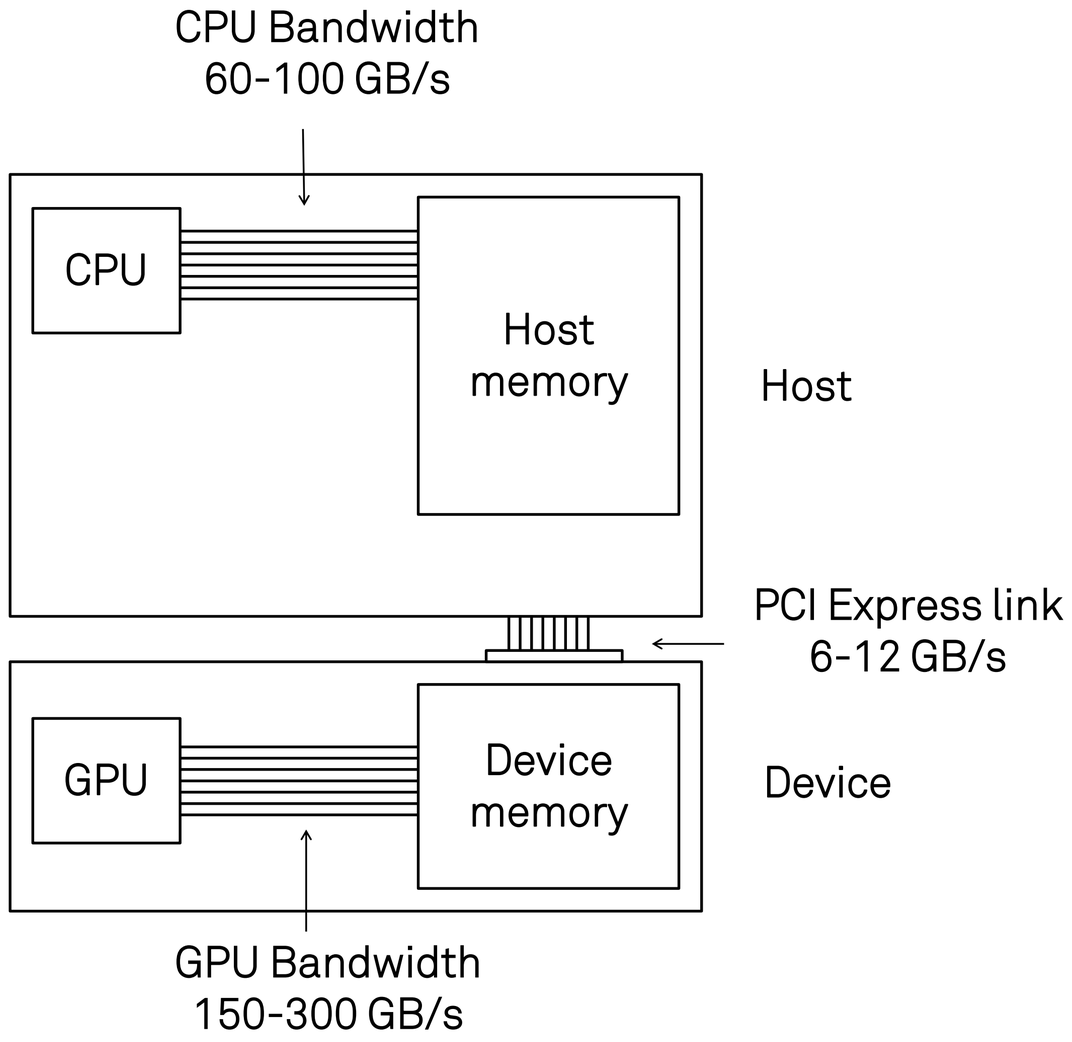
Why is GPU Programming different?

The computer architecture is very different:

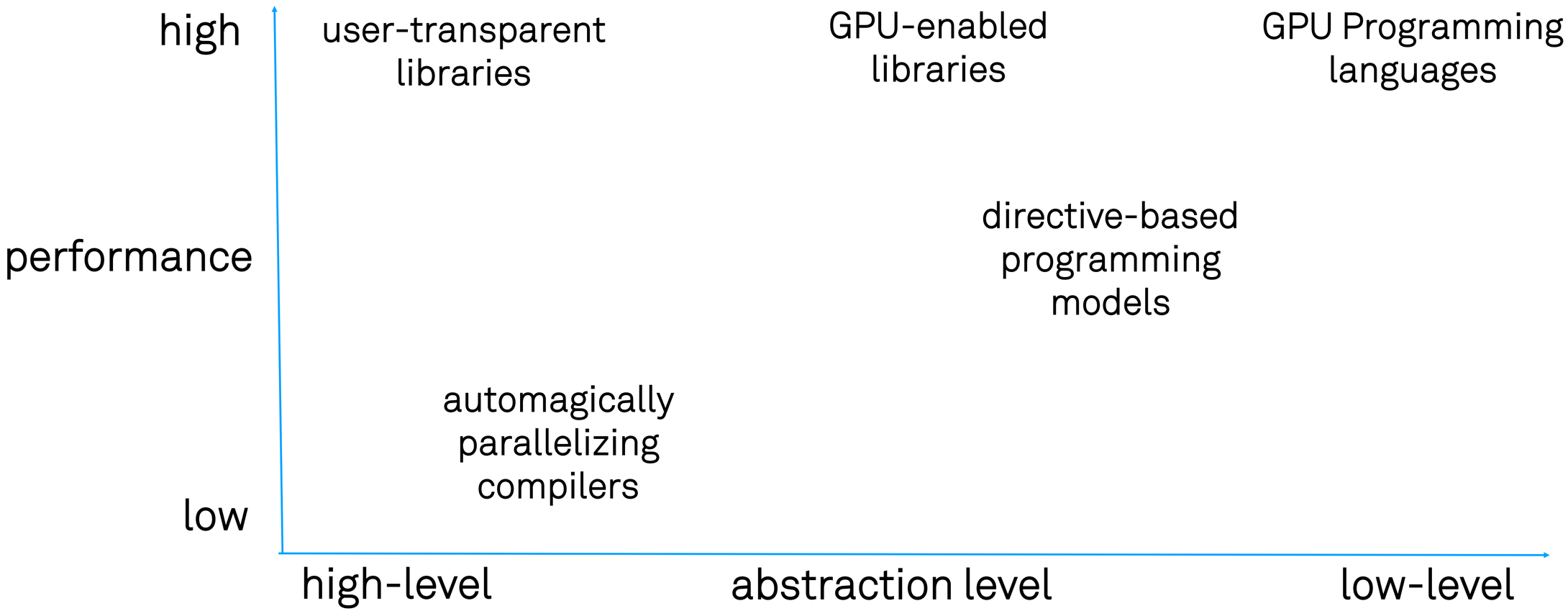
- Algorithms need to be parallelized and mapped to the hardware
- Requires software to be rewritten in specialized programming language
- Optimizing for compute performance requires knowledge about hardware

GPUs are on separate devices:

- Have to deal with separate memory space, limited bandwidth between host and device memory



GPU Programming techniques



A high-level intro to the 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.

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

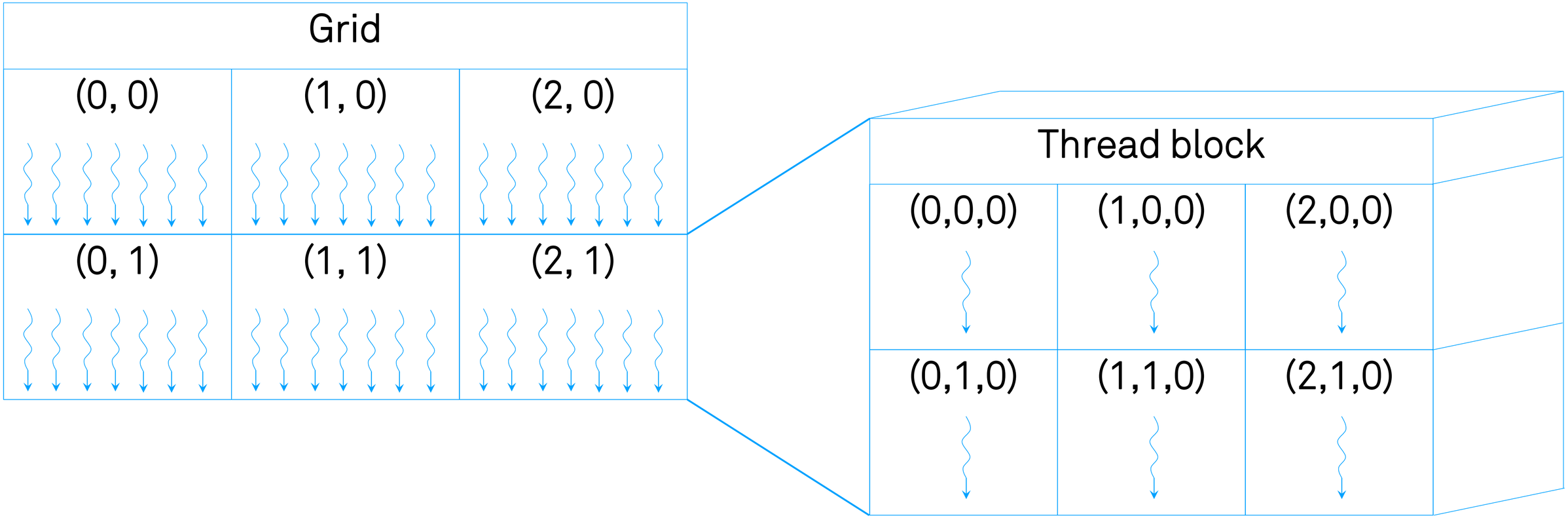
The host part:

- Allocates memory and transfers data between host and device memory, and starts GPU functions

The device part:

- Consists of functions that 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

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



- In the CUDA programming model a thread is the most fine-grained entity that performs computations
- Threads within a kernel all execute the same program
- 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, z);

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

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

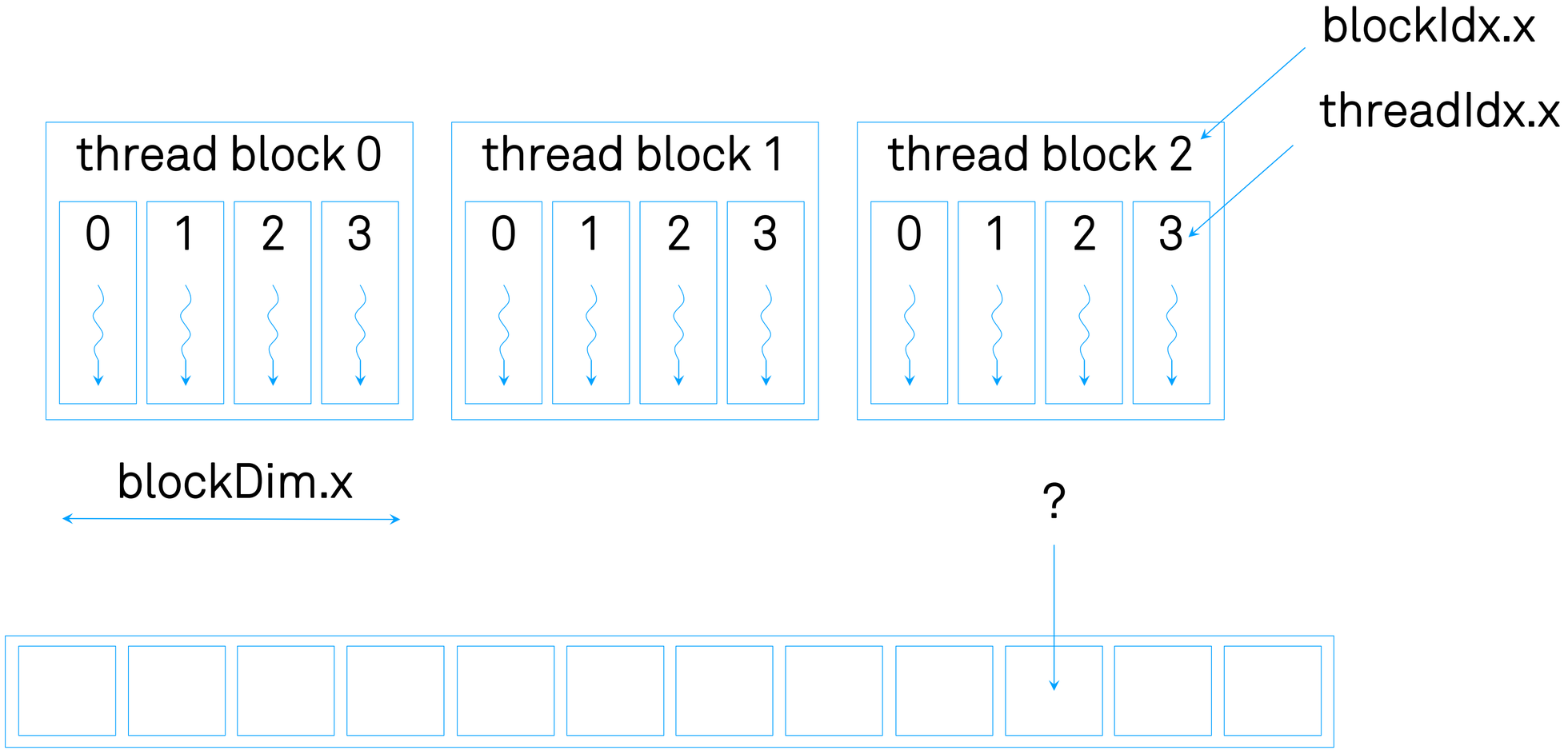
Setup hands-on session

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

- Additional setup if using Python (optional):
  - To install Python:
    - `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` (make sure you've typed `module load cuda80` first)

- 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
    - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#kernels
    - https://documen.tician.de/pycuda/driver.html#pycuda.driver.Function
  - `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
?

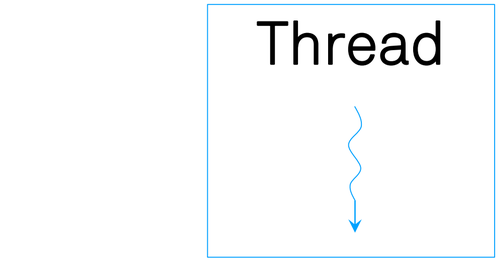
CUDA Memories

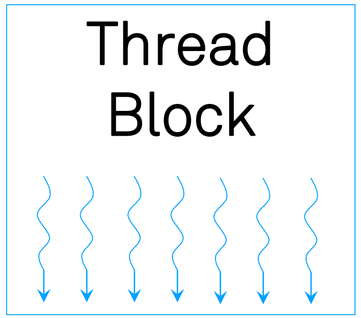
CUDA memory hierarchy

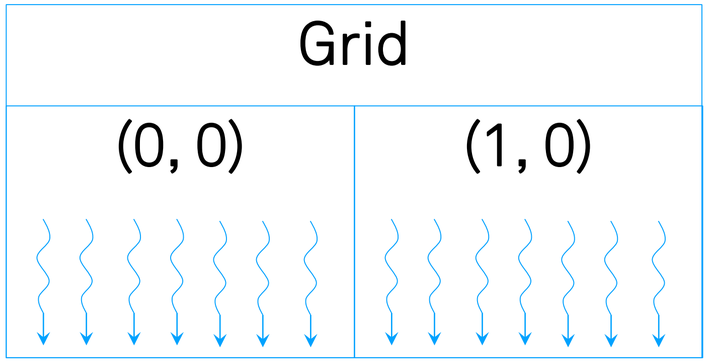
Registers

Shared memory

Global memory
Constant memory







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`**

- 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 (choose a unique 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 (if you change the kernel arguments you have to change the hostcode as well)

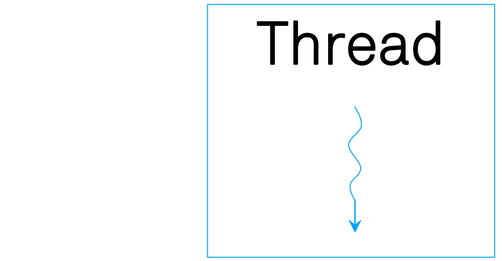
- In C:
  - Add a new `cudaMemcpyToSymbol()` after the `cudaMemcpy()` for vertices and make sure it copies to the right place. Please leave the global memory copy of vertices in place, the reference kernel uses it.
  - See CUDA documentation on `cudaMemcpyToSymbol`
- In Python:
  - Python users can use `memcpy_htod(),` but need to find the symbol to copy to
  - See PyCuda documentation on `get_global`

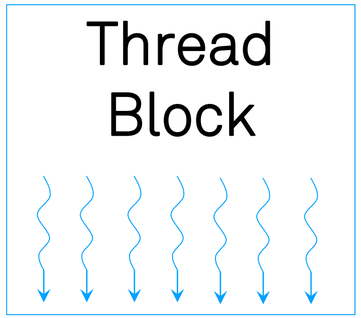
CUDA memory hierarchy

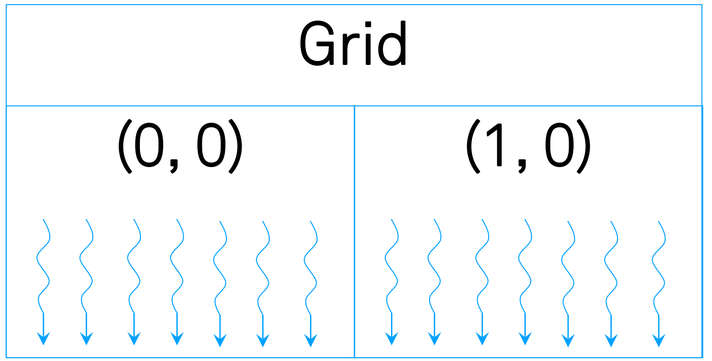
Registers

Shared memory

Global memory
Constant memory







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 a 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

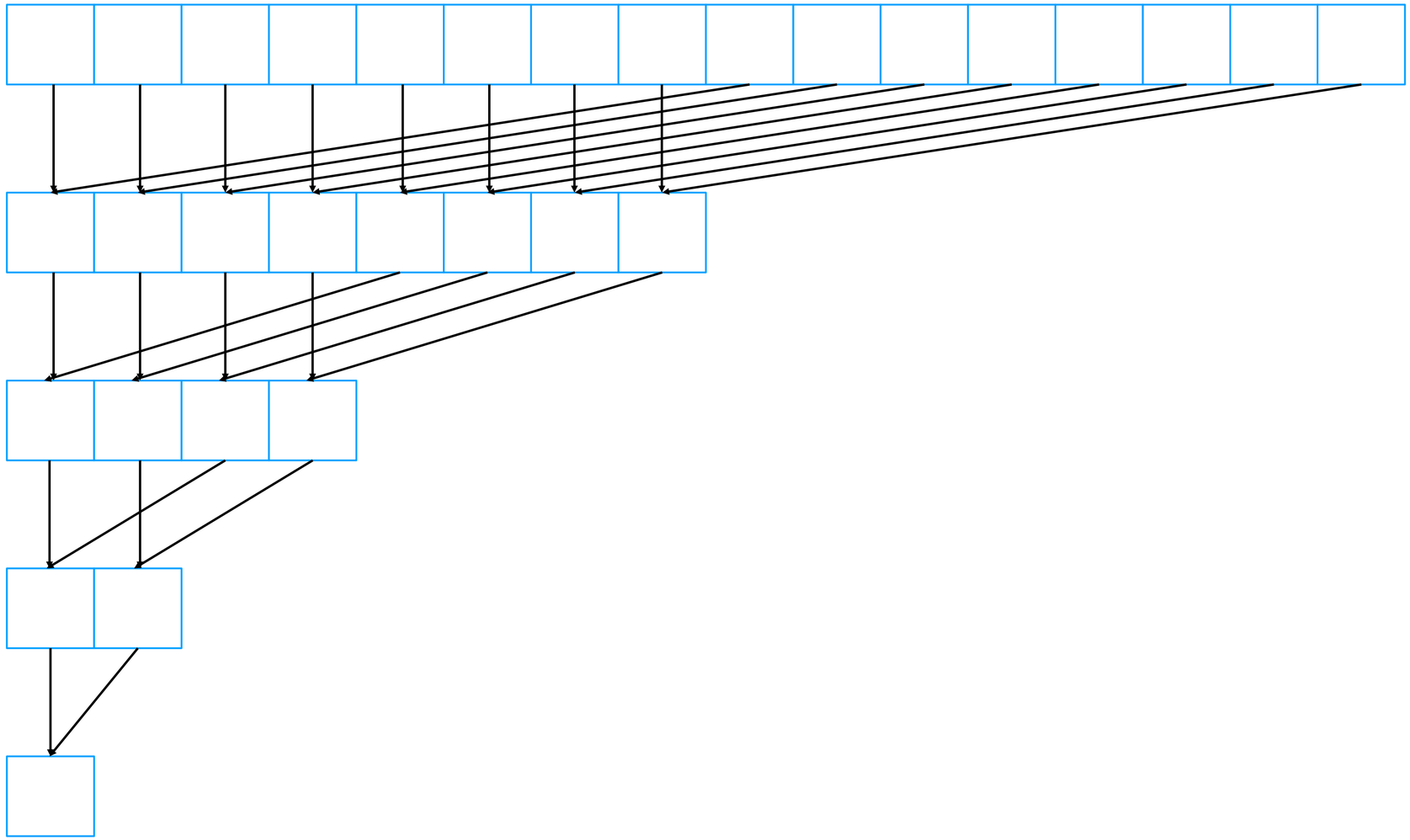
    i = threadIdx.x + blockIdx.y * block_size_y;
    j = threadIdx.y + blockIdx.x * block_size_x;
    if (j < w && i < h) {
       output[j*h+i] = sh_mem[threadIdx.x][threadIdx.y];     //store to global using shared memory
    }
}
```

- 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

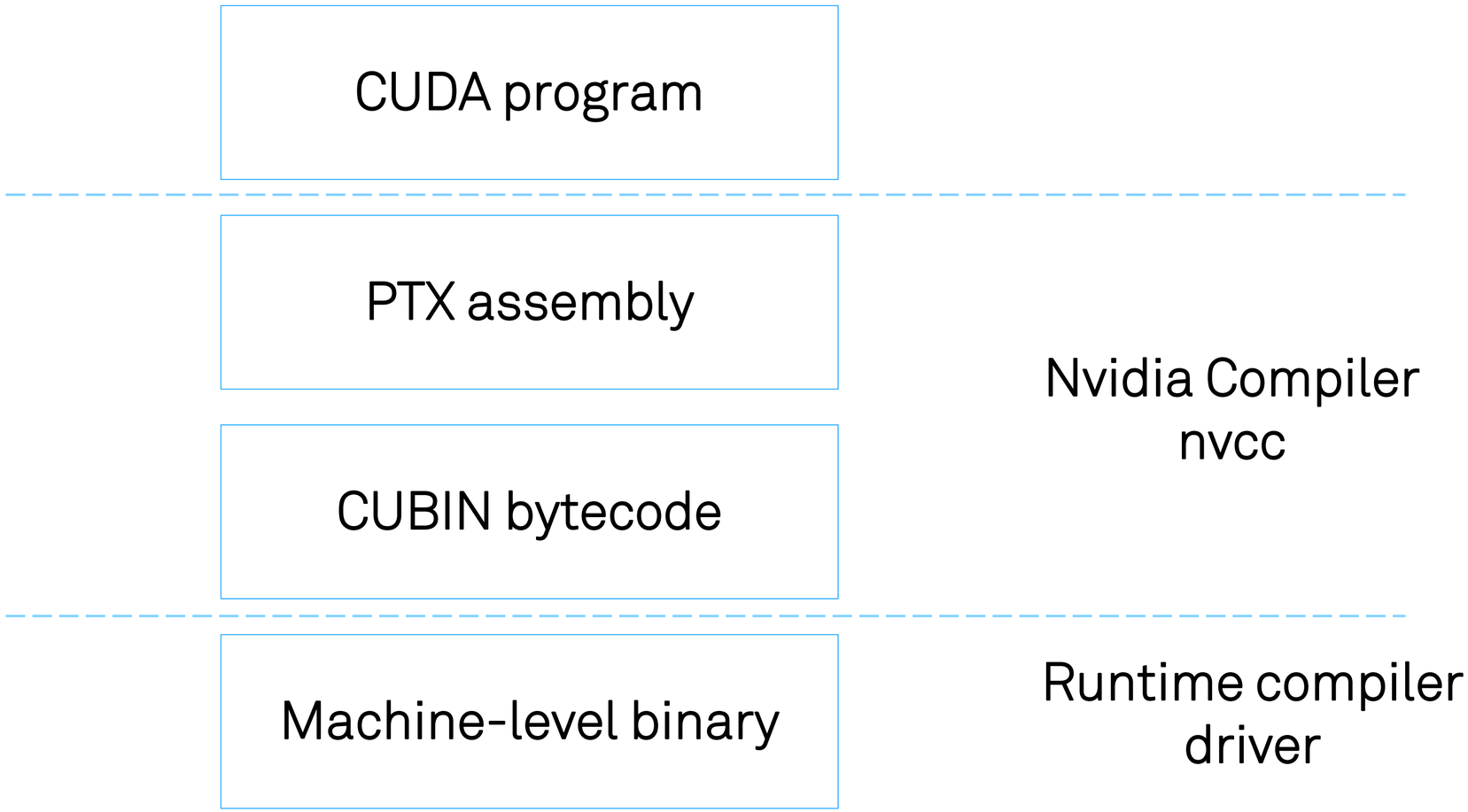
- Hints:
  - The number of thread blocks does not depend on n. All threads from all blocks first iterate (collectively) over the problem size (n) to obtain a per-thread partial sum
  - Within the thread block the per-thread partial sums are to be combined to a per-thread block partial sum
  - Each thread 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 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)
```

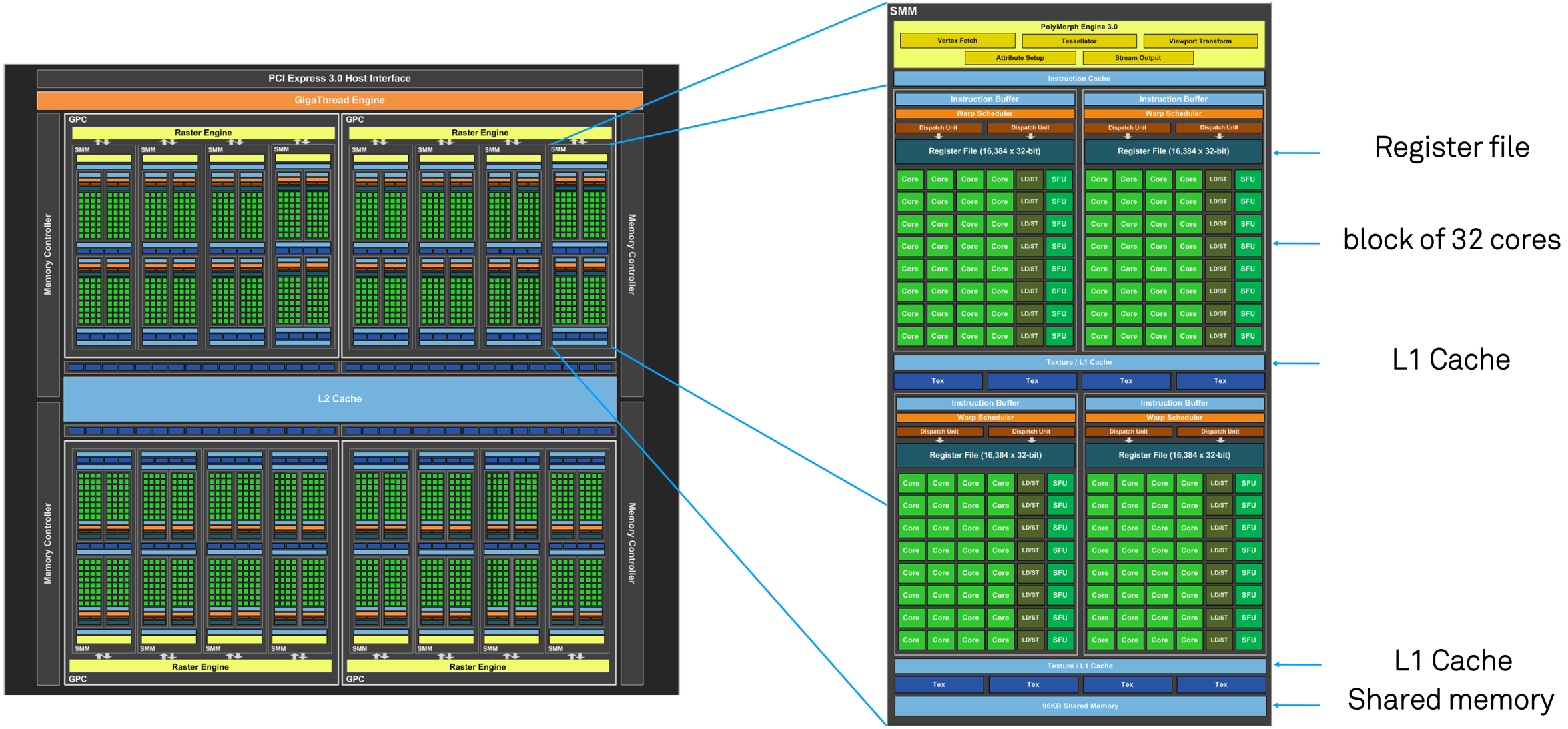
  - This is less efficient, but not always bad.
  - Avoid data-dependent conditional branching if possible

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

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

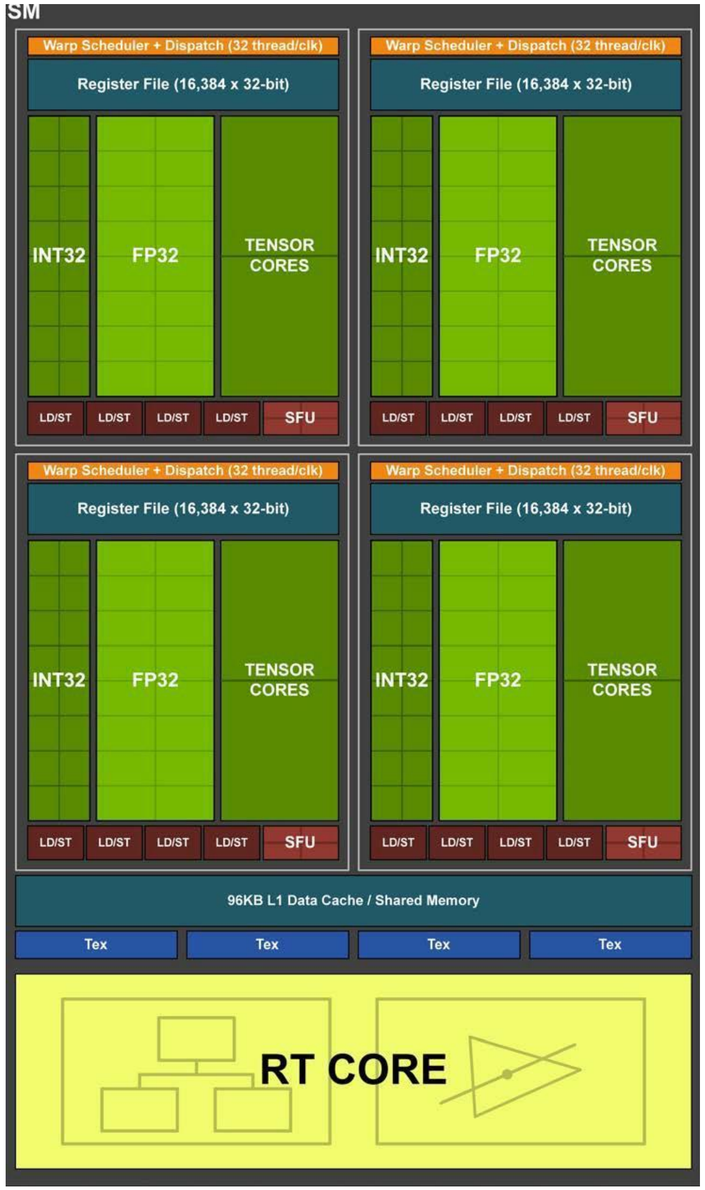
- The Volta architecture replaces predication with a per-thread program counter and call stack. The same performance recommendations apply however.

Maxwell Architecture



Turing architecture

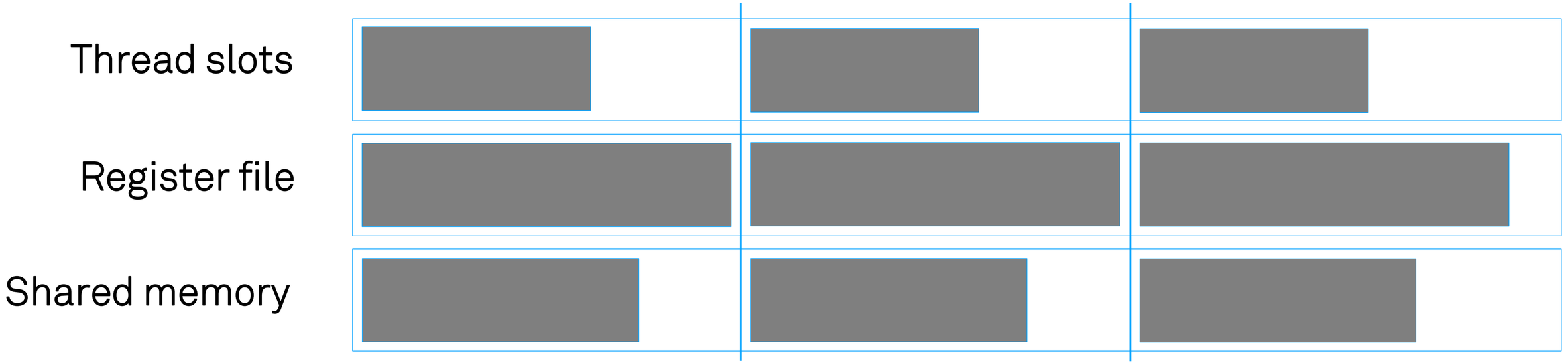
- Features specialized Tensor and RT cores
- Tensor cores can operate on 4/8/16 bit integers and 16 bit half-precision floating points
- RT cores used for Ray-Tracing in graphics workloads



Resource partitioning

- The GPU consists of several (1 to 68) *streaming multiprocessors* (SMs)
- The SMs are fully independent
- Each SM contains several resources: Register file, Shared memory, Thread Slots, and Thread Block slots

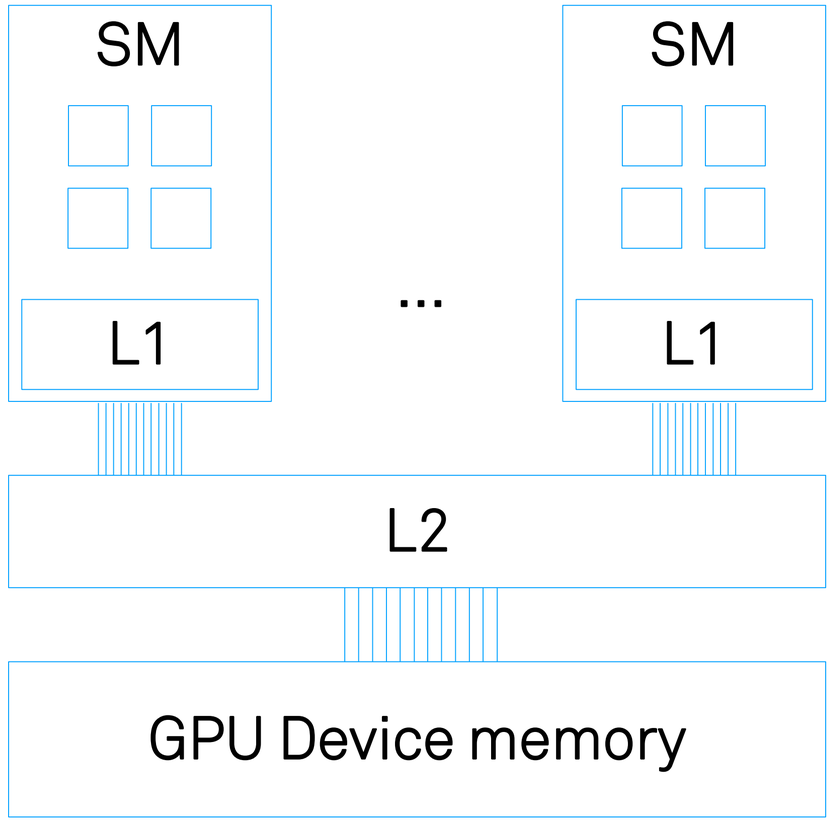
- SM resources are dynamically partitioned among the thread blocks that execute concurrently on the SM, resulting in a certain *occupancy*



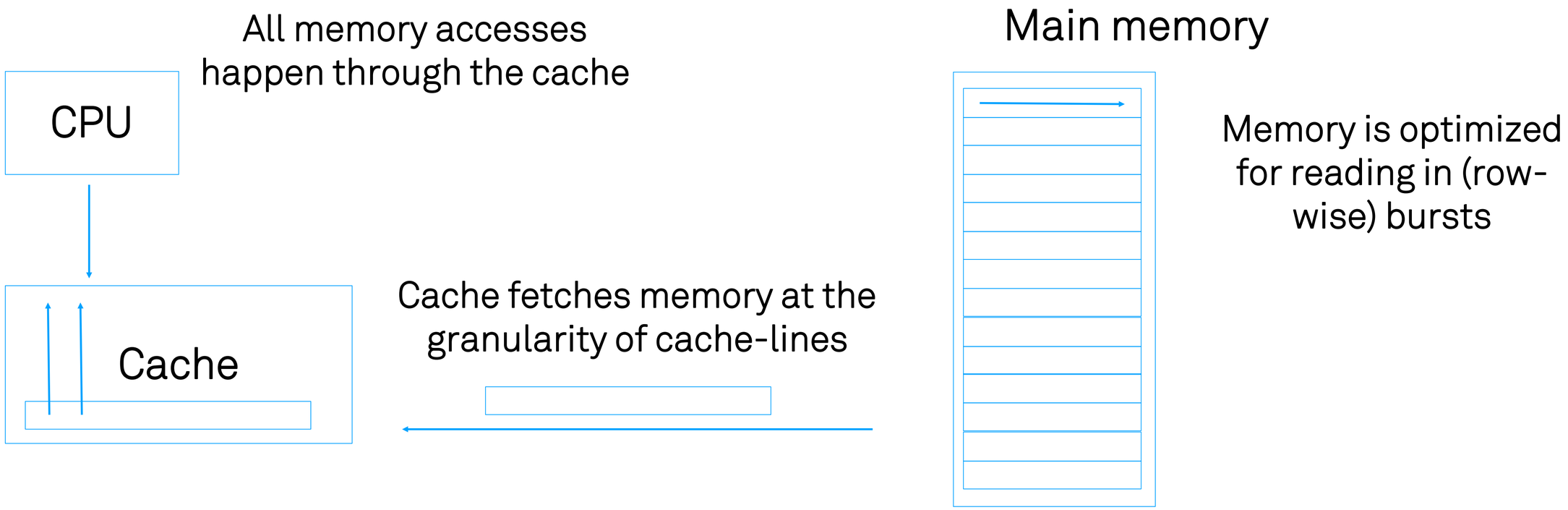
Global Memory access

- 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



The memory hierarchy is optimized for certain access patterns



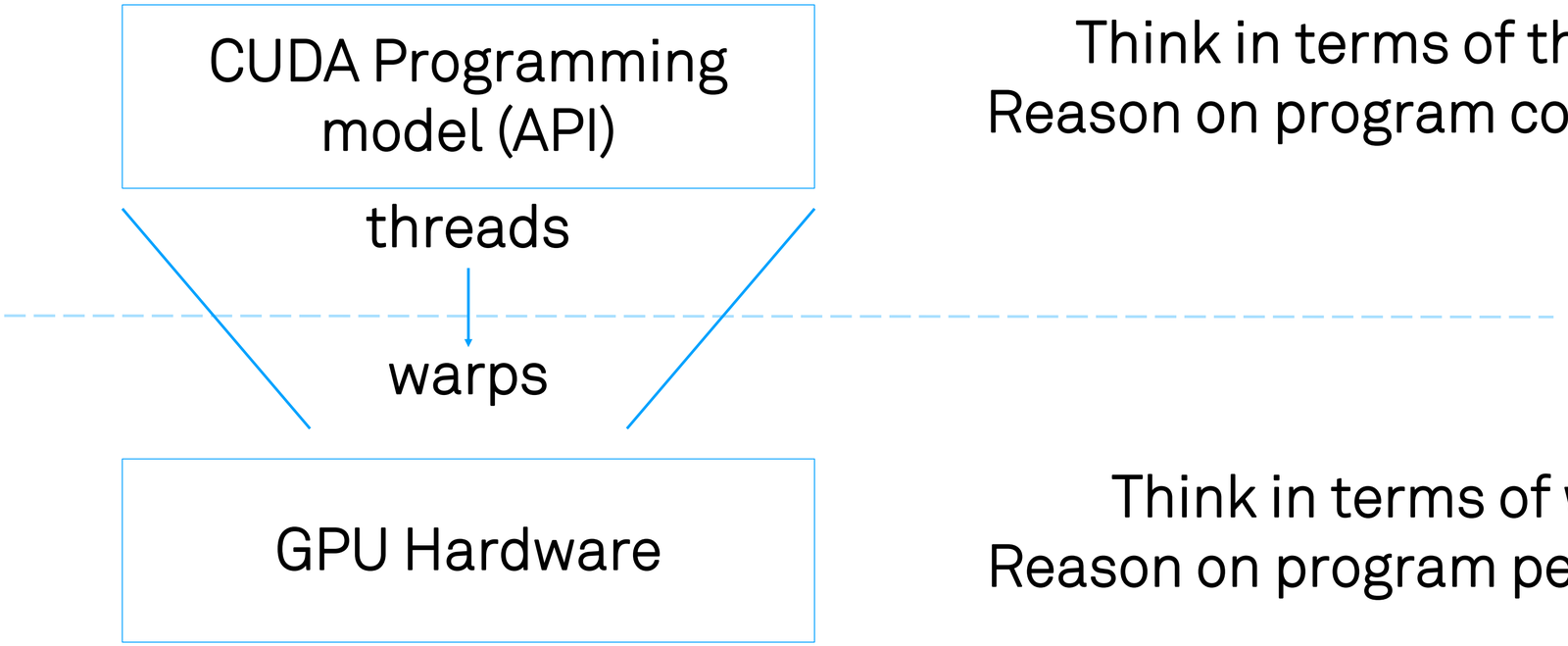
Subsequently accessing values that are adjacent on the same cache line is much faster than when each access requires a new cache line to be fetched

- 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

- 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)



Think in terms of threads
Reason on program correctness

Think in terms of warps
Reason on program performance

netherlands eScience center

GPU Computing in Radio Astronomy

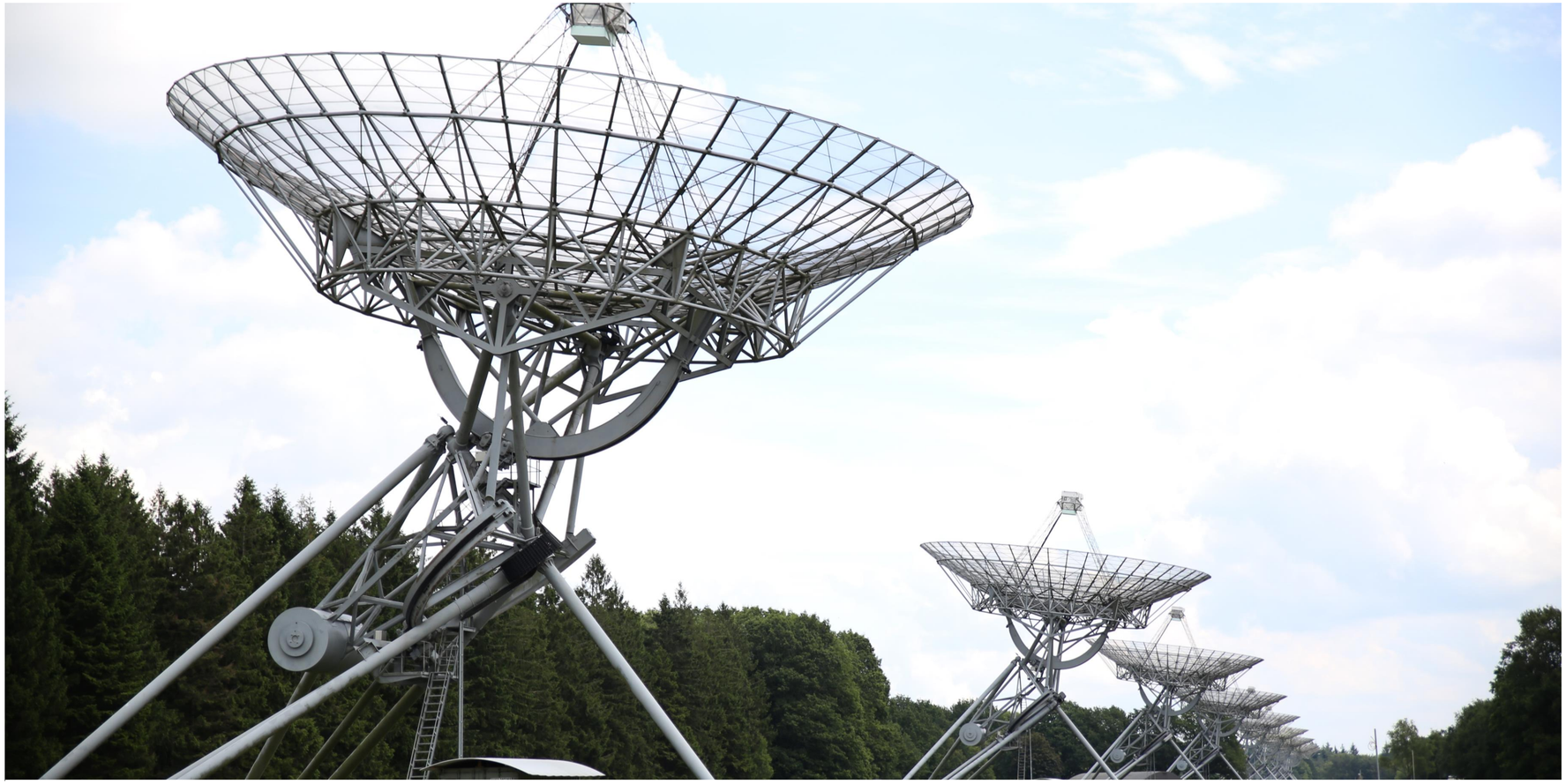
Lessons learned accelerating scientific software

"Astronomy is a natural science that studies celestial objects and phenomena. It applies mathematics, physics, and chemistry, in an effort to explain the origin of those objects and phenomena and their evolution." — Wikipedia

"Radio astronomy is a subfield of astronomy that studies celestial objects at radio frequencies." — Wikipedia

Modern Radio Telescopes

"Radio astronomy is conducted using large radio antennas referred to as radio telescopes, that are either used singularly, or with multiple linked telescopes utilizing the techniques of radio interferometry and aperture synthesis." — Wikipedia



Westerbork Telescope, photo by Elodie Burrillon

- Current trends in experimental science
  - Scale of instruments
    - Large Hadron Collider
    - Virgo
  - Use of software

- Radio astronomy is a good example of these trends
  - Enormous radio telescopes
    - Square Kilometer Array (SKA)
    - Very Large Array (VLA)
    - LOFAR
- Software telescopes

- How to process the data of the SKA, if it “will produce 10 times the global internet traffic”?

- Collect, store, process?
  - Straightforward approach
  - Difficult to scale

- Real-time processing?
  - Scales better
  - Difficult to implement

Computing Challenges of Radio Astronomy: Efficiency

- Real-Time processing of observational data
  - SKA may require a 100 PFLOP/s supercomputer
    - Summit has peak performance of 200 PFLOP/s

- Real applications seldom achieve high efficiency
  - Summit
    - HPL: 143 PFLOP/s — 71%
    - HPCG: 2.9 PFLOP/s — 1.5%

- Do we need 67 Summit supercomputers for the SKA?

Computing Challenges of Radio Astronomy: Portability and Energy Efficiency

- Portability
  - Telescopes are in use for decades
  - Computing infrastructure has much shorter life span

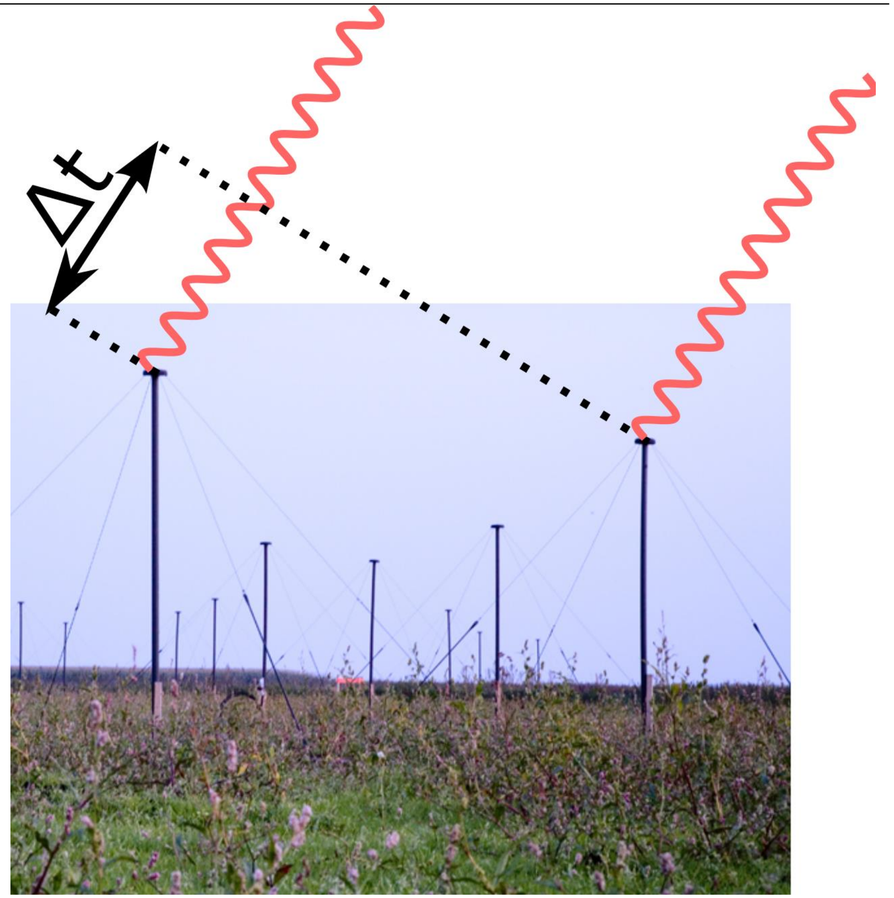
- Power constraints
  - Cost for electricity
  - Availability of power
  - Sustainability

- Fast
  - Massively parallel
  - High throughput
  - High memory bandwidth
- Energy efficient
  - More operations per Watt
  - NVIDIA GTX 1080 Ti: 250 Watt
  - 40 GFLOPs/Watt
- Cheap
  - NVIDIA GTX 1080 Ti: 699 USD
  - 14.3 GFLOPs/USD

Use Case: Beam Forming

Beam Forming

- The beam forming algorithm controls the spatial selectivity of omnidirectional antennas
- Used in radio astronomy, WiFi and 4G networks, radars, medical imaging, audio processing, etc.
- Used in radio astronomy to point software telescopes like LOFAR
- A software beam former permits to point a telescope in hundreds of directions, at the same time



The Beam Forming Algorithm

- How does it work?
  - Linear combination
    - Each received signal is compensated by shifting its phase
    - All signals are added together
  - The phase shift depends on the positions of source and receiver

- The number of operations is O(f * t * p * r * b)
  - f is the number of frequencies
  - t is the number of time samples
  - p is the number of polarizations
  - r is the number of receivers
  - b is the number of beams to form

- Hierarchical parallelism
  - Threads are divided in groups
    - Each group is associated with one frequency
  - Inside a group of threads
    - Each thread is associated with one time sample and all polarizations
    - Each thread is responsible for computing all beams

- The input of a receiver is used to compute every beam
  - Keep it in a register and reuse it
- The algorithm is modified to
  - Compute a block of beams for each iteration

Looking for a Silver Bullet

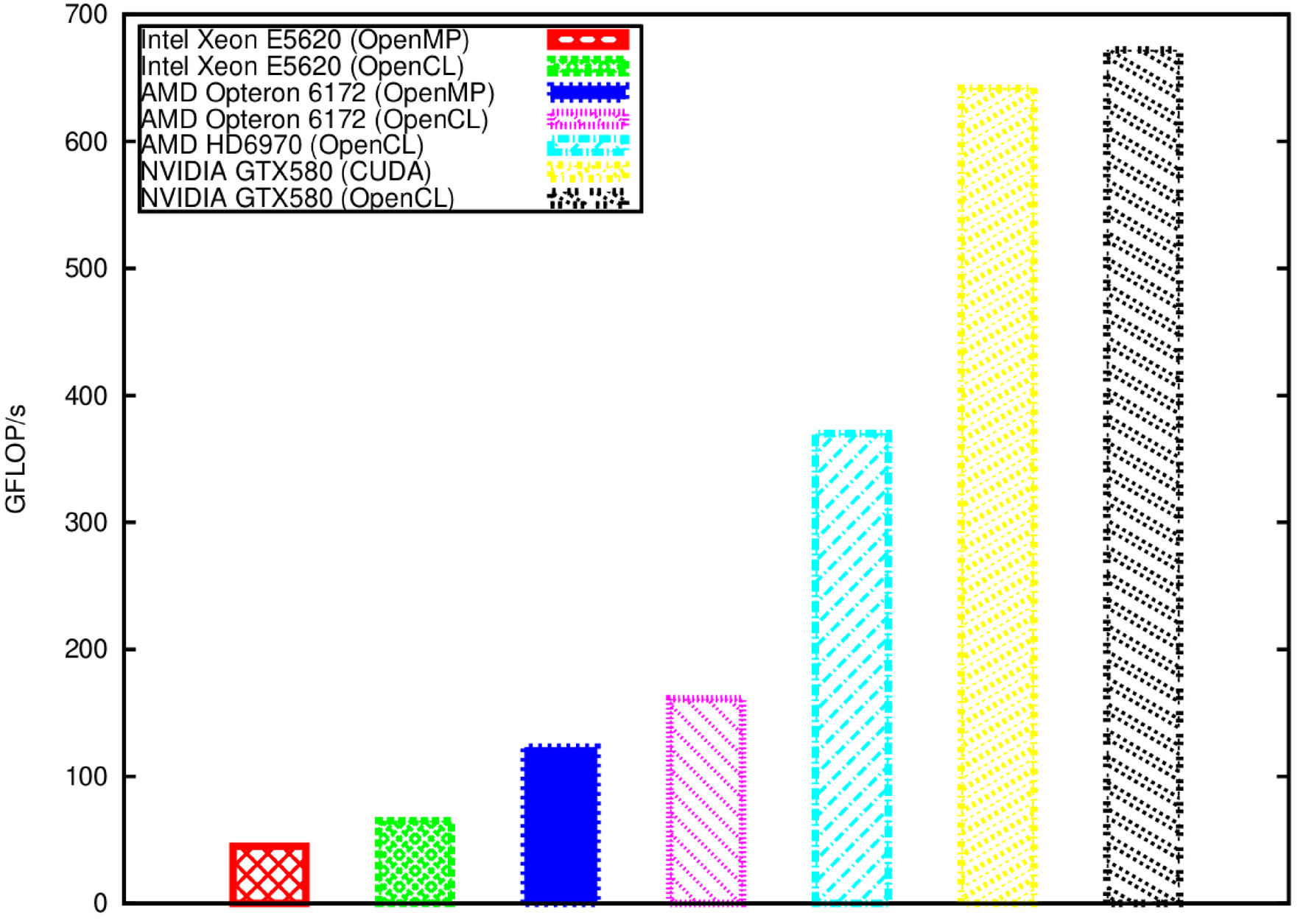
- How many beams should we compute in a single iteration?
  - All of them
  - Some of them

- Is this number different for different hardware platforms?
- And what about different implementation frameworks?

- Taking this decision affects performance

- Auto-Tuning
  - Test different options and pick the best

Performance Results

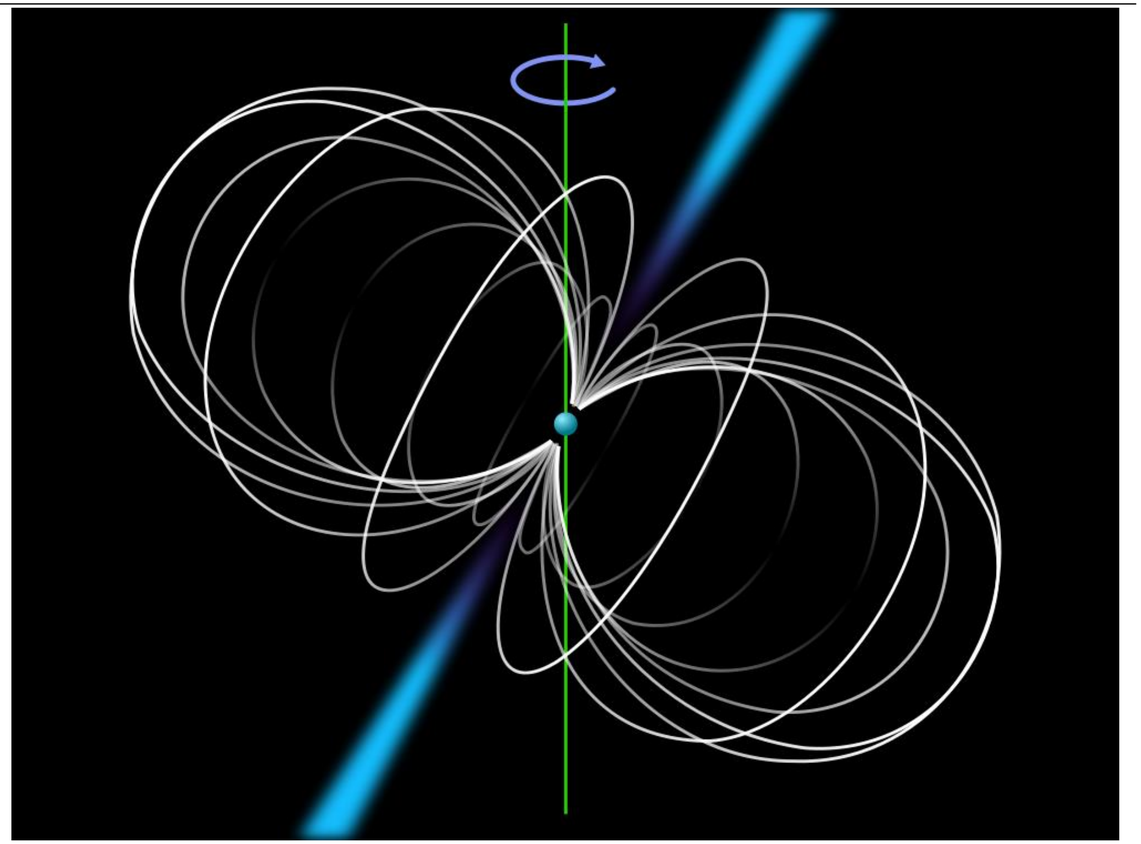


Use Case: Dedispersion

Transients

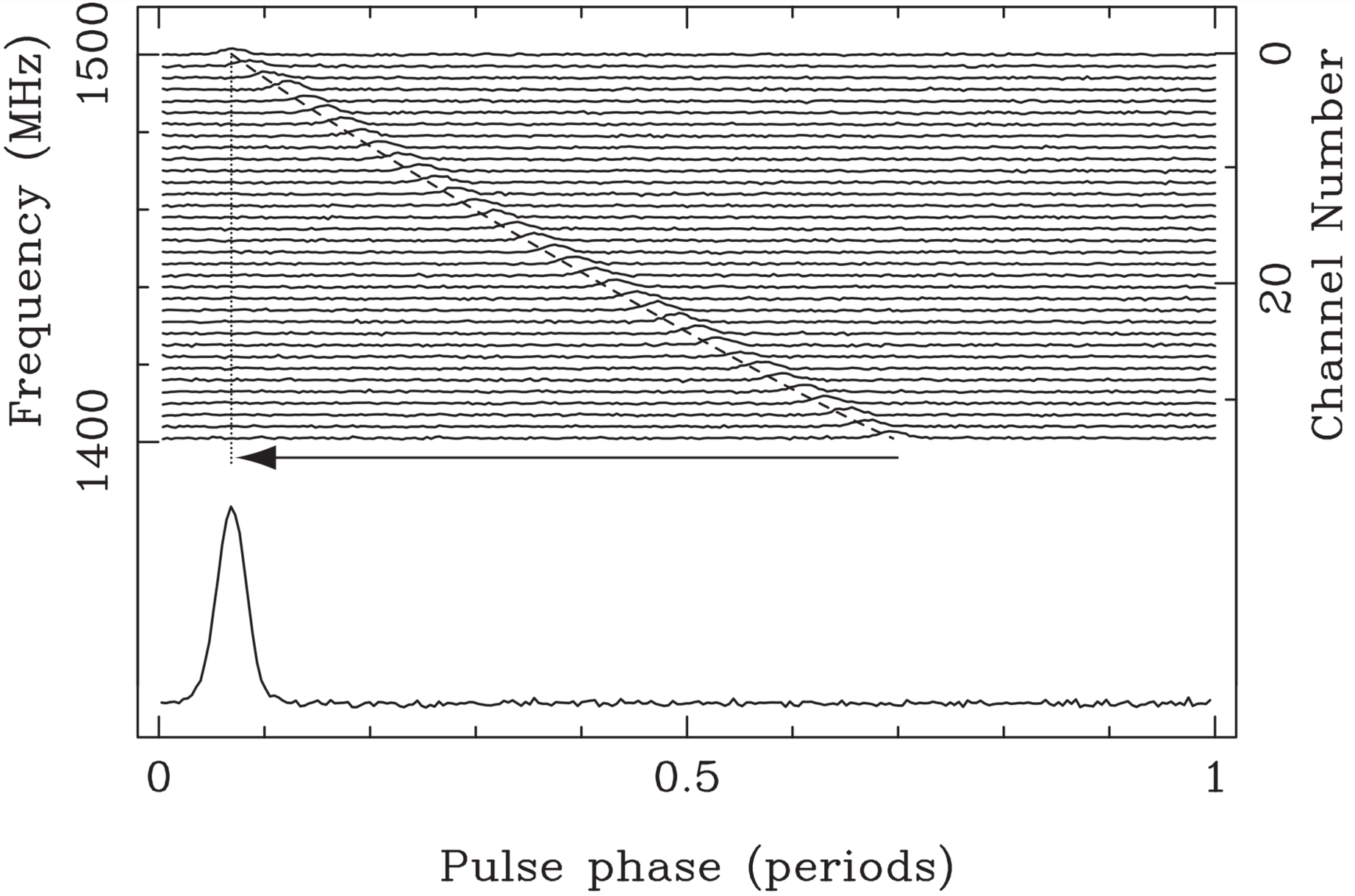
- Finding transients is a hot-topic in astronomy
  - Pulsars
  - Fast Radio Bursts
  - Supernovae

- Interesting objects, but difficult to find
  - Radio Frequency Interference (RFI)
  - Dispersion, scintillation, scattering
  - Gravitational interactions
  - Fast periods
  - Faint signals



Pulsar by Roy Smits

Dispersion



Dispersion by Lorimer

The Complexity of Dealing with Dispersion

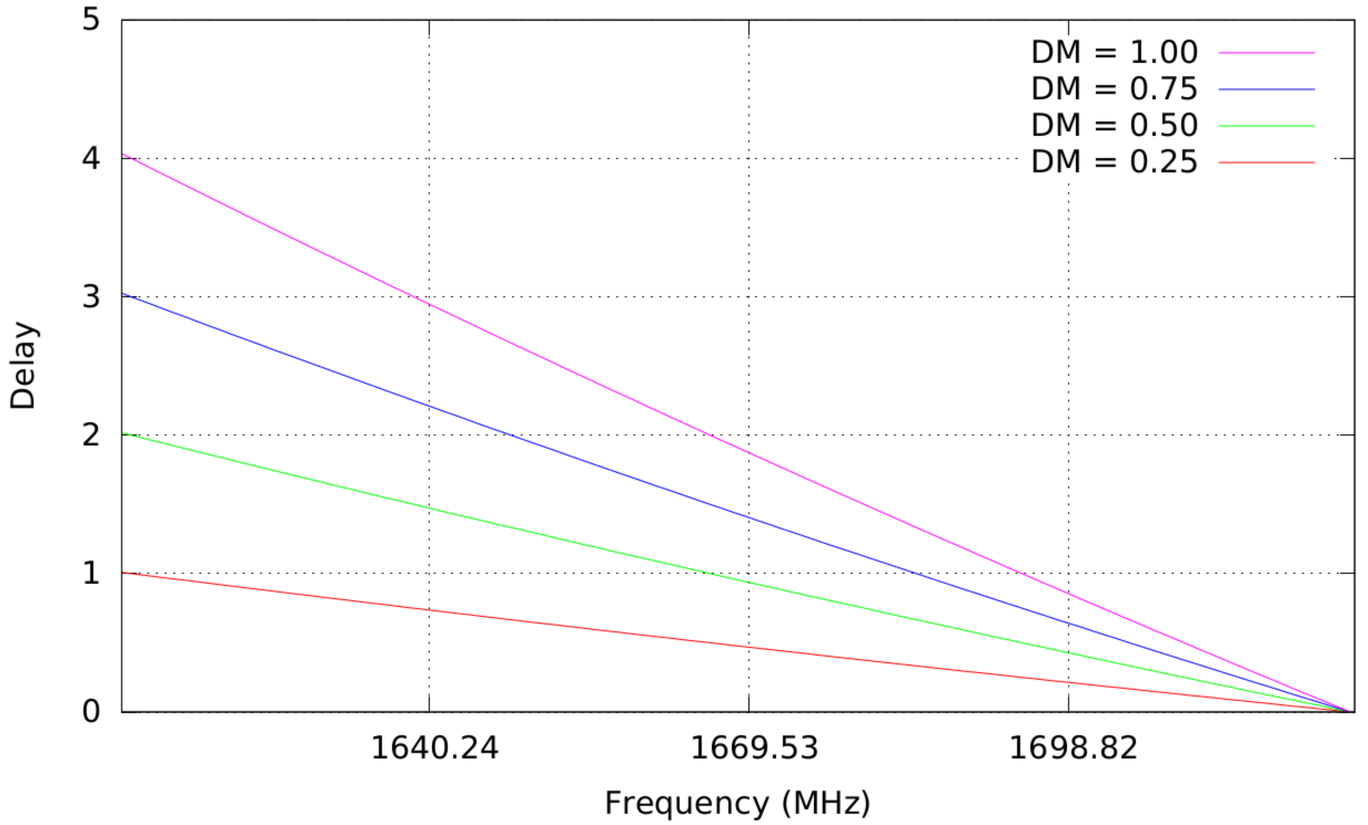
- Complexity:
  - Single step: O(f * t)
  - Search: O(b * d * f * t)
    - d is the number of dispersion measures (DMs)

- Naturally parallel, but is it enough?

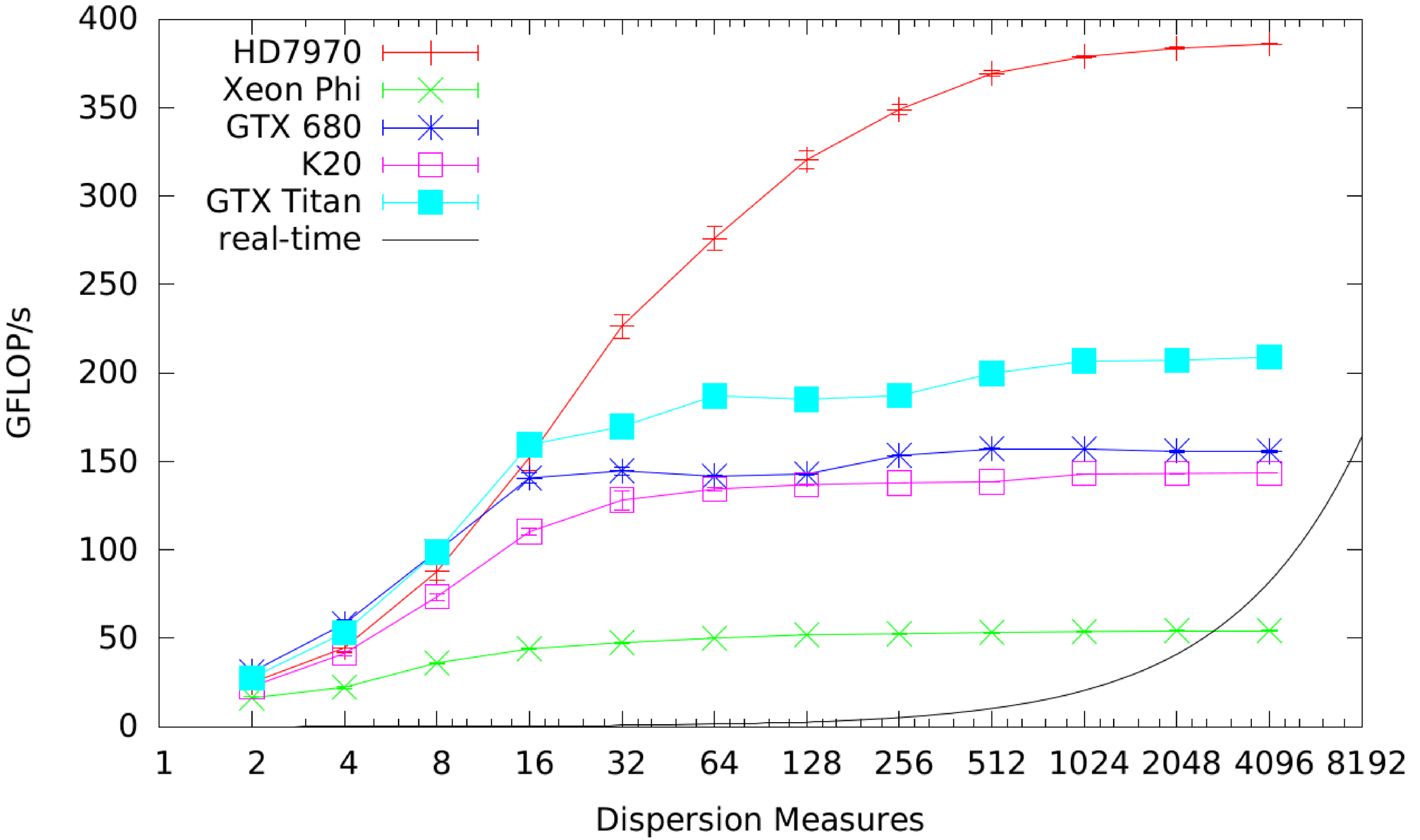
- Memory-bound
  - Arithmetic intensity less than 0.25

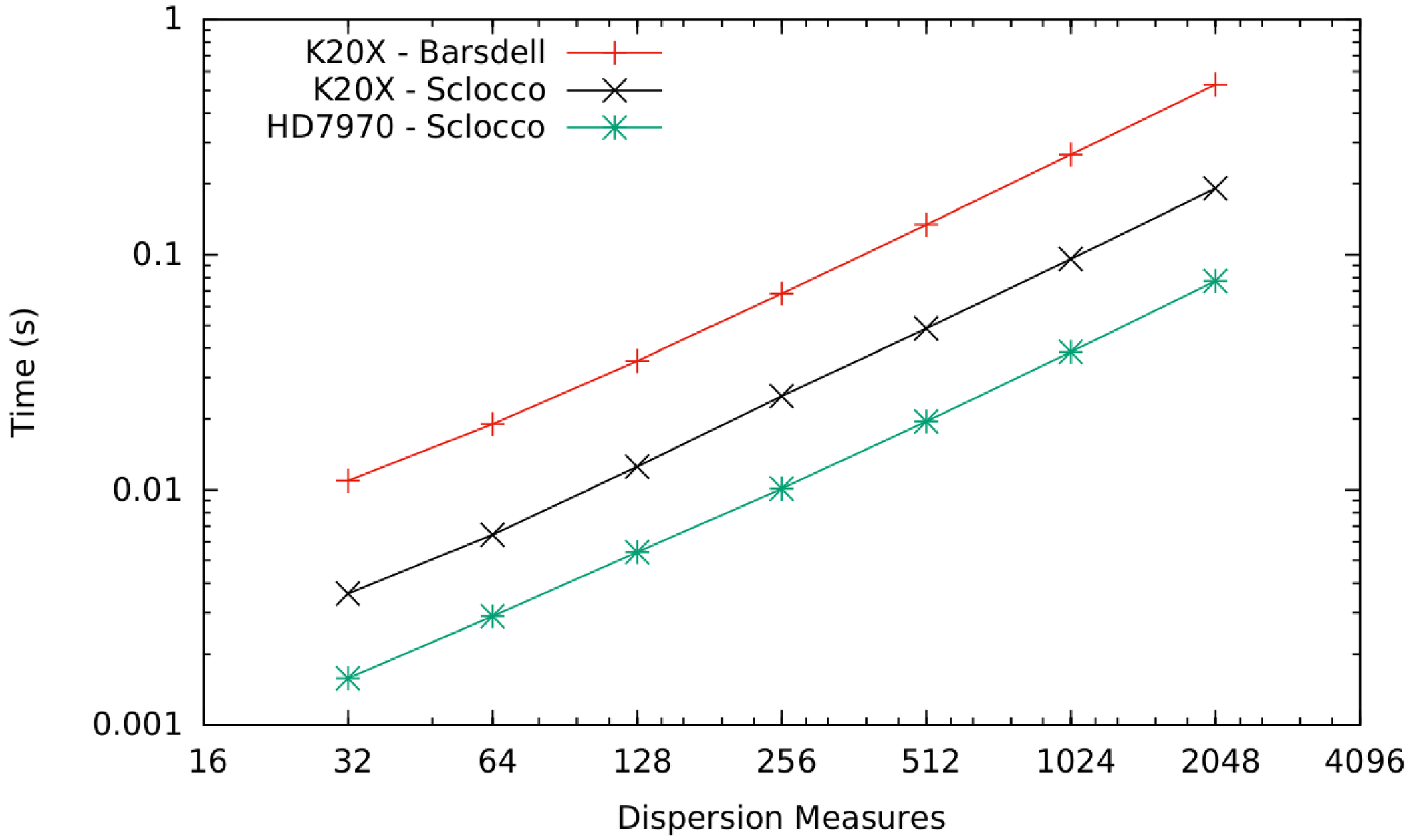
Optimizations

- Discrete delays partially overlap
  - Possible data reuse
- Data reuse affects arithmetic intensity
- Computing more than one DM per thread
- Auto-tuning once more necessary

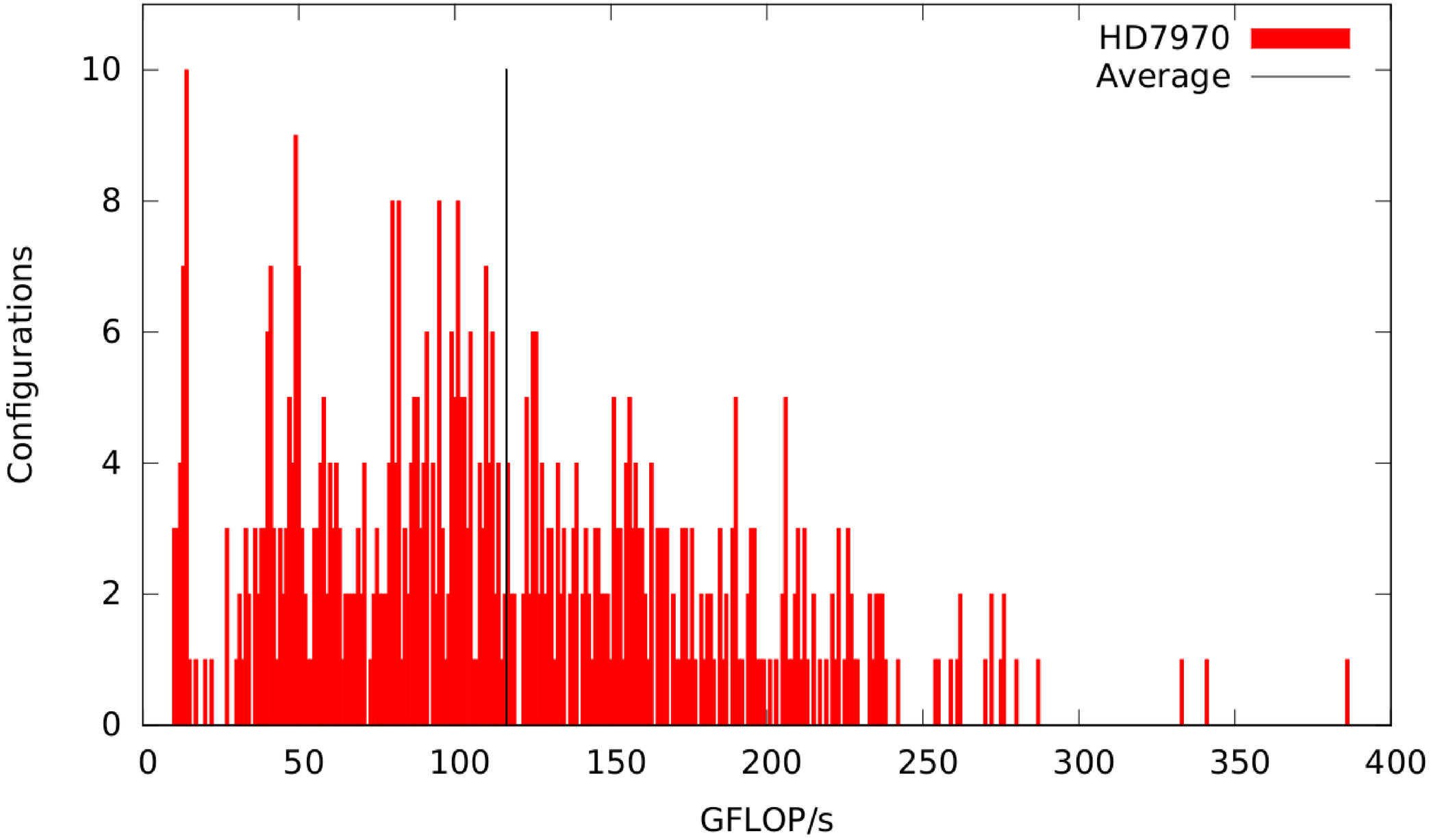


Performance Results




K20X - Barsdell
K20X - Sclocco
HD7970 - Sclocco
Time (s)
1
0.1
0.01
0.001
16
32
64
128
256
512
1024
2048
4096
Dispersion Measures

Auto-Tuning Can Be Difficult



Lessons Learned

- Experimental science needs High-Performance Computing
- GPUs can be used to accelerate science
- Parallelism is important, but may not be enough for performance
- In many cases memory is the bottleneck
  - Optimize for memory
  - Increase data reuse
- Auto-tuning can make a difference

GPU kernel optimization techniques

Optimizing Code (intentionally repeated slide)

- 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

- 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

- Algorithmic Optimization
  - Cleverly rewriting your algorithm to reduce the total amount of computations
  - Reducing the complexity in terms of ‘Big-O notation’

- Execution Optimization
  - Increasing hardware utilization without doing double work
  - Optimizing the code such that it maps efficiently onto the hardware
  - Rewriting the code such that it will run faster on the hardware

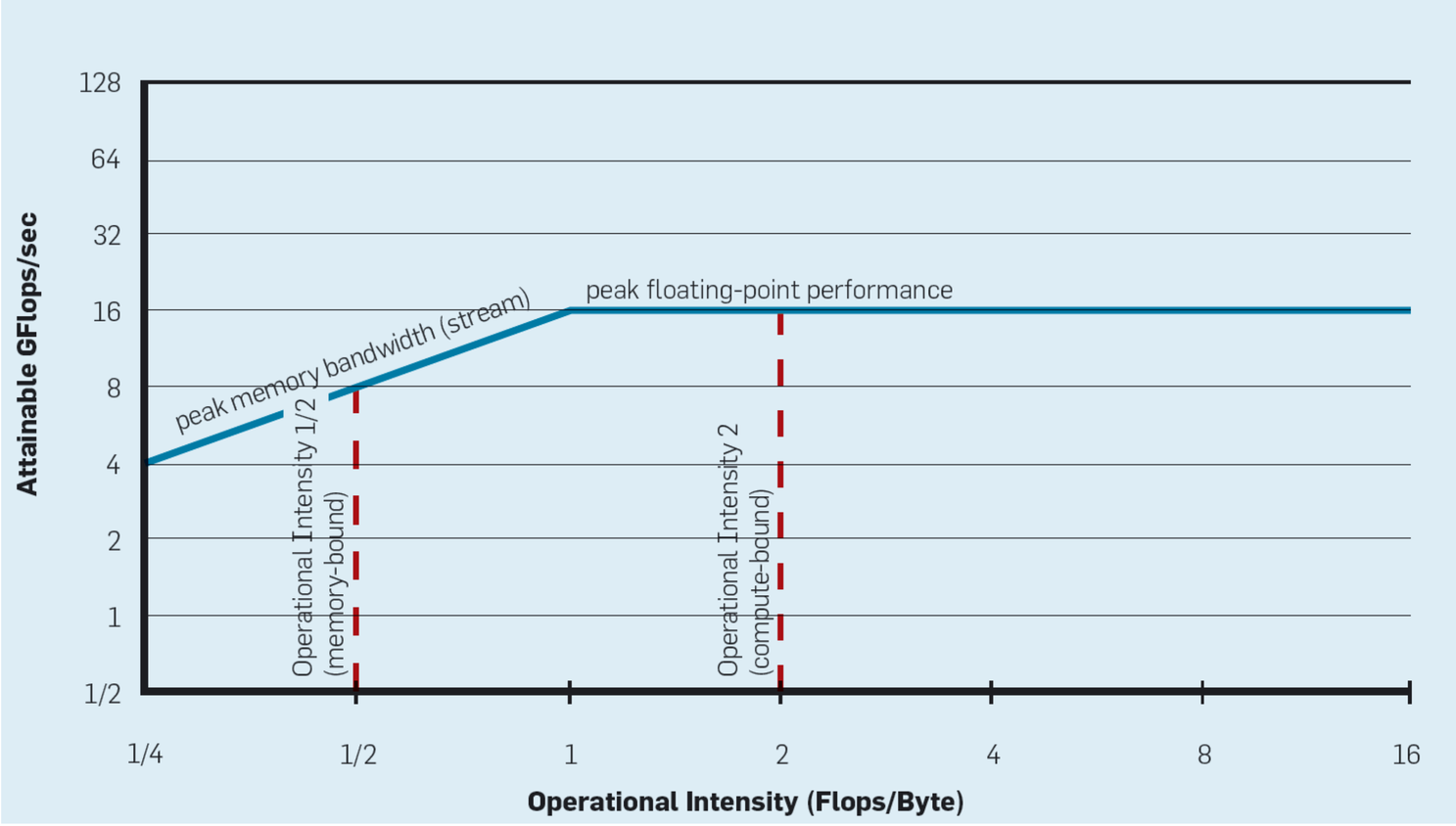
Arithmetic Intensity and Operational Intensity

- Arithmetic Intensity is the ratio between compute operations and memory operations
- $$\text{Arithmetic Intensity} = \frac{\text{total operations}}{\text{total bytes required}}$$
- $$\text{Operational Intensity} = \frac{\text{total operations}}{\text{total bytes actually loaded from memory}}$$
- Actual bytes loaded from memory may be different, because memory loads are performed through the cache-hierarchy

Roofline model

- The roofline model tells us if our kernel is compute-bound or memory-bandwidth bound

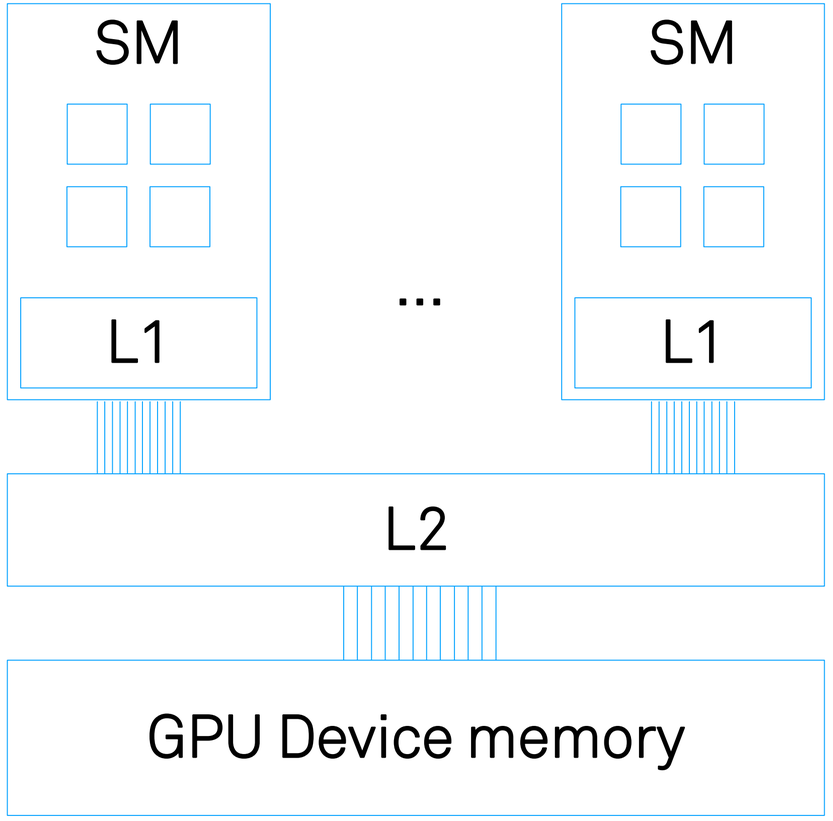
- Attainable performance (e.g. in GFLOP/s) =
  min(theoretical peak compute performance,
  theoretical peak memory bandwidth x operational intensity)



- Roofline model tells us to optimize the operational intensity

- Remember:
  Operational Intensity = $\frac{\text{total operations}}{\text{total bytes loaded}}$

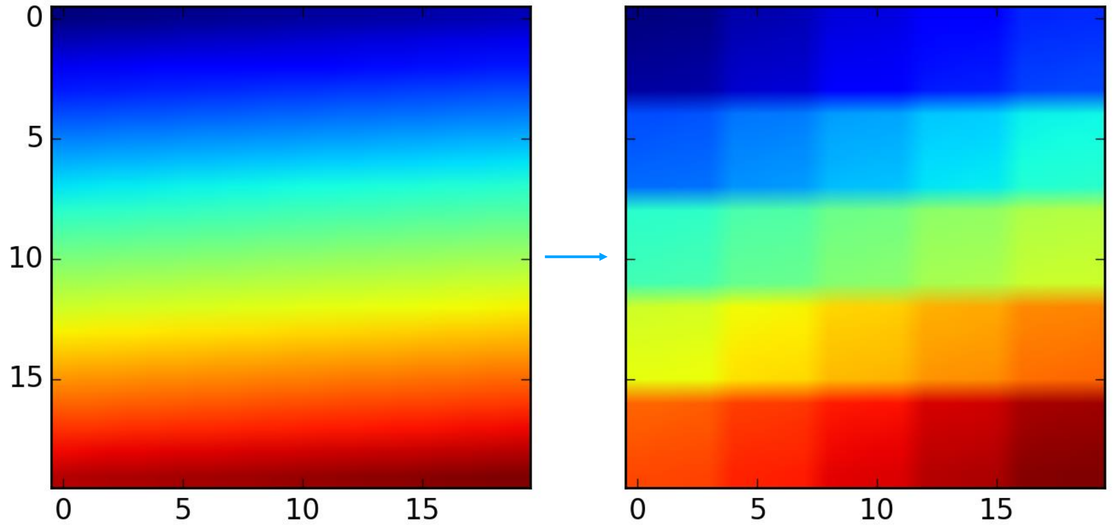
- The most important thing is to mimize the bytes loaded from memory. This means reusing data in memory locations closest to the functional units for as long as possible.



Loop blocking

- Loop blocking a technique to change the iteration order to improve data locality
- Also known as tiling, or loop nest optimization

```
for (int j=0; j<m; j++)
    for (int i=0; i<n; i++)
        some_array[j*n+i]
```



- After applying loop blocking:

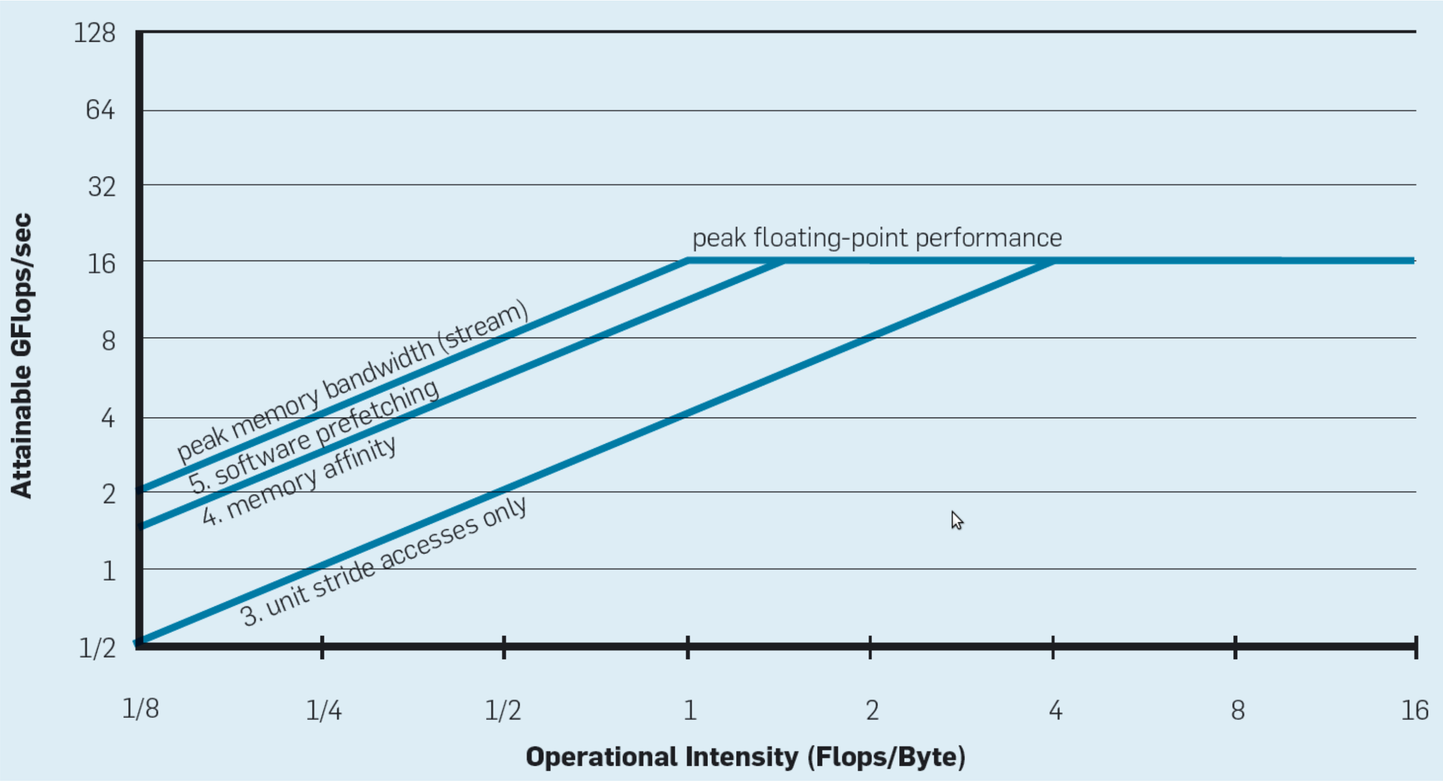
```
for (int j=0; j<m; j+=block_size)                  //iterate over blocks
    for (int i=0; i<n; i+=block_size)

        for (int bj=0; bj<block_size; bj++)  //iterate within each block
            for (int bi=0; bi<block_size; bi++)

                some_array[(j+bj)*n+i]
```

Optimizing achieved memory bandwidth

- However, our GPU code is not likely to achieve the theoretical peak memory bandwidth, unless we pay close attention



If your kernel's performance is bandwidth limited, think of this as the following:

"the performance is limited by the rate at which operands
can be supplied to the functional units"

This means your kernel is either latency or throughput bound

Hiding memory latency is about keeping a sufficient number of data operations in flight

This requires sufficient amounts of parallelism:

- Thread-level parallelism: ensure that the number of executing threads remains high
- Instruction-level parallelism: give threads enough independent work to avoid stalls on earlier instructions, e.g. using vector data types
- Ensure that memory operations can be served in parallel, avoid bank-conflicts in shared memory and registers

GPU Memory is optimized for accesses in a ‘coalesced’ manner:

- This means that threads within a warp (adjacent in the x-dimension, and after that y-dimension), should access consecutive memory locations.

Remember threads are executed in warps:

- Memory accesses by threads are grouped into warps and mapped onto cachelines
- The cachelines are then loaded from or stored to memory
- If threads load data with a stride, for example `array[threadIdx.x * 2]`, the amount of data actually loaded from memory is twice as large
- If you only use each other element, you essentially half your effective bandwidth!

Ideally, threads should only read and write consecutive and aligned memory locations

- Think of cacheline lengths at various cache levels
- Be aware of cacheline alignment in memory

Ensuring sufficient paralellism

- Ensuring sufficient thread-level parallelism is about keeping the GPU's SMs sufficiently occupied, one way to control this is to vary the number of threads per block

- Remember occupancy depends on resource usage: shared memory, registers, thread slots, and thread block slots.

- Thread block dimensions are often fixed at compile, because they are used:
  - to declare the size of shared memory arrays
  - as bounds in for-loops, which may be unrolled if known at compile time
  - as offsets within index calculations that can be optimized away by the compiler

- Registers are per-thread and are determined by the compiler, so the amount of registers needed by the thread block can be controlled by adjusting the size of the thread block

- Figuring out the optimal thread block dimensions for your kernel is usually not possible without experimentation

Further reading on GPU memory optimization

Excellent presentations on GPU memory optimization:

- GPU Memory bootcamp talks by Tony Scudiero [git repo]
  - Best Practices [Slides] [Video]
  - Beyond the Best Practices [Slides] [Video]
  - Collaborative Access Patterns [Slides] [Video]

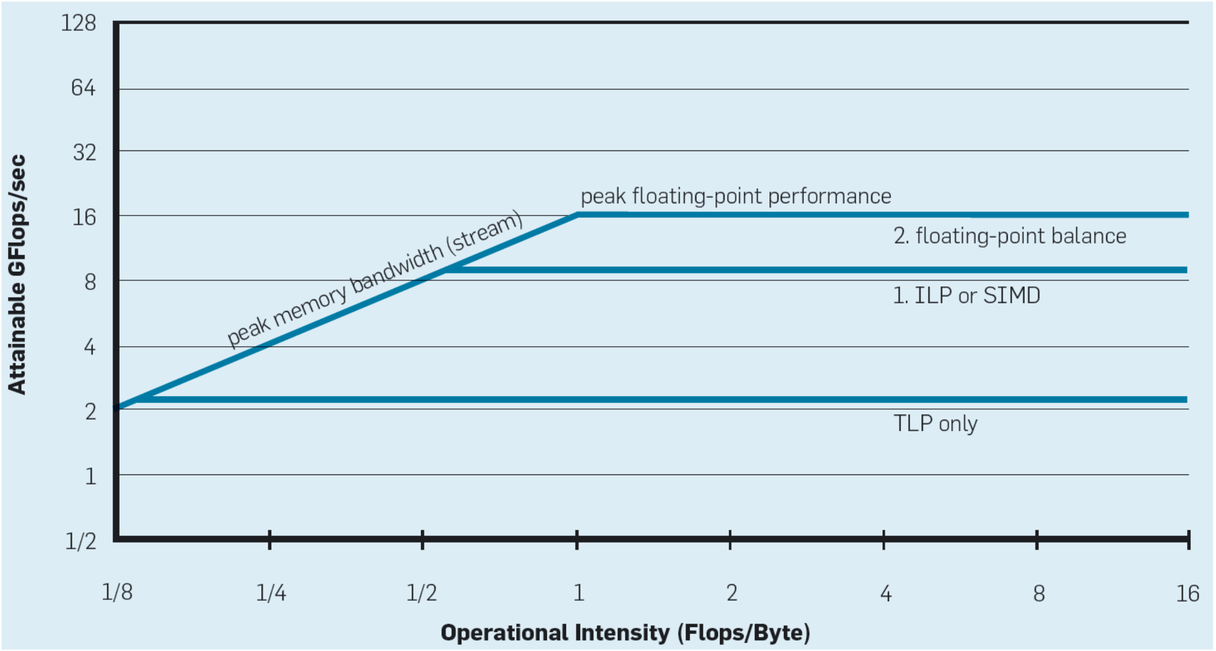
Instruction stream optimization

If your kernel is compute-bound, it might still not achieve the peak theoretical compute performance of your GPU

Most important reason for this is that your kernel is not issuing compute instructions to all functional units at every cycle while it's running

Your kernel is doing more than just floating-point arithmetic, for example: memory operations, barrier synchronizations, index calculations, or checking loop conditions.

Some of this is inherent to the algorithm your kernel implements, but some instruction overhead can be avoided



Increasing work per thread

A very common code transformation for GPU code is increasing work per thread
(a.k.a thread coarsening / 1xN tiling / thread block merge)

- Before:

```
sum = a[i] + b[i];
```

- After:

```
sum0 = a[i] + b[i];
sum1 = a[i + 1] + b[i + 1];
```

- Or the right way for the GPU, without introducing stride-2 memory accesses:

```
sum0 = a[i] + b[i];
sum1 = a[i + block_size] + b[i + block_size];
```

The above assumes `block_size` is a multiple of the warp size

Increasing work per thread

What we just did is doubling the amount of work per thread, so for the same input size, so our kernel now only needs half the number of threads to execute

Advantages:

- We reduce a number of redundant operations that were previously distributed across multiple threads, e.g. array index calculations, loop accounting.

- Additionally, if there are multiple threads that need to load the same data, we are now reusing that data in registers.

Disadvantages:

- We trade thread-level parallelism for instruction-level parallelism, increased resource usage further reduces parallelism when it limits occupancy

- Why only double the amount of work? Why not increase work by an arbitrary factor, for example named `tile_size`?

```
#pragma unroll
for (int ti=0; ti<tile_size; ti++) {
   sum = a[i+ti*block_size] + b[i+ti*block_size];
}
```

Note that we've just increased work in one dimension. If we are working on an N-dimensional problem, we might just increase work in any of up to N of dimensions.

```
#pragma unroll
for (int ti=0; ti<tile_size_x; ti++) {
   #pragma unroll
   for (int tj=0; tj<tile_size_y; tj++) {
      sum = a[(j+tj)*n+i+ti*block_size] + b[(j+tj)*n+i+ti*block_size];
   }
}
```

Loop fusion

Another way to reduce redundant instructions and improve data locality is to fuse loops

- Before:

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

- After:

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

If N is small and known at compile time the compiler will do this for you

(Partial) loop unrolling

Another way to reduce instructions and increase instruction-level parallelism is to unroll loops

- Before:

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

- After:

```
for (int i=0; i<N; i+=2) {
   c[i] = a[i] + b[i];
   c[i+1] = a[i+1] + b[i+1];
}
```

This of course only works if 2 is a divisor of N

- Even better:

```
#pragma unroll <factor>
for (int i=0; i<N; i++) {
   c[i] = a[i] + b[i];
}
```

If N is known at compile-time the compiler will try to apply (partial) loop unrolling

Overview of GPU optimization trade-offs

Improve memory throughput

- Find the best way to map your algorithm's data-parallel iteration space to threads and thread blocks
- Coalesce memory accesses, possibly by using shared memory as intermediate storage
- Increase operational intensity as much as possible by utilizing specialized memories and apply various levels of loop blocking to improve cache utilization

Minimize memory latency

- Reduce/hide memory latency by using vector data types to increase size of memory loads and stores
- Fuse and/or partially unroll loops to condense instruction stream and increase instruction-level parallelism

Condense the instruction stream

- Select the best performing thread block dimensions, trade-off between instruction stream efficiency, data reuse, and latency hiding
- Increase or decrease work per thread in N dimensions to condense instruction stream, at the cost of increased resource usage

Requires that you get all the details exactly right:

- Mapping of the problem to threads and thread blocks
- Thread block dimensions
- Data layouts in the different memories
- Tiling factors
- Loop unrolling factors
- How to overlap computation and communication
- …

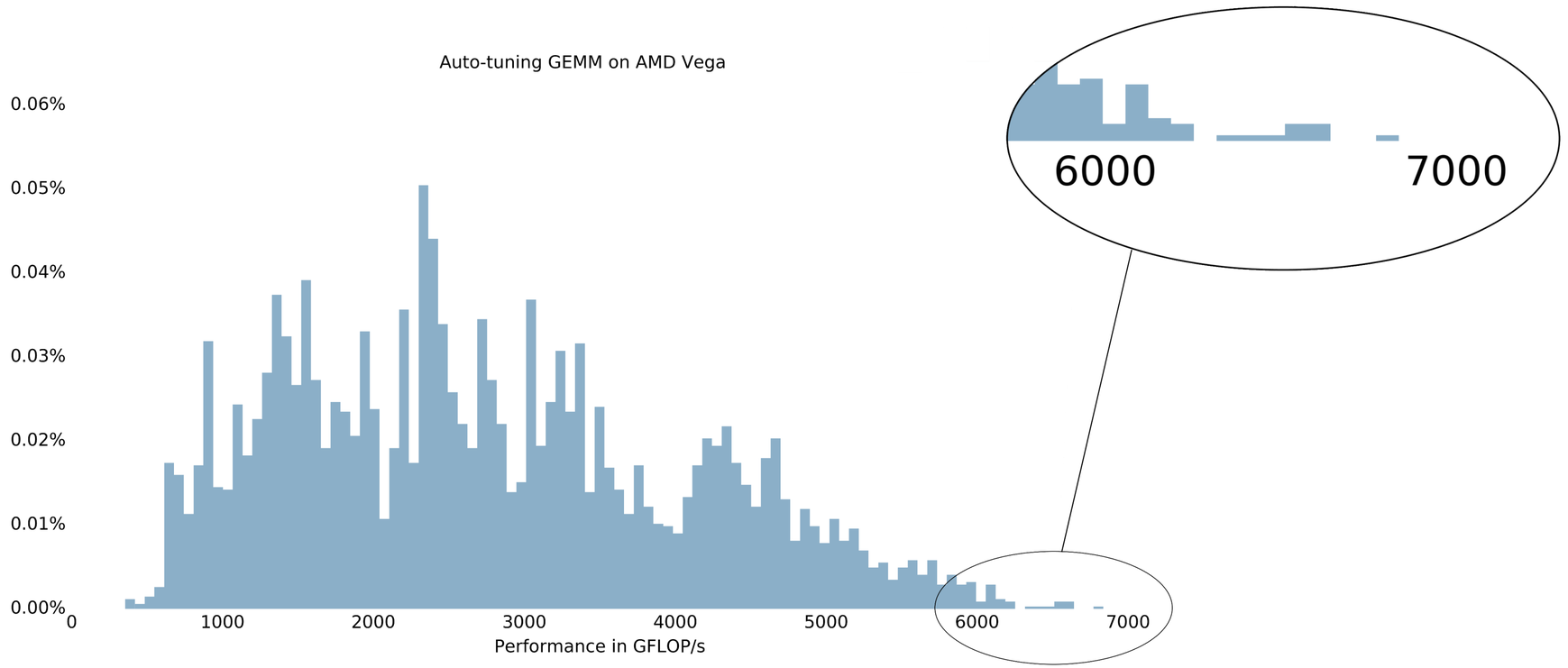
Problem:

Creates a very large and discontinuous search space

Auto-Tuning for performance

- The number of combinations to try explodes rather quickly, even for single kernels, not to mention for tuning pipelines
- The best performing combination of tunable parameters will be different on different GPUs, and for different input sizes
- The best performing combination is often very hard to find!

- **Auto-tuning** is the process of automatically searching for the best performing kernel configuration



Kernel Tuner: a generic auto-tuner in Python

Easy to use:

- Can be used directly on existing kernels and code generators
- Inserts no dependencies in the kernels
- Kernels can still be compiled with regular compilers

Supports:

- Tuning functions in OpenCL, CUDA, C, and Fortran
- Large number of effective search optimizing algorithms
- Output verification for auto-tuned kernels and pipelines
- Tuning parameters in both host and device
- Python-based unit testing of GPU code
- …

https://github.com/benvanwerkhoven/kernel_tuner