ASCI A24 GPU Course

December 14 2022

Alessio Sclocco & Ben van Werkhoven

netherlands eScience center

Introduction: Alessio Sclocco

Alessio Sclocco
eScience Research Engineer
Netherlands eScience Center
a.sclocco@esciencecenter.nl

Background:

- 2011-2012 junior researcher at VU Amsterdam
  - Working on GPUs for radio astronomy
- 2012-2017 PhD "Accelerating Radio Astronomy with Auto-Tuning" at VU Amsterdam
  - Under the supervision or professors Henri Bal and Rob van Nieuwpoort
- 2015-2016 scientific programmer at ASTRON, the Netherlands Institute for Radio Astronomy
  - Designing and developing a real-time GPU pipeline for the Westerbork radio telescope
- 2019 visiting scholar at Nanyang Technological University in Singapore
- 2017-2022 eScience Research Engineer at the Netherlands eScience Center
  - Radio astronomy, climate modeling, biology, natural language processing, high-energy physics

Introduction: Ben van Werkhoven

Ben van Werkhoven
Senior Research Engineer
Netherlands eScience Center
b.vanwerkhoven@esciencecenter.nl

Background:

- 2010-2014 PhD “Scientific Supercomputing with Graphics Processing Units” at the VU University Amsterdam in the group of prof. Henri Bal
- 2014-now working at the Netherlands eScience Center as the GPU expert in many different scientific research projects

GPU Programming since early 2009, worked on applications in computer vision, digital forensics, climate modeling, particle physics, geospatial databases, radio astronomy, and localization microscopy

Schedule

- 09:00 – 09:10 - Course introduction
- 09:10 – 09:25 - Introduction to GPU computing
- 09:25 – 09:50 - Introduction to GPU programming
- 09:50 – 10:10 - Overview of GPU programming models
- 10:10 – 10:30 - Introduction to CUDA programming

- 10:30 – 11:00 - Coffee break
- 11:00 – 11:35 - Hands-on exercise 1
- 11:35 – 11:55 - Host code
- 11:55 – 12:10 - CUDA memories (part 1)
- 12:10 – 12:30 - Hands-on exercise 2

- 12:30 – 13:30 - Lunch
- 13:30 – 13:45 - CUDA memories (part 2)
- 13:45 – 14:15 - Hands-on exercise 3
- 14:15 – 14:30 - Asynchronous processing
- 14:30 – 14:45 - CUDA program execution

- 15:00 – 15:30 - Coffee break
- 15:30 – 16:00 - GPU optimizations
- 16:00 – 16:30 - Auto-tuning
- 16:30 – 16:55 - Hands-on exercise 4
- 16:55 – 17:00 - Closing remarks

- Get your own copy of the slides so you can read along and click on links
  - See: https://github.com/benvanwerkhoven/gpu-course/
  - Clone the repository to have access to slides and hands-on exercises

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

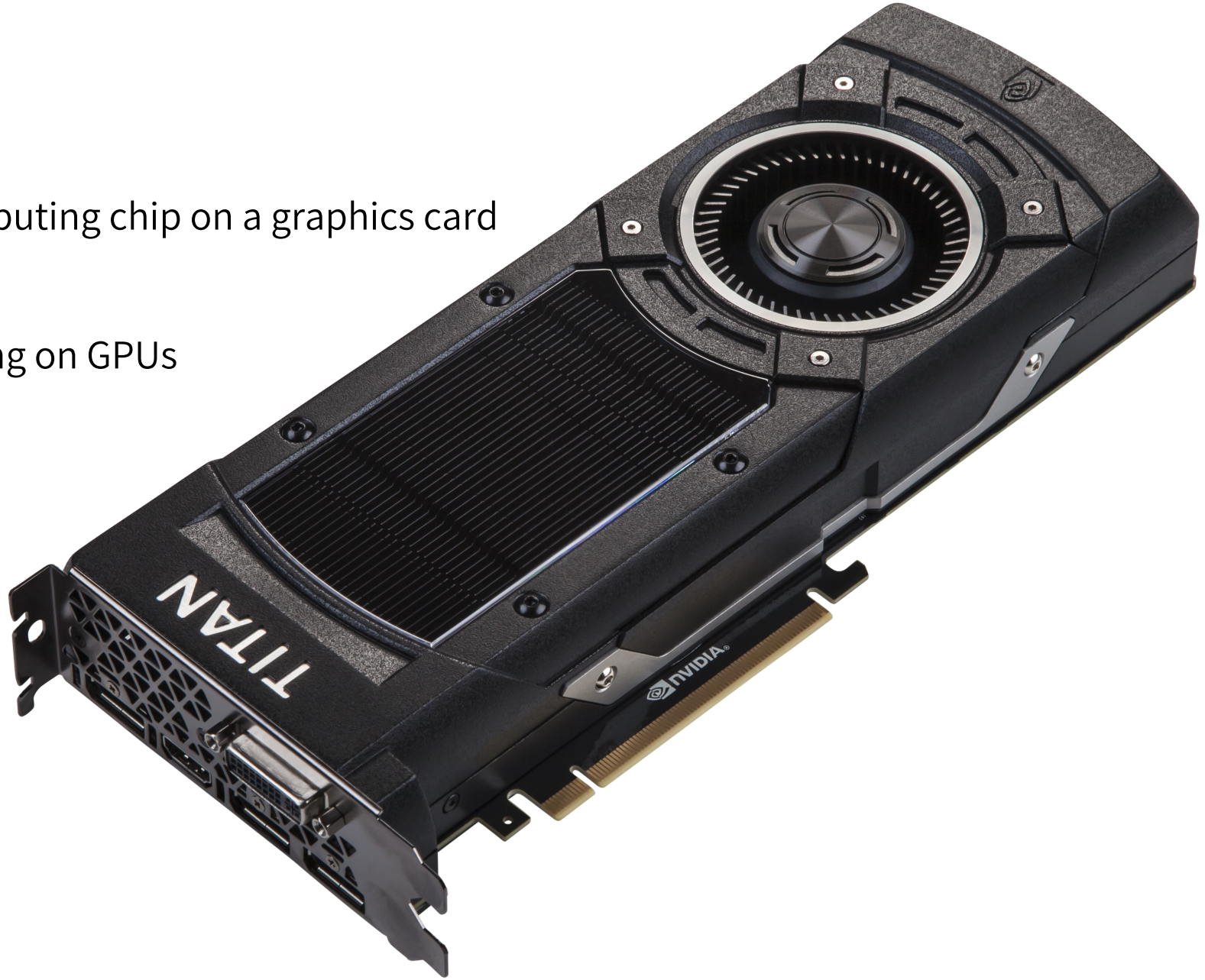
- In code samples on the slides we sometimes abbreviate the code a bit to save space

netherlands eScience center

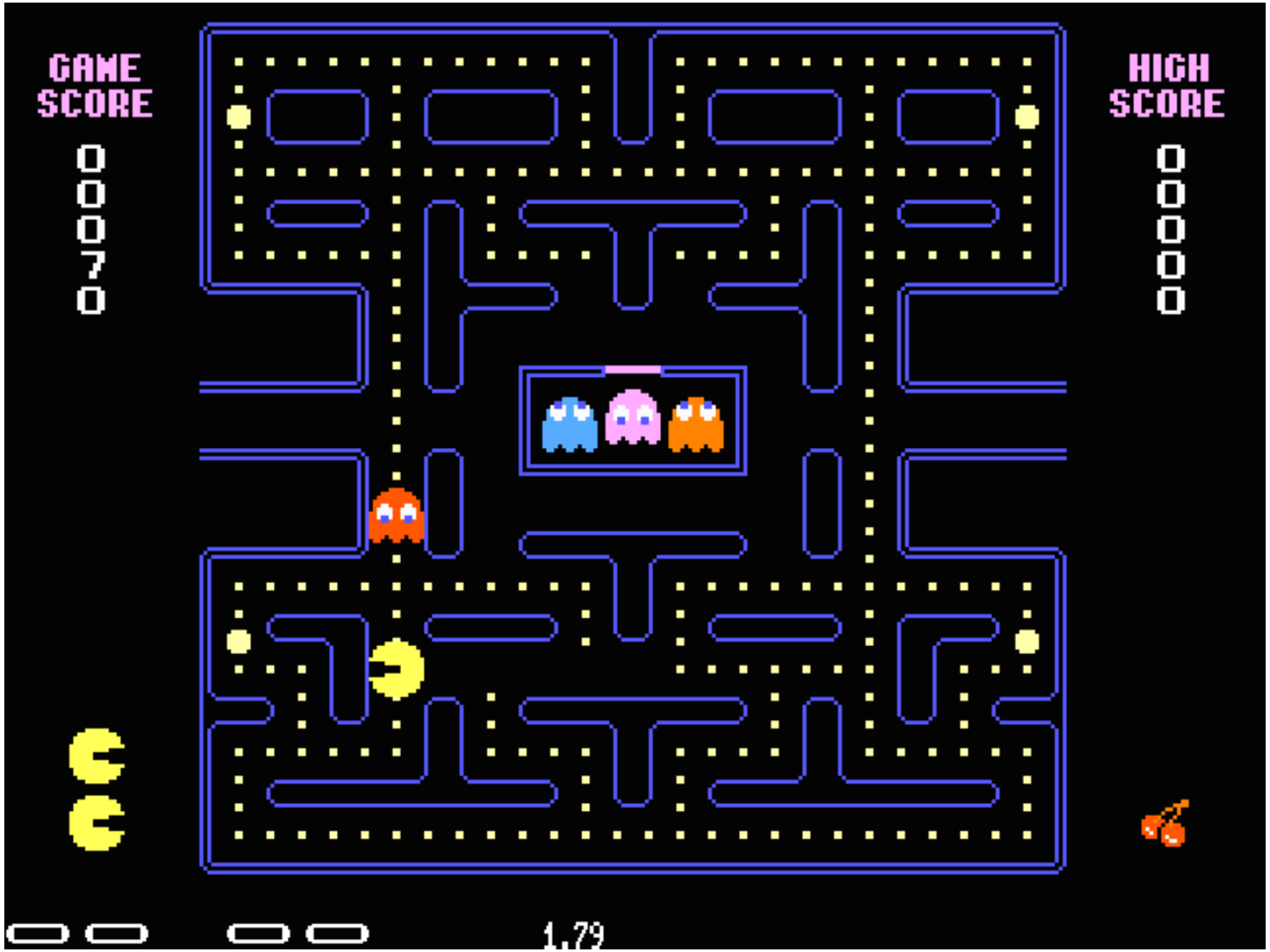
Introduction to GPU Computing

What is a GPU?

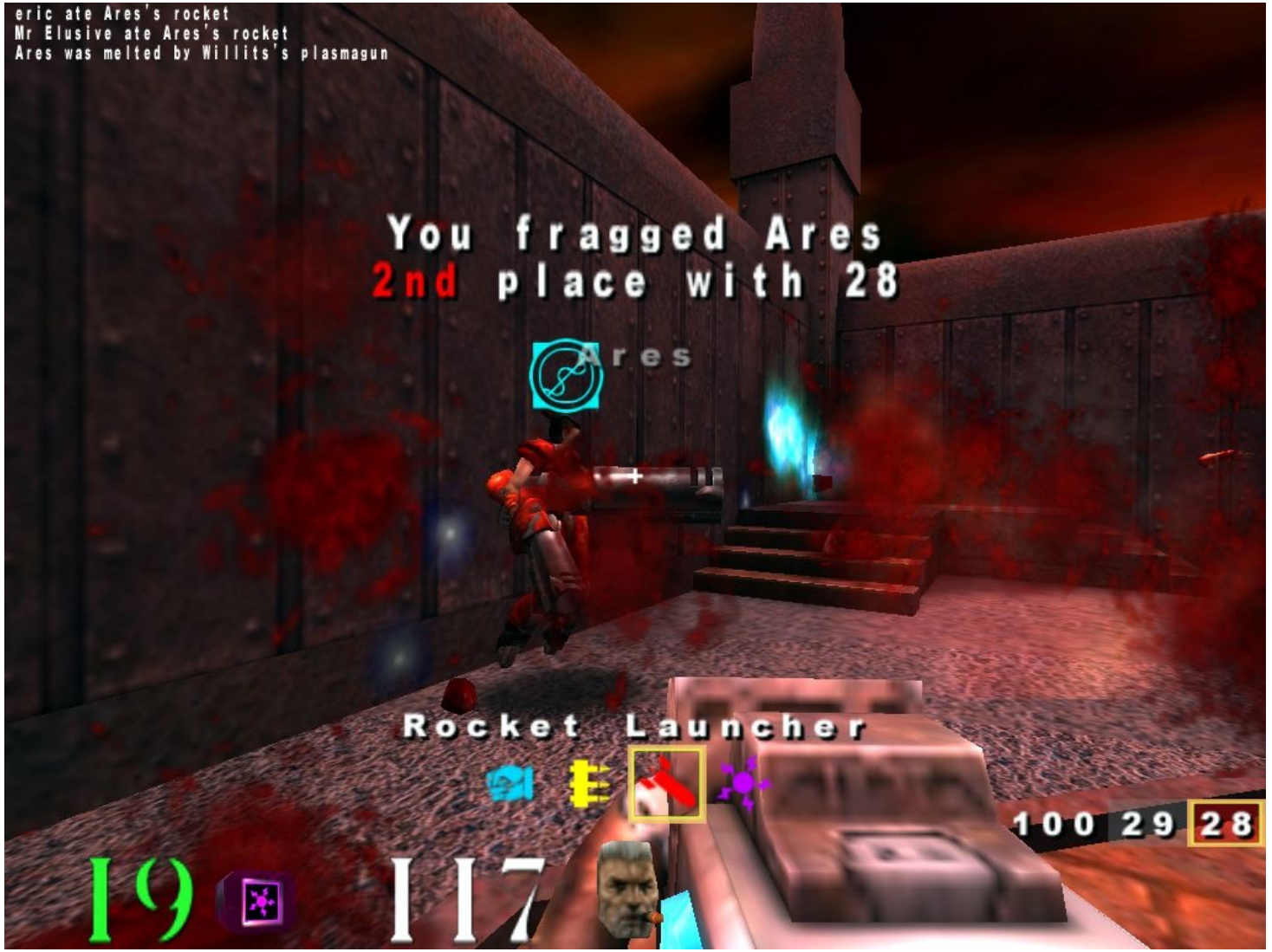
- Graphics Processing Unit: the computing chip on a graphics card
- GPGPU: General Purpose computing on GPUs



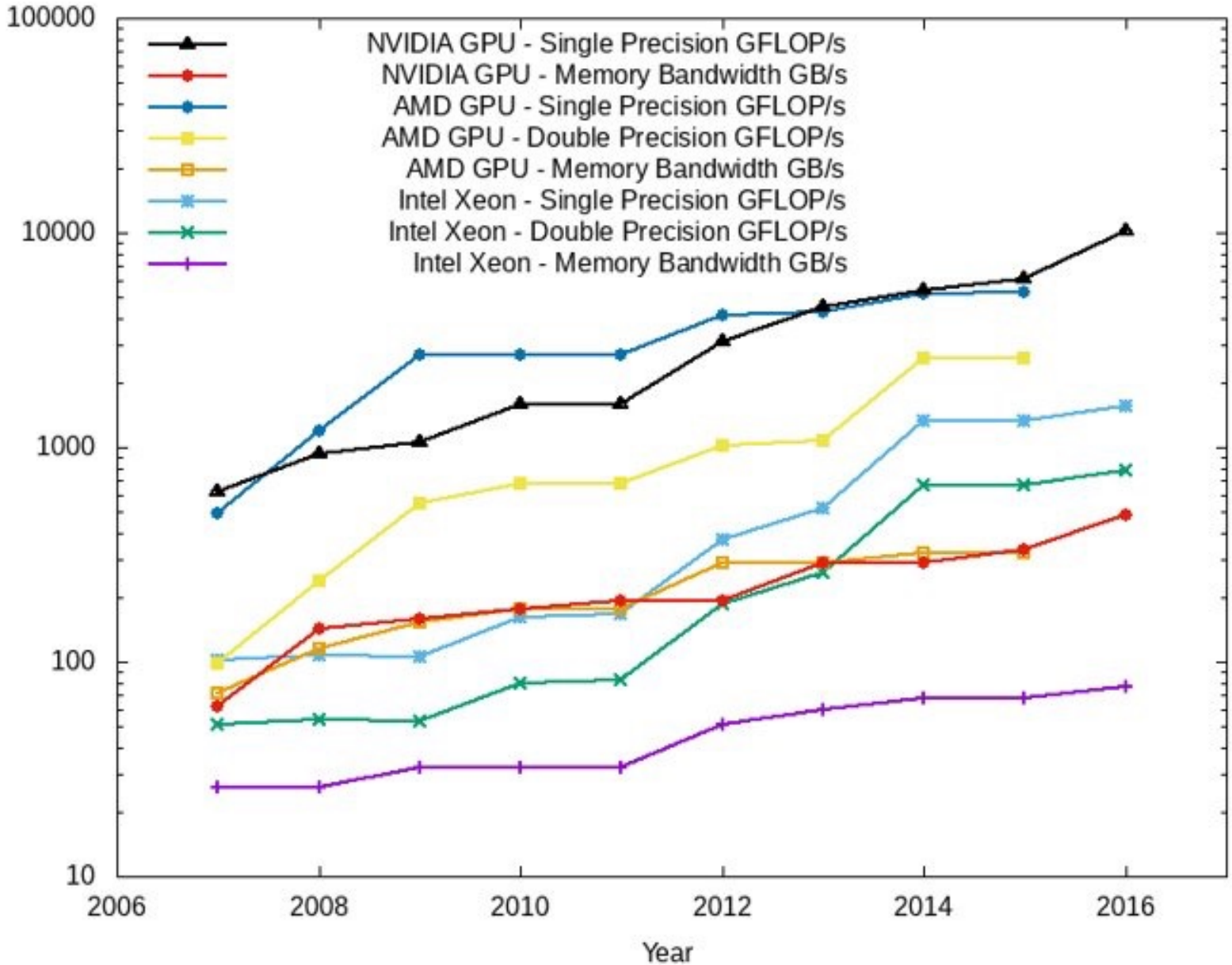
Graphics in 1980



Graphics in 2000



Performance Comparison: CPUs vs GPUs



The World's Fastest Supercomputers: Oak Ridge's Frontier

- Number 1 in TOP500 list (2022/11)
  - 1.6 TFLOP/s peak
  - 1,000 cabinets
  - 21 MW power consumption
  - CPUs
    - AMD EPYC 64C 2GHz
  - GPUs
    - AMD Instinct MI250X
  - 8,730,112 cores

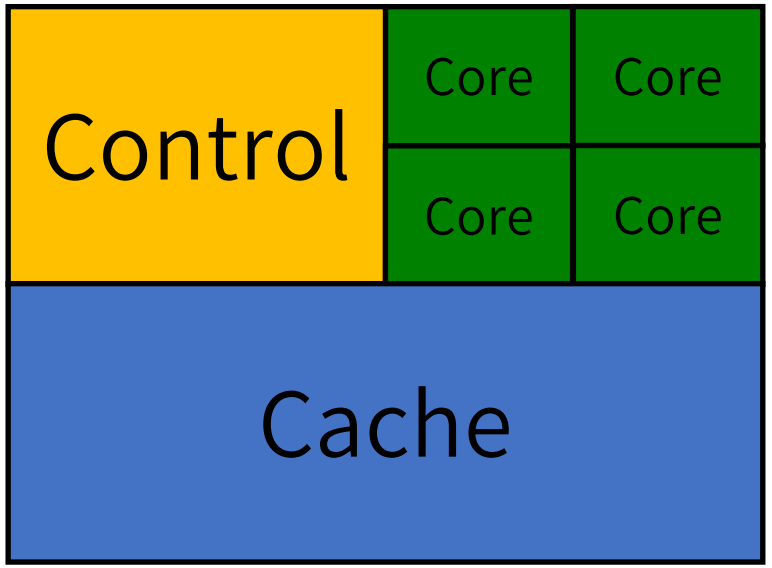
- Seven systems in top 10 using GPUs

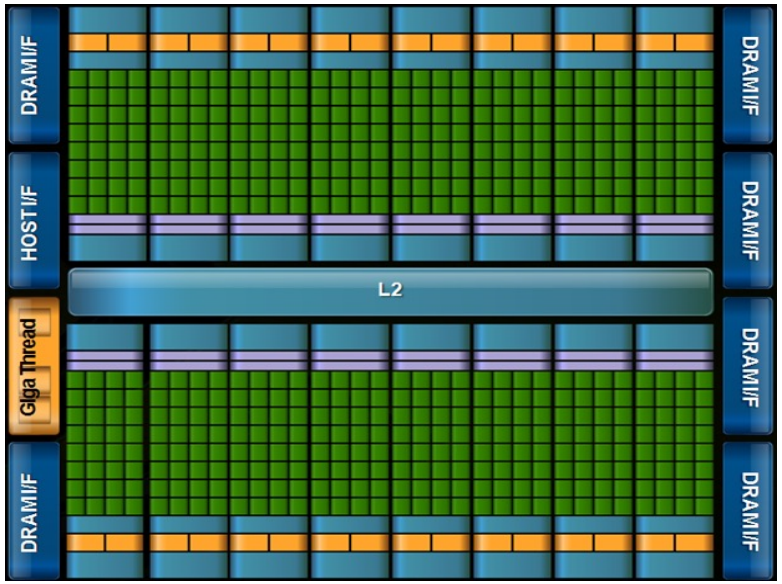
Design: CPUs vs GPUs

- Different goals produce different designs
  - GPU assumes workload 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





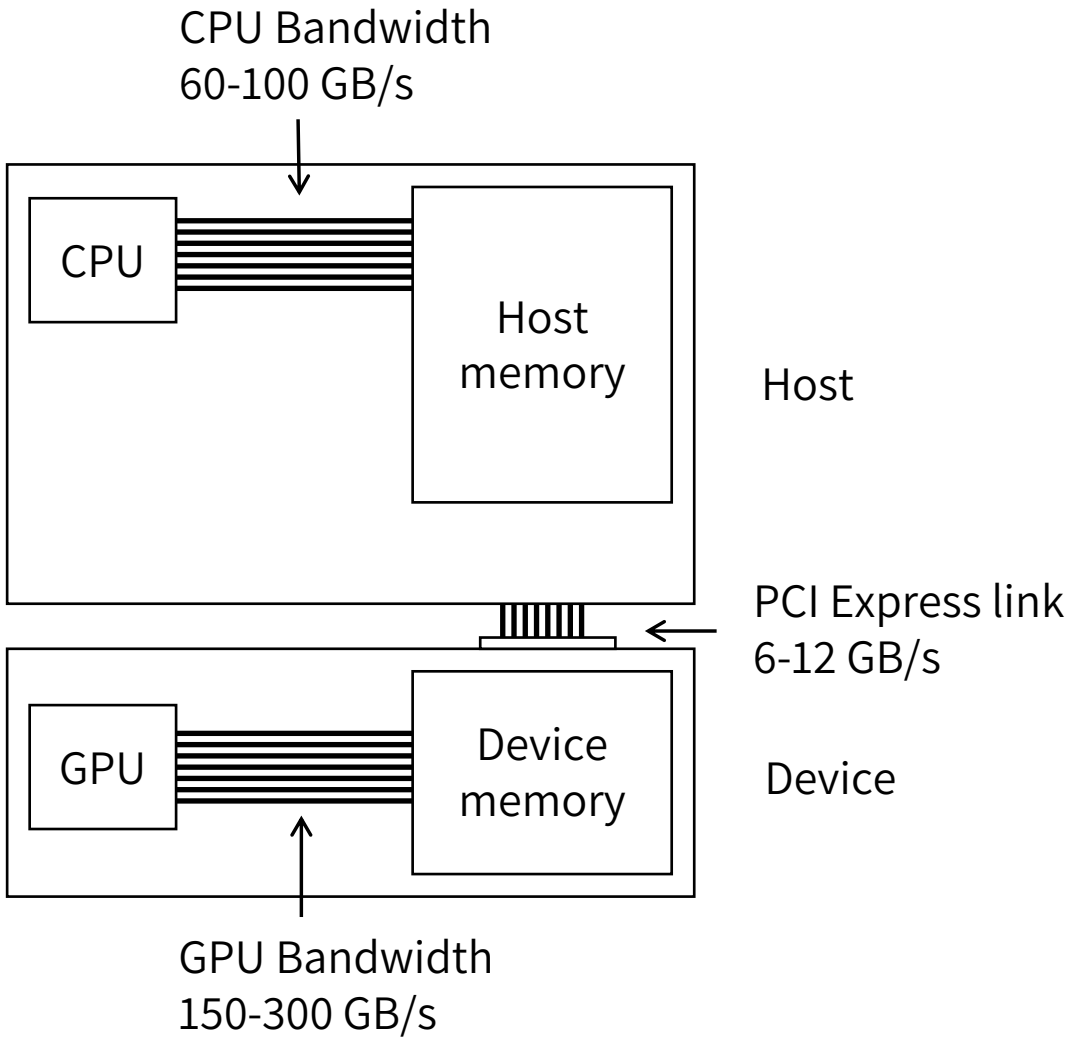
Why is GPU Programming different?

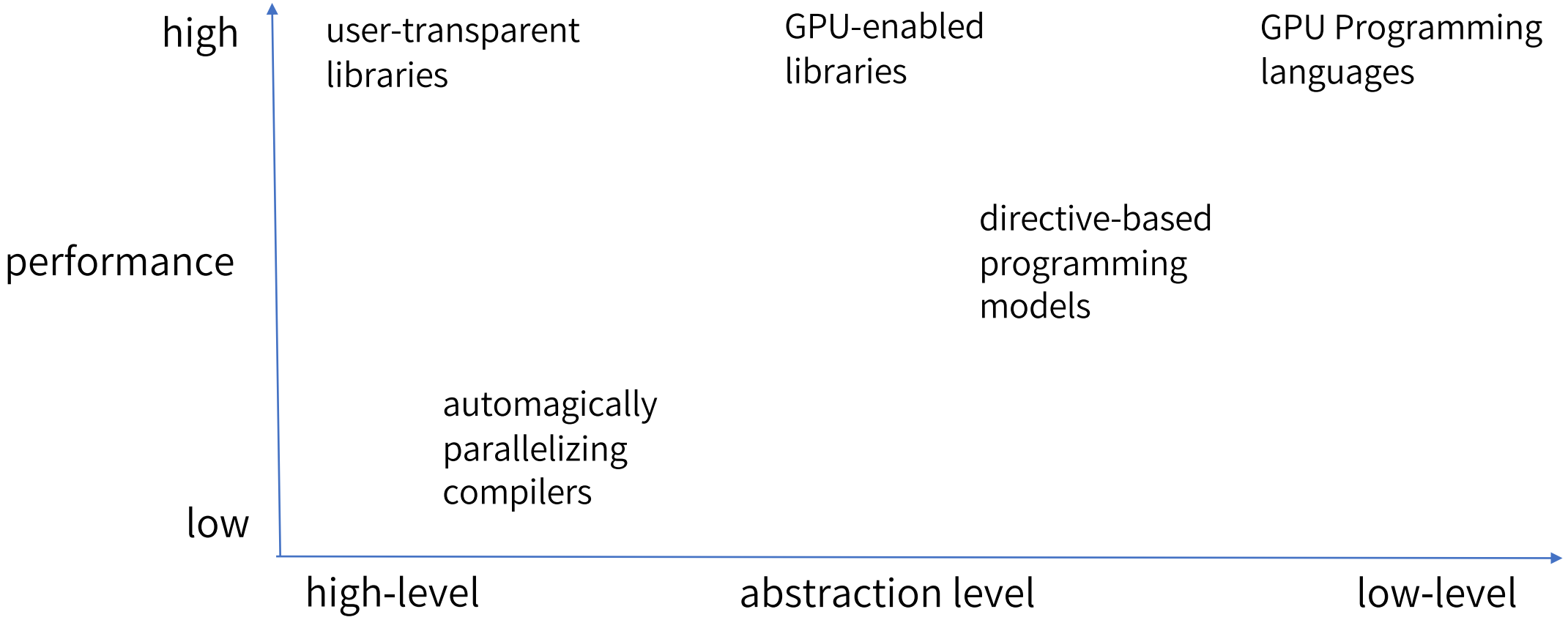
The computer architecture is very different:

- 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




high
performance
low
user-transparent libraries
GPU-enabled libraries
GPU Programming languages
directive-based programming models
automagically parallelizing compilers
high-level
abstraction level
low-level

netherlands eScience center

Introduction to GPU Programming

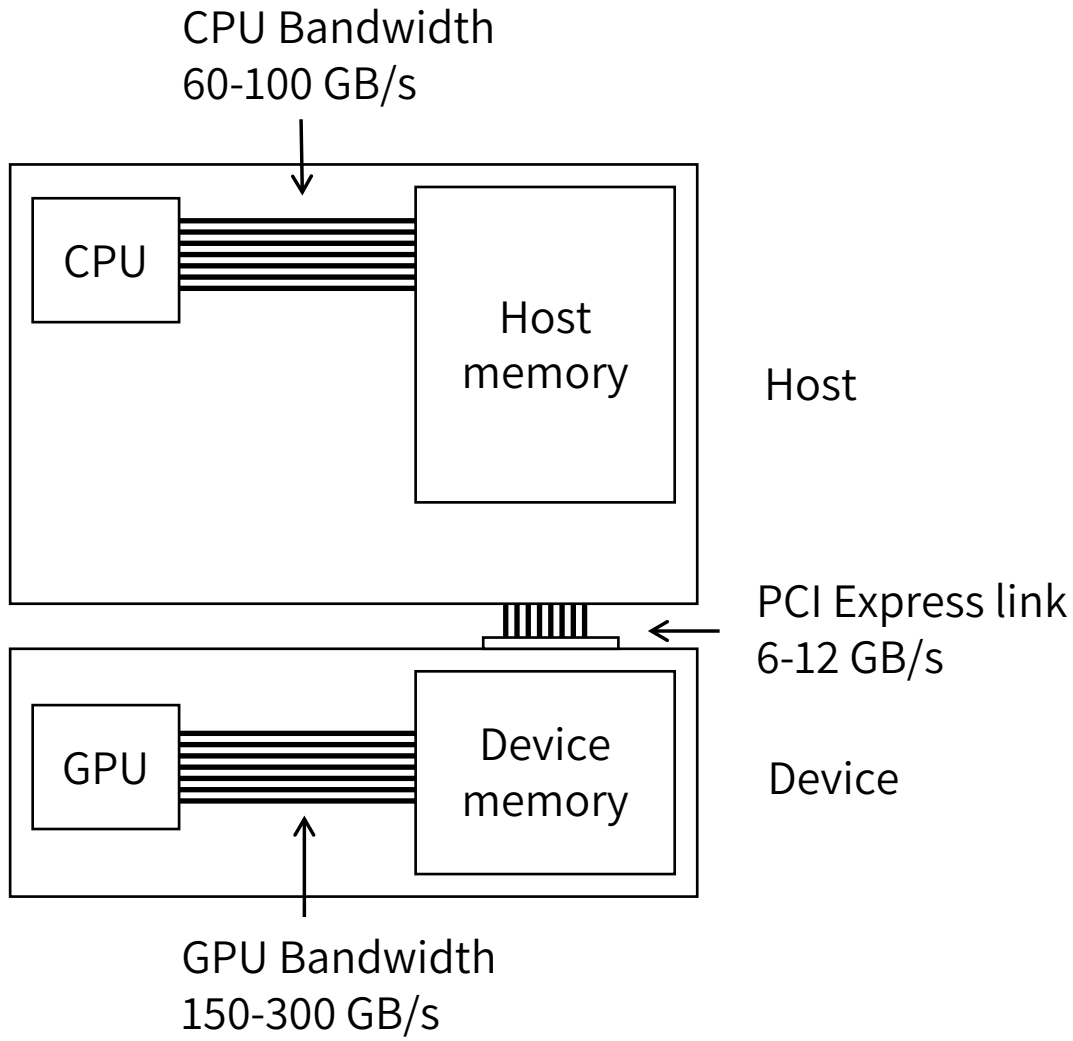
Why is GPU Programming different?

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 Programs consists of a *host* (CPU) and a *device* (GPU) part

The host part manages:
- Both host and device memory
- Data transfers between host and device memory
- Starting device *kernels* (functions on the device)

The device part consist of kernels, that:
- Are executed by huge amounts of parallel threads at the same time
- Divide the data-parallel workload among these threads
- Switches execution between groups of threads to hide memory latency

For the host code:

- Several language bindings for GPU Programming exist:
  - C/C++: CUDA, OpenCL, HIP, SYCL
  - Python: PyCuda, PyOpenCL, CuPy
  - Java: JCuda and JOCL
  - Fortran: CudaFortran
  - Matlab: MexCuda (using mexfiles)

For the device code:

- Mainly three options:
  - Write your own kernels in CUDA or OpenCL
  - Use GPU-enabled libraries (kernels written by someone else)
  - GPU Code generators (kernels written by compilers)

- There are many code optimizations that can be parameterized:
  - The number of threads per thread block in each dimension
  - Loop unrolling factors
  - The number of items processed per thread
  - The total work per thread block
  - Different schemes for using shared memory
  - Different parallelization schemes

- Optimizing GPU code is finding the best performing combination for all these parameters

- Auto-tuners are used to automate the search process

Managing GPU Memory

- GPU memory is typically smaller than host memory (12GB vs 64GB)
- Multiple GPUs each have their own device memory space
- Data copied to the GPU may become stale on the host

- Transferring data to the GPU is expensive (because of the relatively low PCIe bandwidth, better with NVLink)

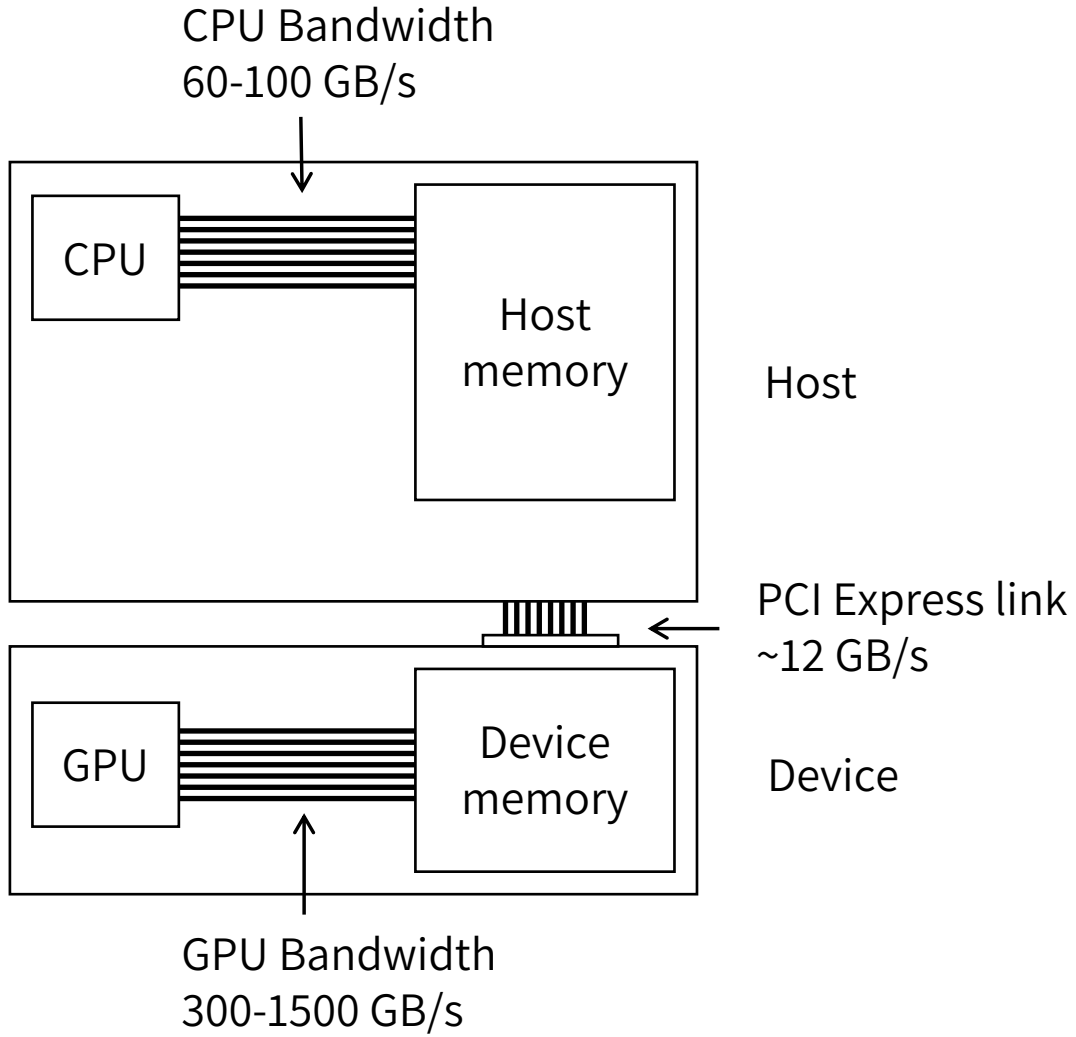
- In general it’s best to keep working on transferred data for as long as possible

- It’s possible to overlap data transfers with GPU computations and data transfers in the opposite direction

Summary

Main differences normal and GPU programming:

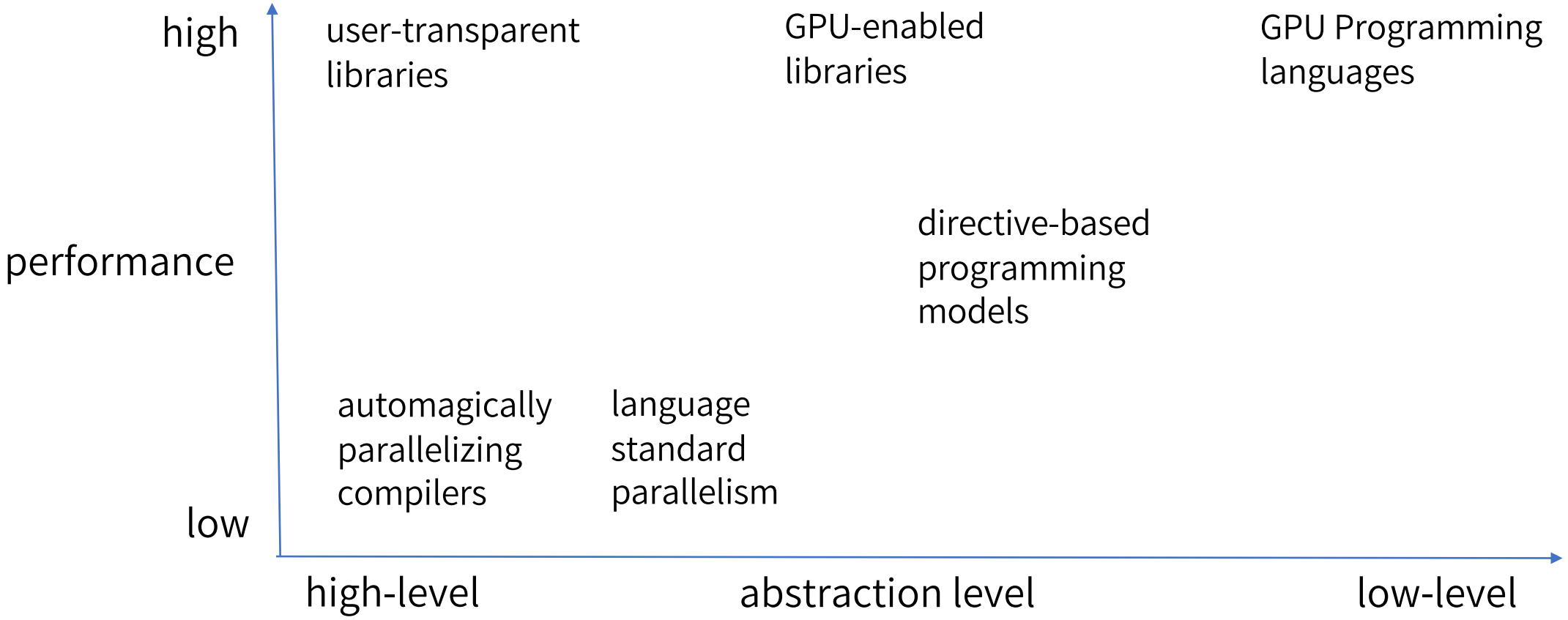
1. Algorithms need to be parallelized and mapped to the hardware
2. Requires software to be rewritten in specialized programming language
3. Optimizing for compute performance requires knowledge about hardware
4. Must deal with separate memory space, limited bandwidth between host and device memory

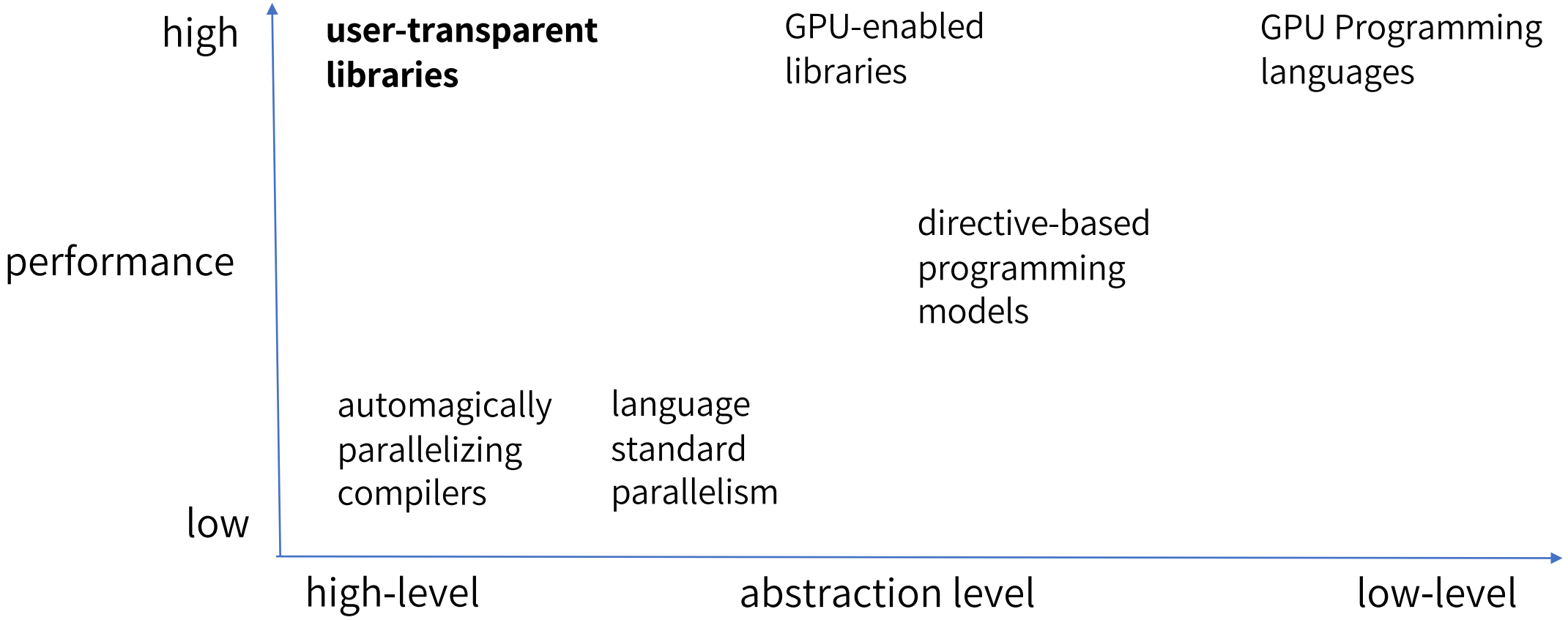


netherlands eScience center

Overview of GPU Programming models

GPU Programming techniques

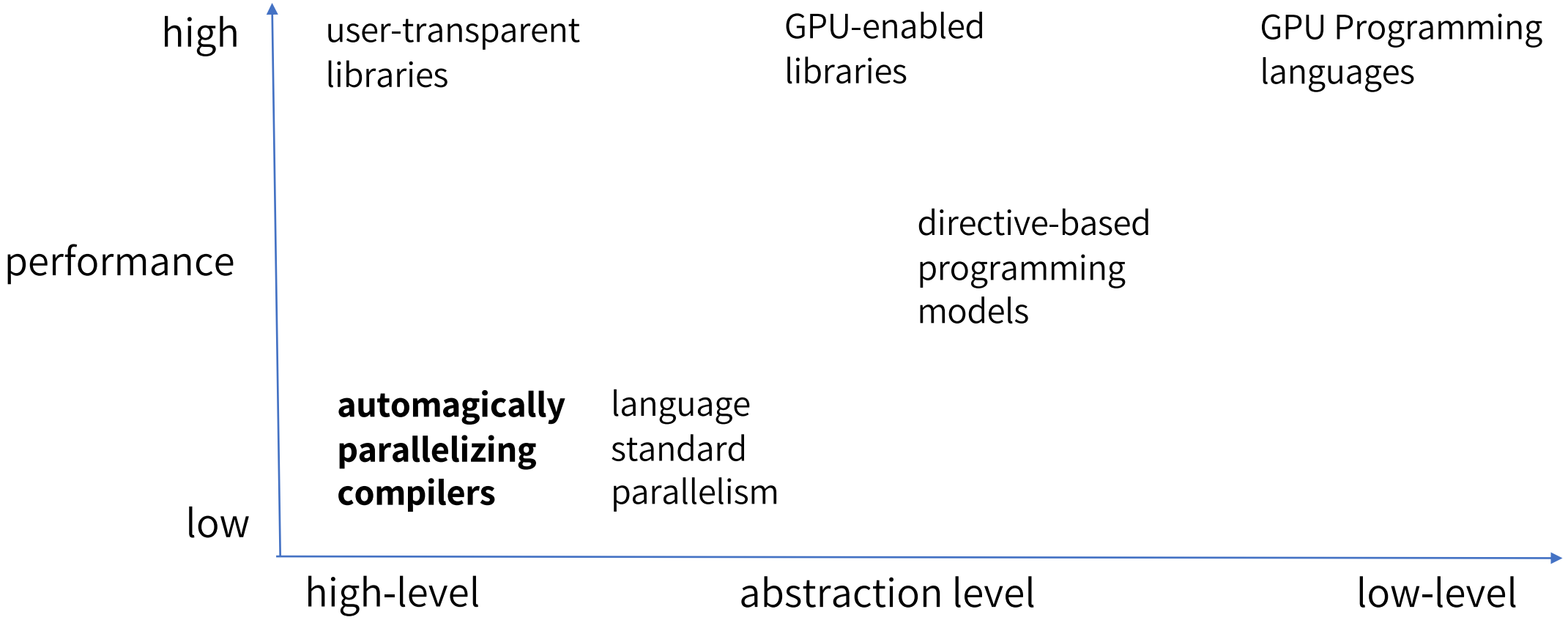



high
user-transparent libraries
GPU-enabled libraries
GPU Programming languages
performance
directive-based programming models
automagically parallelizing compilers
language standard parallelism
low
high-level
abstraction level
low-level

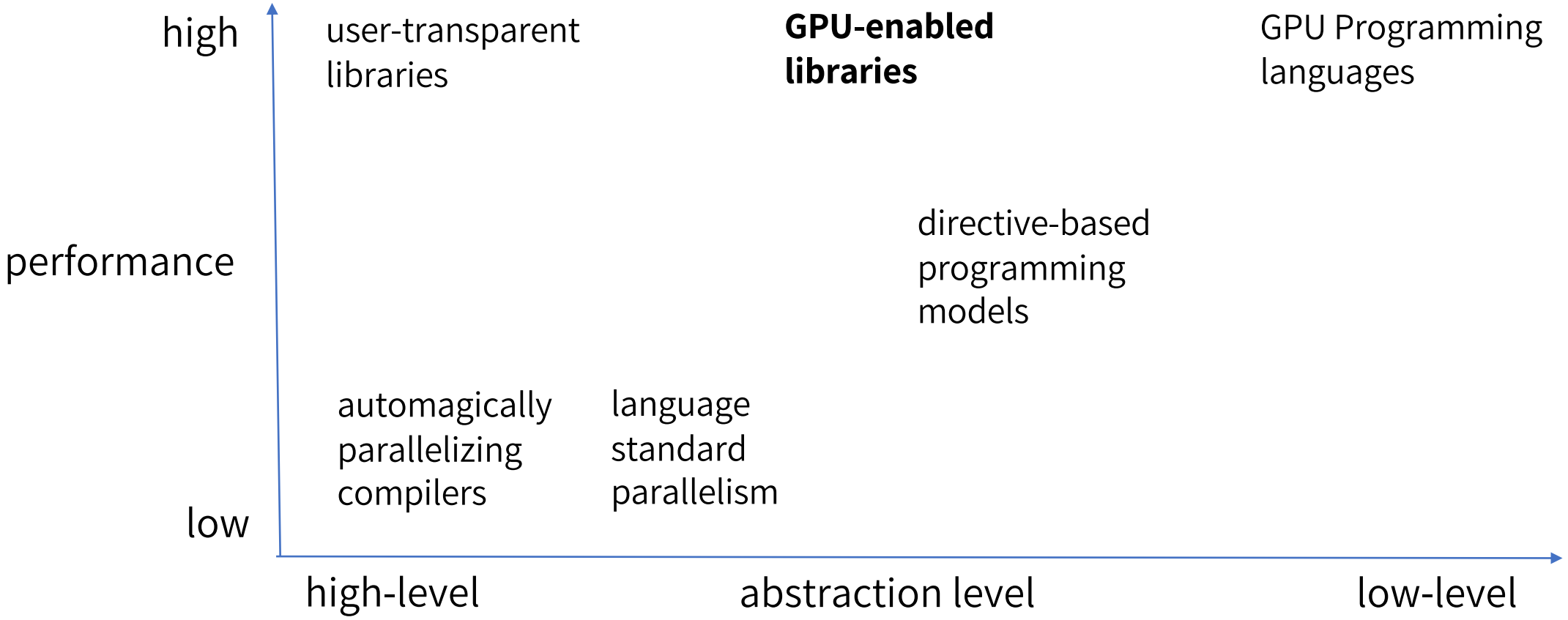
User-Transparent libraries

- Act as drop-in replacements for CPU libraries
- There aren’t that many, and their application is often limited

- Difficult to build:
  - Library must maintain state, any init() or destroy() method already breaks transparency
  - Library designer must decide how to manage GPU memory
- Difficult to use:
  - Optimizing application performance is hard when you don’t know what happens inside the library


high
user-transparent libraries
GPU-enabled libraries
GPU Programming languages
performance
directive-based programming models
automagically parallelizing compilers
language standard parallelism
low
high-level
abstraction level
low-level

GPU Programming techniques



GPU-Enabled Libraries

- Generally, the user is responsible for managing GPU memory
- Often use specialized objects that represents data in GPU memory
- Easy access to highly-optimized and tuned GPU routines

- Either focused on specific functionality or offering a ‘GPU Array’-like datatype

Function-Oriented Libraries

- Fast Fourier Transforms: cuFFT, clFFT, hipFFT, rocFFT, vkFFT
- BLAS (linear algebra): cuBLAS, clBlas, rocBlas, clBlast (auto-tuned), SYCL-BLAS
- Random number generation: cuRAND, rocRAND
- Sparse matrix operations: cuSparse, hipSparse
- Deep neural networks: cuDNN, OpenCV DNN, SYCL-DNN

- Practically all of these can be used directly from C++, many have Python bindings, bindings for other languages are not that commonly available or only supported by relatively small open-source projects

Datatype-Oriented Libraries

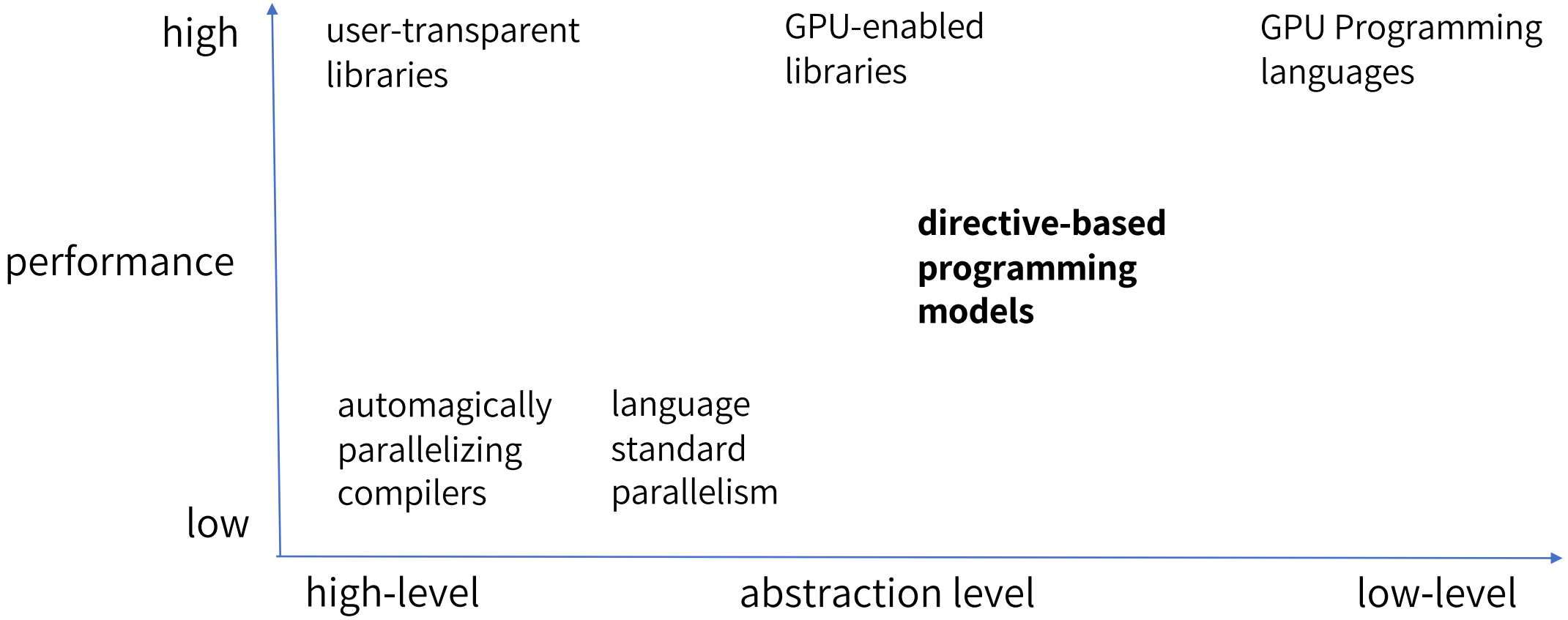
- Matlab: gpuArray
  - Provides access to many operations using cuBLAS, cuFFT underneath
  - Also JIT-compiles groups of pointwise array operations into CUDA kernels
- Python
  - CuPy: A NumPy-compatible array library accelerated by CUDA
    - Includes functionality for compiling ‘raw’ CUDA kernels
    - Includes bindings to cuBLAS, cuFFT, cuDNN, …
  - PyTorch: Open source machine learning framework
    - Includes Tensor data type with CUDA backend
    - Can be used to interface cuBLAS, cuRAND, cuFFT, cuDNN, cuSPARSE, …
  - Numba: Open source JIT compiler for Python/Numpy code
    - Compiles to CUDA or ROCm
    - Includes Python bindings to CUDA
  - cuDF: A Pandas-like GPU DataFrame library
    - Supports operations for loading, joining, aggregating, filtering, and otherwise manipulating data
    - Integrates with Dask for distributed and out-of-core computations
- ArrayFire, can be used from C, Rust, or Python
    - Includes functions for many image processing, linear algebra, and machine learning operations

CUDA SDK Samples include many simple example codes to illustrate how to use cuBLAS, cuFFT, and so on: https://developer.nvidia.com/cuda-code-samples

Examples on how to use libraries from AMD's ROCm platform are included in the documentation: https://rocmdocs.amd.com

The datatype-oriented libraries often include their own memory managers, which can be great but sometimes complicates interoperability.

If you are not using C++ or Python, it is generally possible to write a small C++ code that calls the library function and can be called from another language, e.g. Fortran.


high
user-transparent libraries
GPU-enabled libraries
GPU Programming languages
performance
directive-based programming models
automagically parallelizing compilers
language standard parallelism
low
high-level
abstraction level
low-level

OpenACC and OpenMP:

- Open standards for *directives* that can be implemented by compilers
- Directives are language constructs that specify how compilers should process their input

- What does a directive look like?
  - In C: `#pragma acc` *directive-name [clauses]*
  - In Fortran: `!$acc` *directive-name [clauses]*

- Example:
  - `#pragma acc parallel`
    Tells the compiler that the following structured block should be executed in parallel on the current accelerator device

Pros and Cons

- Advantages:
  - Program is kept in the original language, with directives
  - Easy to get some performance improvement
  - Can serve as a gentle introduction to GPU Programming
- Drawbacks:
  - False sense of security: Directives move the responsibility for program correctness from the compiler to the user. If you say something is parallel, the compiler will parallelize it regardless of whether it actually is
  - False sense of simplicity: If you want high performance, you still need to know a great deal about (and provide device-specific parameters for) the device your code targets
  - Directives can become really numerous and can obfuscate the original program, having a separate source can arguably be cleaner
  - Accelerating a program with directives for high performance may still require changes to the original code, such as changing data layouts, reordering and merging loops, and so on

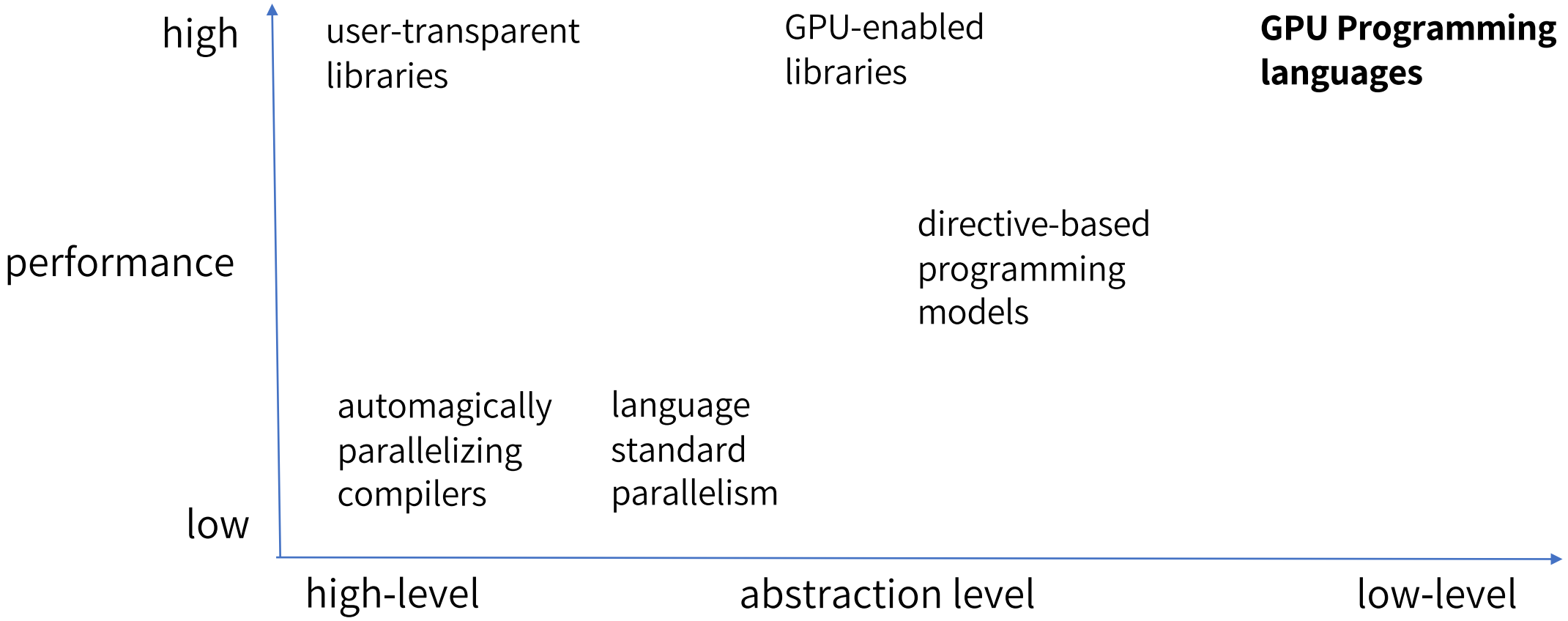
Memory management

- From the OpenACC specification:
  “In the OpenACC model, data movement between the memories can be implicit and managed by the compiler, based on directives from the programmer. However, the programmer must be aware of the potentially separate memories for many reasons, including but not limited to:”
  - Memory bandwidth between host memory and device memory
  - Device memory can be smaller than host memory
  - Pointers to host memory can not be dereferenced on the device and vice versa

- So, while it’s *“implicit and managed by the compiler”* you must specify all information in a similar way as you would with CUDA or OpenCL, only using directives instead of function calls.

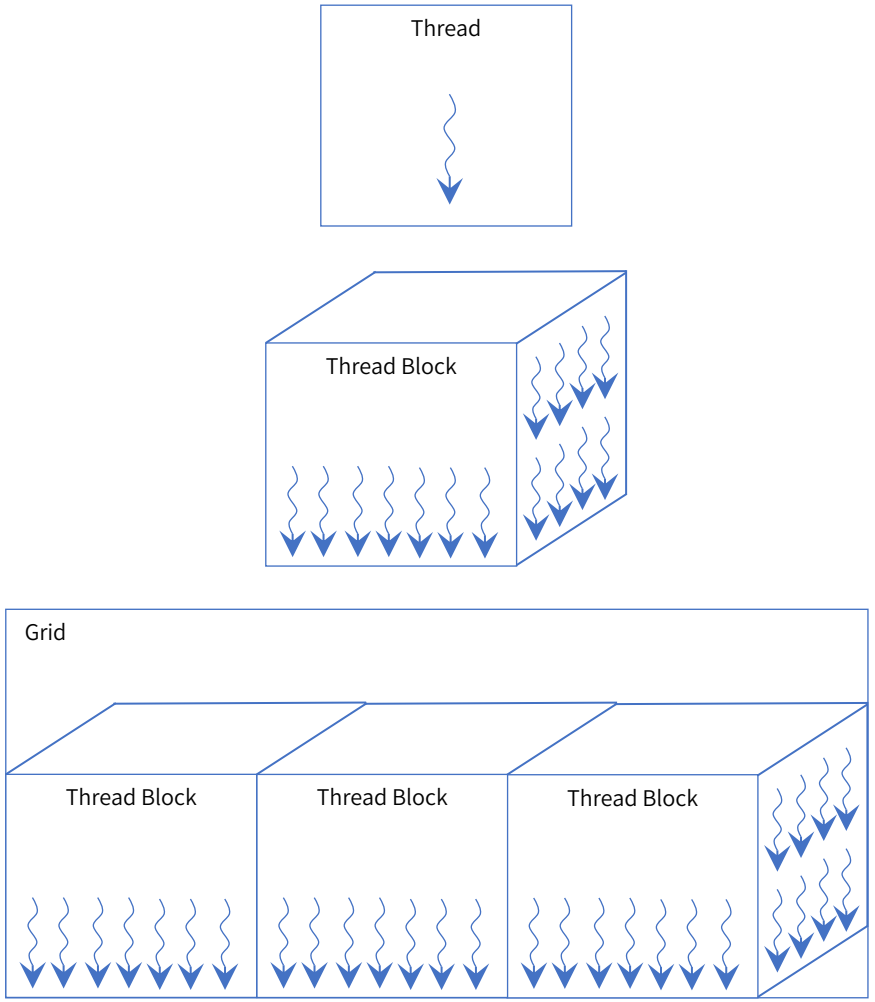
Compiler support

- Commercial compilers:
  - NVC (part of Nvidia HPC SDK), Cray, and CAPS
- ‘Research’ compilers (developed by universities):
  - OpenUH, OpenARC, accULL, …
- Open compilers:
  - OpenACC 2.6 is supported in GCC as of version 10
  - OpenMP 4.5 is supported in GCC as of version 11


high
user-transparent libraries
GPU-enabled libraries
GPU Programming languages
performance
directive-based programming models
automagically parallelizing compilers
language standard parallelism
low
high-level
abstraction level
low-level

- CUDA
- OpenCL
- HIP
- SYCL

- Domain-specific languages (DSLs): Futhark, Halide, …

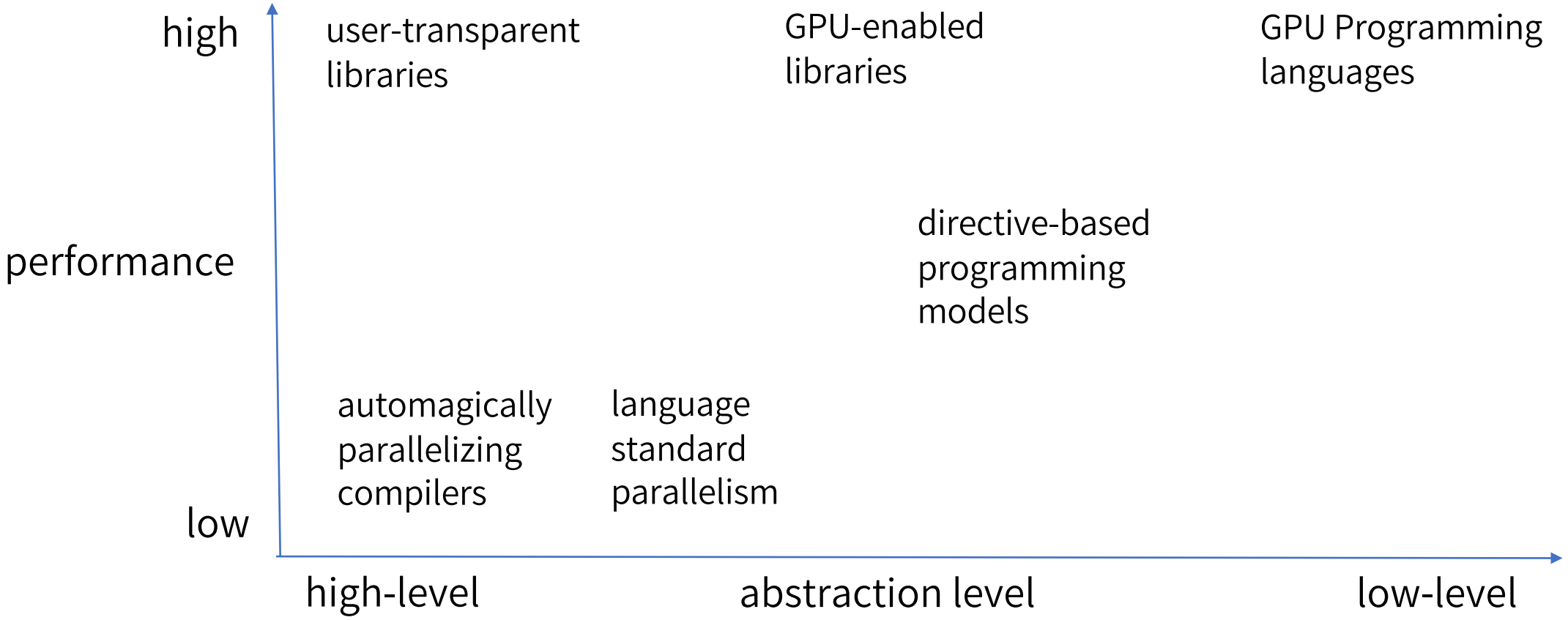


Registers

Shared memory

Global memory
Constant memory

GPU Programming techniques: Summary



netherlands eScience center

Introduction to CUDA Programming

Before we start:

- We will explain the CUDA **Programming model**
- We'll try to avoid talking about the hardware for now
- For the moment, please **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

- Parallelizing a computation sometimes requires to rethink algorithms, for example:

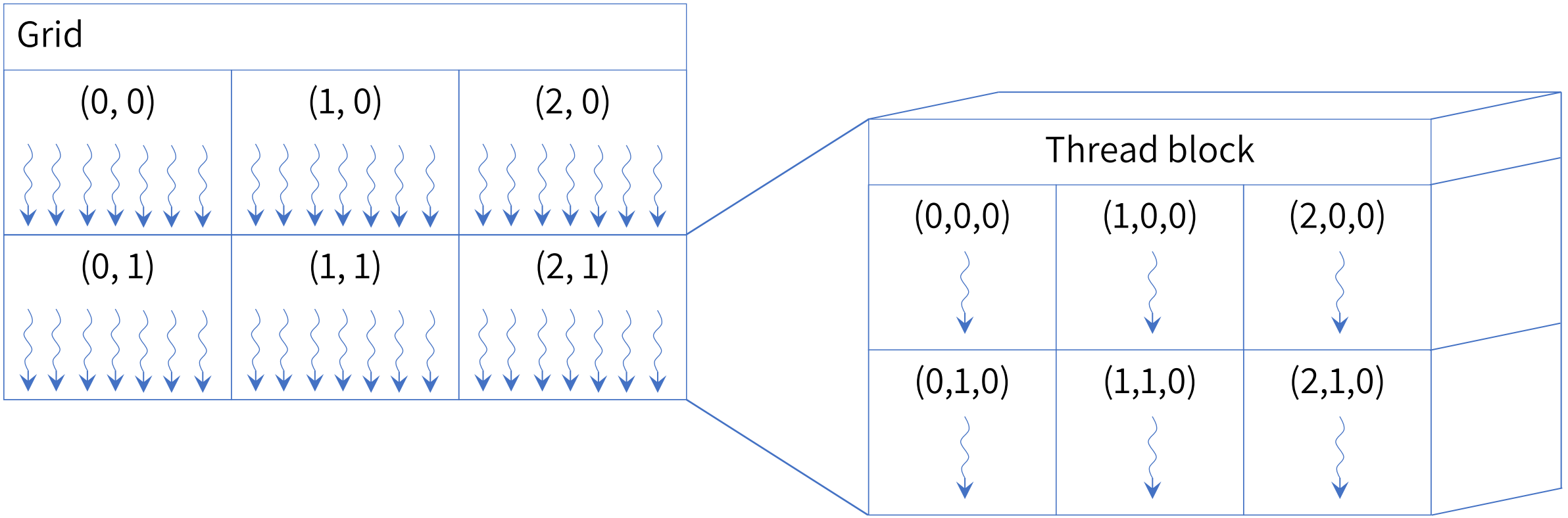
```
//some sort of stencil, performs 1 read for every 3 writes
for (int i=1; i<N-1; i++) {
  double my_a = a[i];
  a_new[i-1] += 0.25*my_a;
  a_new[i] += 0.5*my_a;
  a_new[i+1] += 0.25*my_a;
}

//same, but with 3 reads for every 1 write
for (int i=1; i<N-1; i++) {
  a_new[i] = 0.25*a[i-1] + 0.5*a[i] + 0.25*a[i+1];
}
```

The latter is much easier to parallelize because it avoids concurrent writes to the same memory locations

Kernel programming language

- The programming language for kernels is CUDA. It's mostly C/C++, but with some additions and limitations
- Additions:
  - Function qualifiers `__global__`, `__device__`, and `__host__` can be used to declare a function as being a kernel, a device function, or a host function
  - Kernel and device functions have built-in variables, like `threadIdx.xyz` or `blockIdx.xyz`
  - Memory qualifiers `__constant__` and `__shared__` can be used to declare a variable to reside in special memory spaces
  - Many intrinsic functions, e.g. `__sincosf()`, exist to use special function units in the hardware
- Limitations:
  - You cannot use any existing C functions, only functions with the `__device__` qualifier can be called from kernels.
  - A lot of standard C library functionality is not present, for example there is no `malloc()`, and for the first couple of years of CUDA there wasn't even a `printf()` function

- 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 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 in C

- 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();
```

- 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
threads = (x, y, z)
grid = (x, y, z)

# launch the kernel
vector_add([c, a, b], block=threads, grid=grid)

# wait for the kernel to complete
context.synchronize()
```


Coffee break

netherlands eScience center

First Hands-on

Setup hands-on session

- Login on DAS-5 (fs0.das5.cs.vu.nl)
- Execute (recommended to add both to your .bashrc):
  - `module load cuda11.1`
  - `alias gpurun="srun -N 1 -C TitanX --gres=gpu:1"`

- Installing Python 3
  - Follow these steps:
    - `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 jupyter`
  - `pip install pycuda` (make sure you've typed `module load cuda11.1` first)
  - `pip install kernel_tuner`

- Follow the guide to use Jupyter notebooks on DAS-5
  - https://guide.esciencecenter.nl/#/nlesc_specific/e-infrastructure/das5

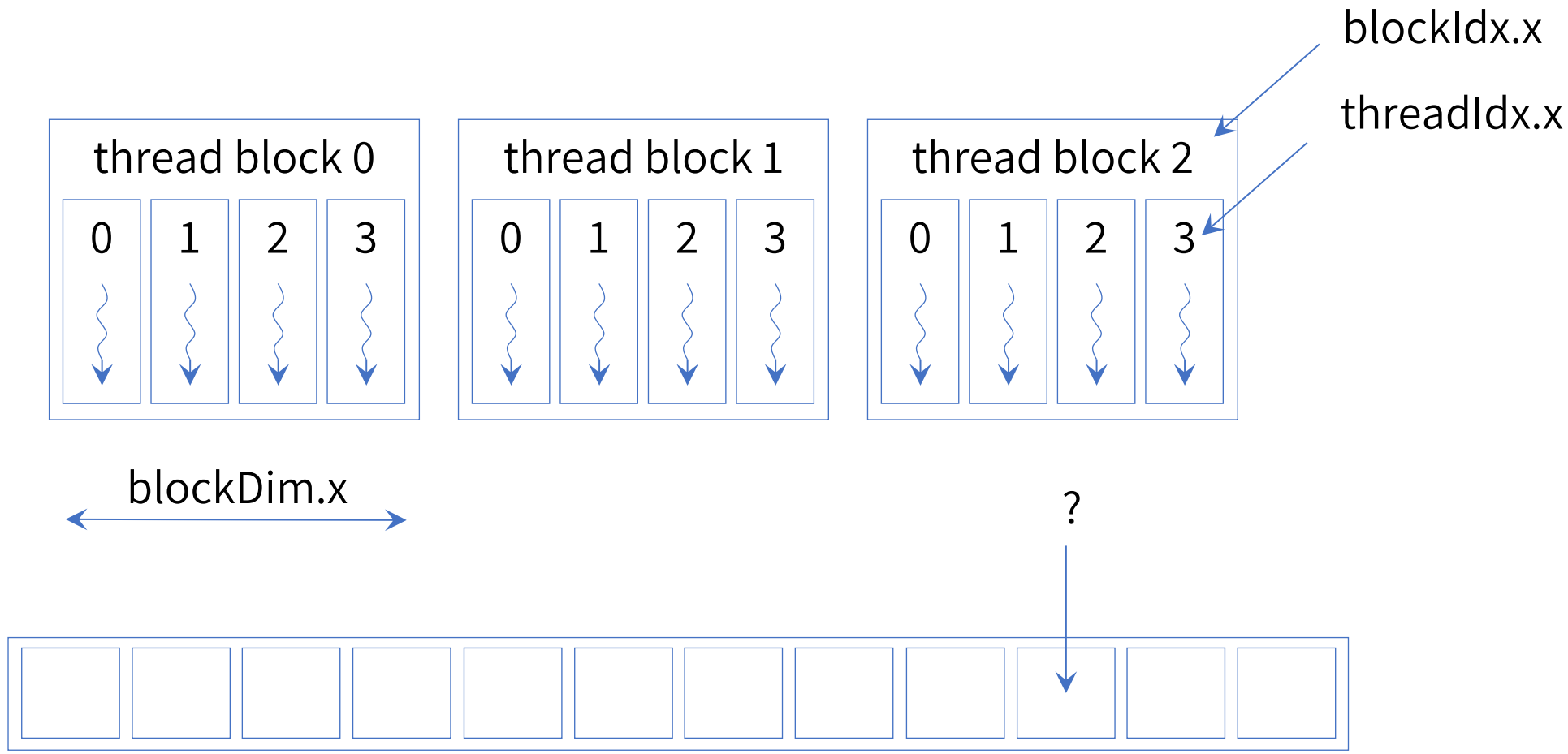
First Hands-on Session: Vector Add

- Select the notebook for the first exercise
  - https://github.com/benvanwerkhoven/gpu-course/blob/master/notebooks/vector_add.ipynb

- 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://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
?

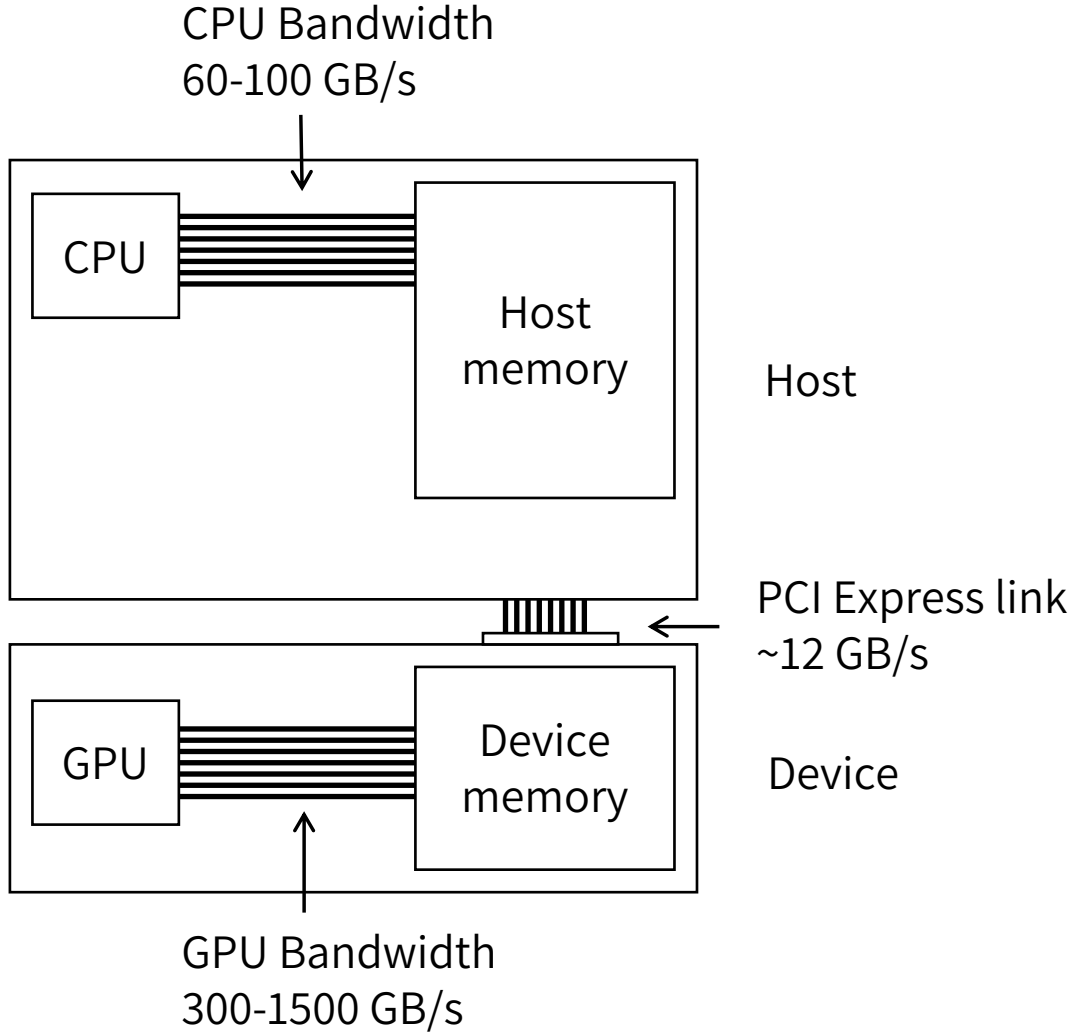
netherlands eScience center

Host code

Host code

- Every CUDA program has a host and a device part

- The host code runs on the CPU and is responsible for:
  - GPU memory management
  - Transferring data between host and device memories
  - Launching GPU kernels (asynchronously)
  - Synchronization



Runtime vs Driver API

- CUDA has two APIs, with largely overlapping functionality
- The Runtime API:
  - Easier to use for simple applications, several resources are implicitly initialized
  - Allows device kernels to be launched like function calls
- The Driver API:
  - Largely the same, but more verbose, more fine-grained control
  - Interoperable with NVRTC (Nvidia Runtime Compiler)

Different language bindings

- The main CUDA API offered by Nvidia is in C, with some parts in C++

- Several Python bindings exist: PyCUDA, CuPy, Torch, Numba, …
- In 2021, Nvidia publicly released cuda-python, a Cython binding of the C API

Error handling

- CUDA Runtime API function return a `cudaError_t` value that can be passed to `cudaGetErrorString(cudaError_t error)` to get the error message as a string

- Kernel launches in the runtime API do not return anything, so use:
  - `cudaGetLastError(void)` or `cudaPeekAtLastError(void)` to retrieve the last error

- You should always check the return value on every call to the CUDA Runtime API
  - It makes the code more verbose but helps in catching bugs

GPU Memory management

- The memory used on the GPU is managed by the host

- GPU memory can be allocated and freed using:
  - `cudaError_t cudaMalloc(void** devPtr, size_t size)`
  - `cudaError_t cudaFree(void* devPtr)`

- GPU kernels usually only read from and write to GPU memory
- This means, all kernel inputs and outputs must be stored in GPU memory

Moving data around

- The main function to move data in and out of the GPU is:
  - `cudaError_t cudaMemcpy(void* dst, const void* src, size_t count, cudaMemcpyKind kind)`

- `cudaMemcpyKind` specifies the direction of the copy:
  - `cudaMemcpyHostToHost`
  - `cudaMemcpyHostToDevice`
  - `cudaMemcpyDeviceToHost`
  - `cudaMemcpyDeviceToDevice`

CUDA managed memory

- `cudaError_t cudaMallocManaged(void** devPtr, size_t size, unsigned int  flags = cudaMemAttachGlobal )`
- Allocates memory that will be managed automatically
  - Manual transfers from and to the device are not necessary
- Advantage is programming simplicity, performance may not be optimal
- As of Pascal architecture:
  - GPU memory can be oversubscribed
  - Data is migrated using page-faults as needed
  - System-wide (CPU & GPU) atomics are supported
- Cleanup using `cudaFree()`

- 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();
```

Starting a Kernel In Python (using PyCUDA)

- 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
threads = (x, y, z)
grid = (x, y, z)

# launch the kernel
vector_add([c, a, b], block=threads, grid=grid)

# wait for the kernel to complete
context.synchronize()
```

Starting a Kernel In Python (using cuda-python)

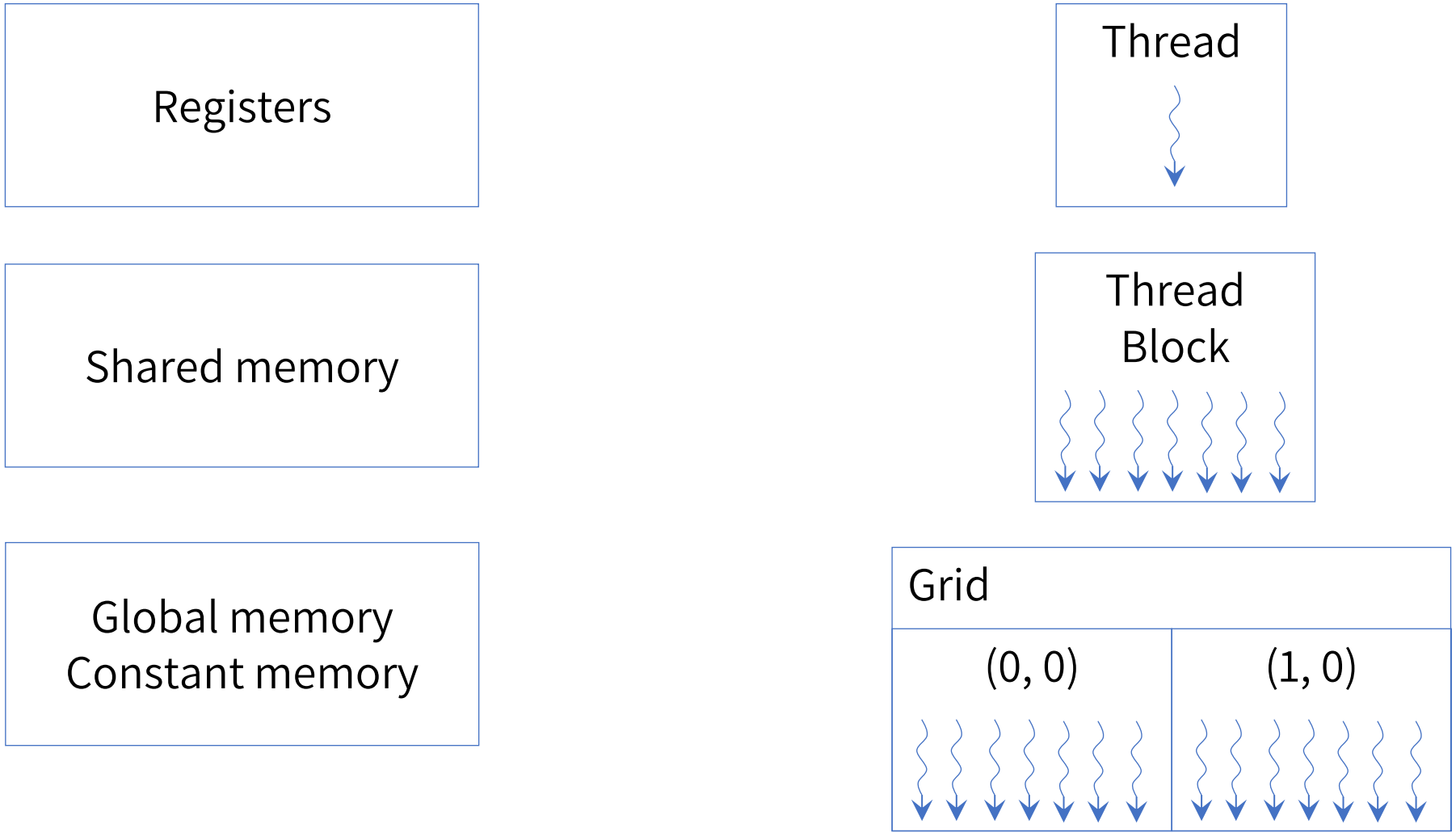
- To launch a kernel using cuda-python you must use the driver API

```
# create variables to hold grid and thread block dimensions
threads = (x, y, z)
grid = (x, y, z)

# launch the kernel
err = cuda.cuLaunchKernel(kernel, grid[0], grid[1], grid[2],
                          threads[0], threads[1], threads[2],
                          0, 0, kernel_args, 0)

# wait for the kernel to complete
err = cudart.cudaDeviceSynchronize()
```

netherlands eScience center

Cuda Memories (part 1)

CUDA memory hierarchy



- 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

__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]; //j and i do not depend on 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`

netherlands eScience center

Second hands-on

- Select the notebook for the second exercise
  - https://github.com/benvanwerkhoven/gpu-course/blob/master/notebooks/pnpoly.ipynb

- Make sure you understand everything in the code, and complete the exercise!

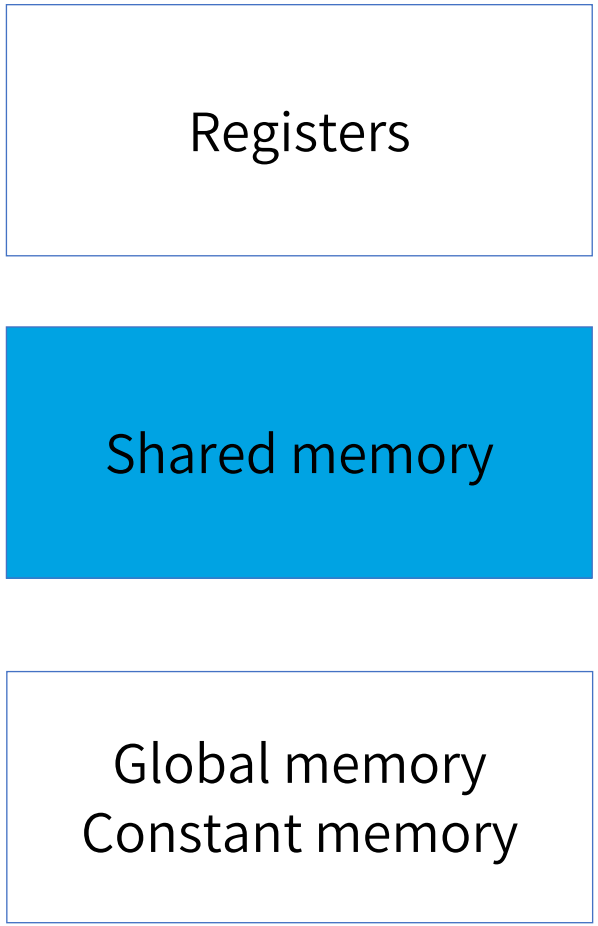
- Hints:
  - Use constant memory instead of global memory for the list of vertices
  - Python users can use `memcpy_htod()`, but need to find the symbol to copy to
  - See PyCuda documentation on `get_global`

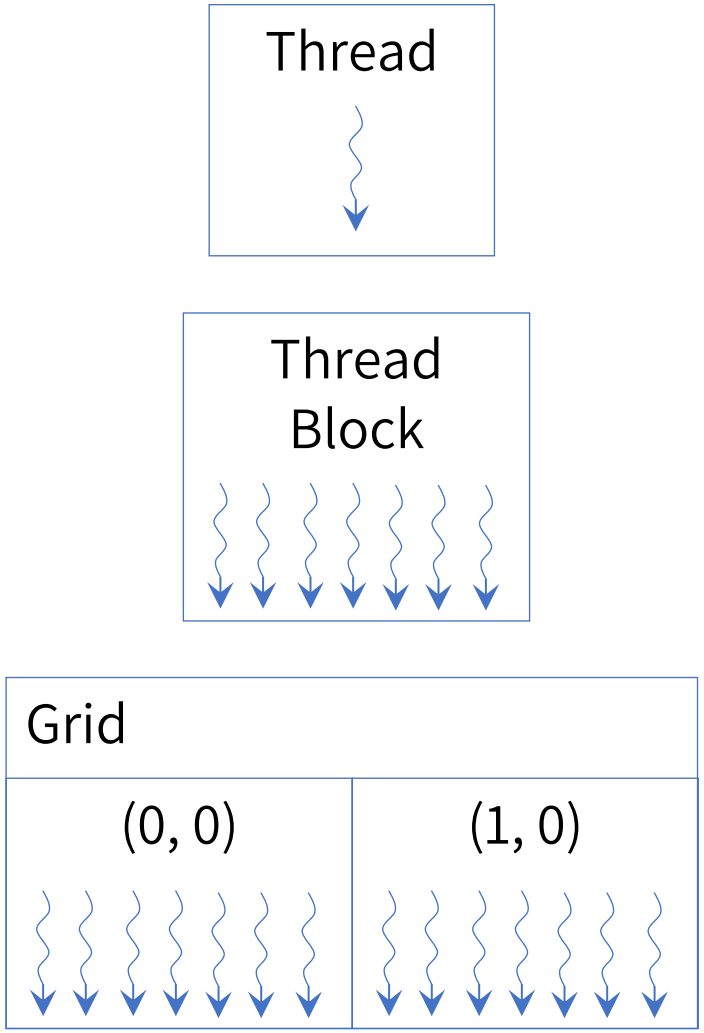

Lunch break

netherlands eScience center

Cuda Memories (part 2)

CUDA memory hierarchy





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

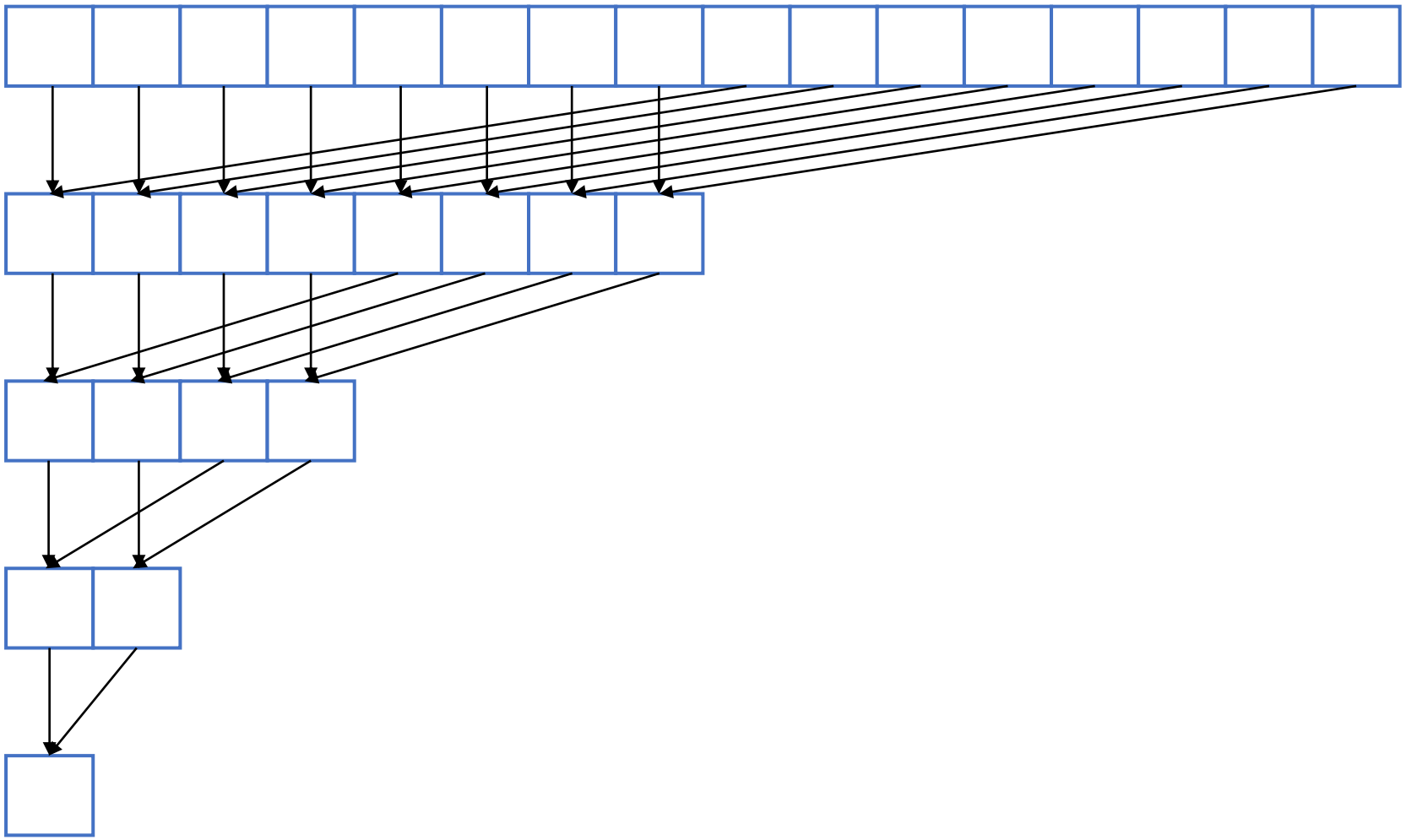
```
__global__ void transpose(int h, int w, float* output, float* input) {
    int i = threadIdx.y + blockIdx.y * block_size_y;
    int j = threadIdx.x + blockIdx.x * block_size_x;

    __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 threads in the 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
    }
}
```

netherlands eScience center

Third hands-on

- Select the notebook for the third exercise
  - https://github.com/benvanwerkhoven/gpu-course/blob/master/notebooks/reduction.ipynb
- Implement the kernel such that shared memory is used to sum the per-thread partial sums into a single per-thread block partial sum
- Make sure you understand everything in the code, and complete the exercise!
- 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



netherlands eScience center

Asynchronous processing

- The host and device(s) can perform tasks simultaneously
- The host can block CPU execution to wait for operations on the GPU to complete

- GPU operations can also be assigned to **streams**
- The host can also wait for operations in a specific stream to complete

- Streams are sequences of commands that are *sequentially consistent*
  - Commands issued in a stream will happen one after the other
  - Commands issued in different streams may execute out of order with respect to one another or concurrently

- Example use:

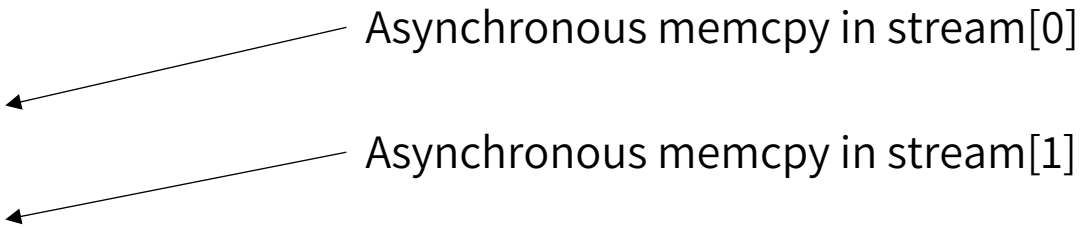
```
cudaStream_t my_stream;                                  //placeholder for stream id
cudaStreamCreate(&my_stream);                            //create a stream

//issue commands to my_stream

my_kernel<<<grid, threads, 0, my_stream>>>(...);  //kernel launch in a stream
cudaStreamDestroy(my_stream);                            //cleanup the stream
```

Streams example

```
cudaStream_t stream[2];
for (i=0; i<2; i++)
    cudaStreamCreate(&stream[i]);

cudaMemcpyAsync(d_A, h_A, A_size,
         cudaMemcpyHostToDevice, stream[0]);
cudaMemcpyAsync(d_B, h_B, B_size,
         cudaMemcpyHostToDevice, stream[1]);
kernel_A<<<grid, threads, 0, stream[0]>>>(d_A);
kernel_B<<<grid, threads, 0, stream[1]>>>(d_B);

cudaDeviceSynchronize();

for (i=0; i<2; i++)
    cudaStreamDestroy(stream[i]);
```

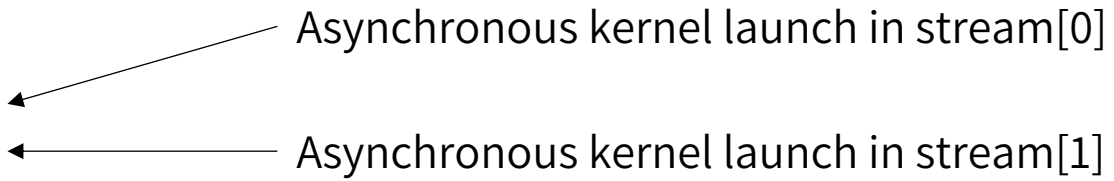
Streams example

```
cudaStream_t stream[2];
for (i=0; i<2; i++)
    cudaStreamCreate(&stream[i]);

cudaMemcpyAsync(d_A, h_A, A_size,
         cudaMemcpyHostToDevice, stream[0]);
cudaMemcpyAsync(d_B, h_B, B_size,
         cudaMemcpyHostToDevice, stream[1]);
kernel_A<<<grid, threads, 0, stream[0]>>>(d_A);
kernel_B<<<grid, threads, 0, stream[1]>>>(d_B);

cudaDeviceSynchronize();

for (i=0; i<2; i++)
    cudaStreamDestroy(stream[i]);
```



Streams example

```
cudaStream_t stream[2];
for (i=0; i<2; i++)
    cudaStreamCreate(&stream[i]);

cudaMemcpyAsync(d_A, h_A, A_size,
         cudaMemcpyHostToDevice, stream[0]);
cudaMemcpyAsync(d_B, h_B, B_size,
         cudaMemcpyHostToDevice, stream[1]);
kernel_A<<<grid, threads, 0, stream[0]>>>(d_A);
kernel_B<<<grid, threads, 0, stream[1]>>>(d_B);

cudaDeviceSynchronize();

for (i=0; i<2; i++)
    cudaStreamDestroy(stream[i]);
```



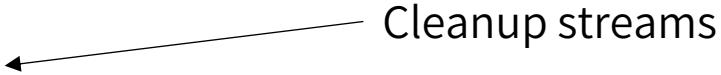
Streams example

```
cudaStream_t stream[2];
for (i=0; i<2; i++)
    cudaStreamCreate(&stream[i]);

cudaMemcpyAsync(d_A, h_A, A_size,
         cudaMemcpyHostToDevice, stream[0]);
cudaMemcpyAsync(d_B, h_B, B_size,
         cudaMemcpyHostToDevice, stream[1]);
kernel_A<<<grid, threads, 0, stream[0]>>>(d_A);
kernel_B<<<grid, threads, 0, stream[1]>>>(d_B);

cudaDeviceSynchronize();

for (i=0; i<2; i++)
    cudaStreamDestroy(stream[i]);
```

Block CPU execution until all GPU operations have completed

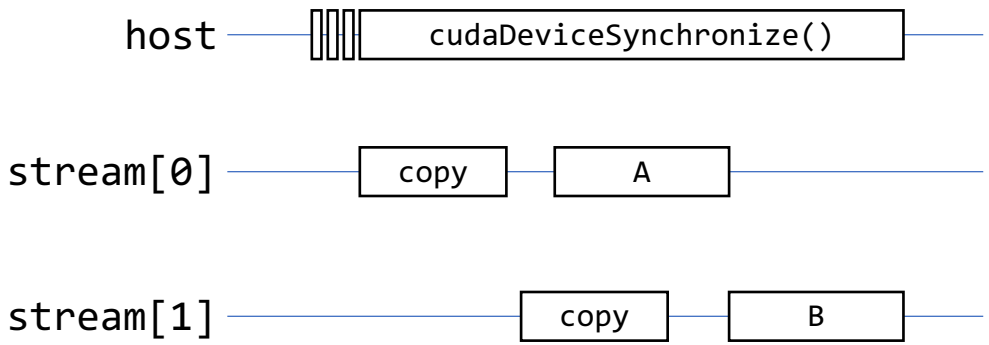
Streams example

```
cudaStream_t stream[2];
for (i=0; i<2; i++)
    cudaStreamCreate(&stream[i]);

cudaMemcpyAsync(d_A, h_A, A_size,
         cudaMemcpyHostToDevice, stream[0]);
cudaMemcpyAsync(d_B, h_B, B_size,
         cudaMemcpyHostToDevice, stream[1]);
kernel_A<<<grid, threads, 0, stream[0]>>>(d_A);
kernel_B<<<grid, threads, 0, stream[1]>>>(d_B);

cudaDeviceSynchronize();

for (i=0; i<2; i++)
    cudaStreamDestroy(stream[i]);
```



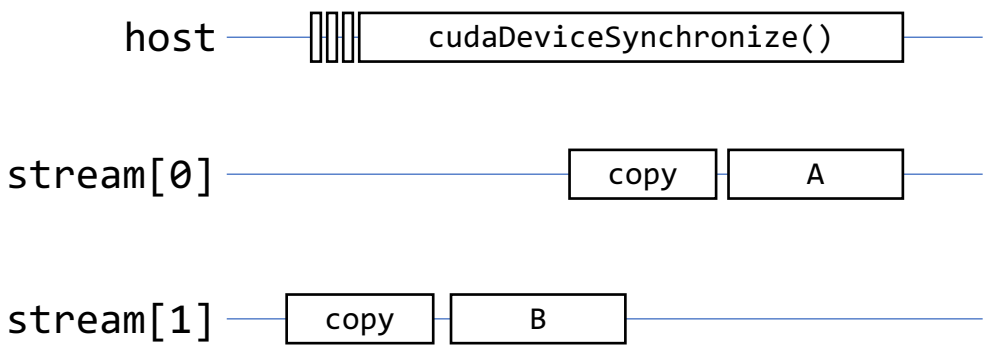
Streams example

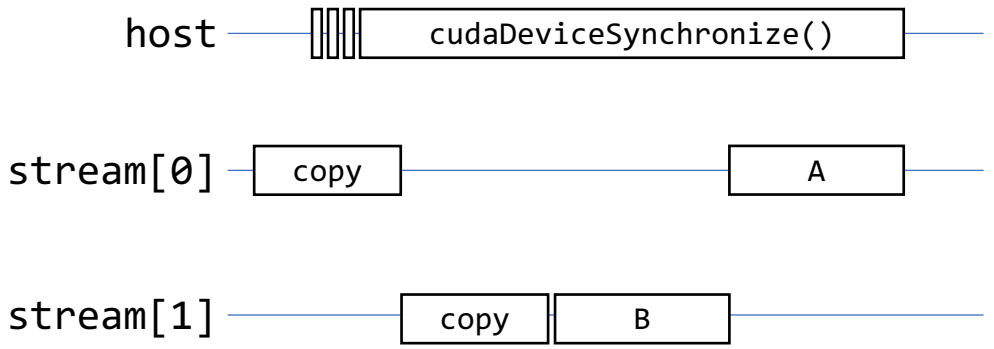
```
cudaStream_t stream[2];
for (i=0; i<2; i++)
    cudaStreamCreate(&stream[i]);

cudaMemcpyAsync(d_A, h_A, A_size,
         cudaMemcpyHostToDevice, stream[0]);
cudaMemcpyAsync(d_B, h_B, B_size,
         cudaMemcpyHostToDevice, stream[1]);
kernel_A<<<grid, threads, 0, stream[0]>>>(d_A);
kernel_B<<<grid, threads, 0, stream[1]>>>(d_B);

cudaDeviceSynchronize();

for (i=0; i<2; i++)
    cudaStreamDestroy(stream[i]);
```



Sequential consistency

- Within a stream commands are executed in order, but among streams operations may execute in any order
- Therefore, program correctness should next never depend on this order
- These executions of our previous example are also possible under the sequential consistency model:





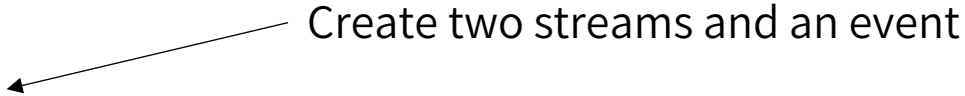
CUDA Events

- Events can be used to record points of execution in a stream
  - `cudaEventRecord(cudaEvent_t event, cudaStream_t stream = 0)`
- This is also used to measure execution time of CUDA kernels:
  - `cudaEventElapsedTime(float *ms, cudaEvent_t start, cudaEvent_t end)`
  - However, not safe when executing operations in different streams
- Block host execution to wait for an event
  - `cudaEventSynchronize(cudaEvent_t event)`
- Force operations in one stream to wait for an event, possibly in another stream, can be used to synchronize operations in different streams:
  - `cudaStreamWaitEvent(cudaStream_t stream, cudaEvent_t event, unsigned int flags)`

Cross-stream synchronization

```
cudaStream_t stream[2];
cudaEvent_t event;
for (i=0; i<2; i++)
    cudaStreamCreate(&stream[i]);
cudaEventCreate(&event);


kernel_A<<<grid, threads, 0, stream[0]>>>(d_out_A, d_A);
cudaEventRecord(event, stream[0]);
kernel_B<<<grid, threads, 0, stream[1]>>>(d_out_B, d_B);
cudaStreamWaitEvent(stream[1], event, 0);
kernel_C<<<grid, threads, 0, stream[1]>>>(d_out, d_out_A, d_out_B);
```

```
cudaStream_t stream[2];
cudaEvent_t event;
for (i=0; i<2; i++)
    cudaStreamCreate(&stream[i]);
cudaEventCreate(&event);


kernel_A<<<grid, threads, 0, stream[0]>>>(d_out_A, d_A);
cudaEventRecord(event, stream[0]);
kernel_B<<<grid, threads, 0, stream[1]>>>(d_out_B, d_B);
cudaStreamWaitEvent(stream[1], event, 0);
kernel_C<<<grid, threads, 0, stream[1]>>>(d_out, d_out_A, d_out_B);
```



Cross-stream synchronization

```
cudaStream_t stream[2];
cudaEvent_t event;
for (i=0; i<2; i++)
    cudaStreamCreate(&stream[i]);
cudaEventCreate(&event);


kernel_A<<<grid, threads, 0, stream[0]>>>(d_out_A, d_A);
cudaEventRecord(event, stream[0]);
kernel_B<<<grid, threads, 0, stream[1]>>>(d_out_B, d_B);
cudaStreamWaitEvent(stream[1], event, 0);
kernel_C<<<grid, threads, 0, stream[1]>>>(d_out, d_out_A, d_out_B);
```

Record event in stream[0]

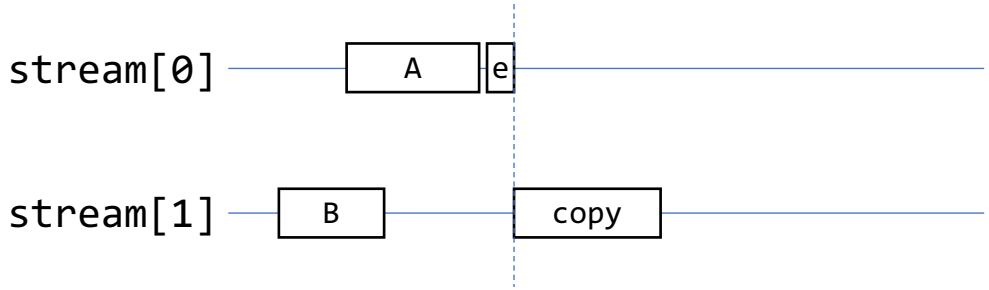
Cross-stream synchronization

```
cudaStream_t stream[2];
cudaEvent_t event;
for (i=0; i<2; i++)
    cudaStreamCreate(&stream[i]);
cudaEventCreate(&event);


kernel_A<<<grid, threads, 0, stream[0]>>>(d_out_A, d_A);
cudaEventRecord(event, stream[0]);
kernel_B<<<grid, threads, 0, stream[1]>>>(d_out_B, d_B);
cudaStreamWaitEvent(stream[1], event, 0);
kernel_C<<<grid, threads, 0, stream[1]>>>(d_out, d_out_A, d_out_B);
```

Block stream[1] until event

Cross-stream synchronization

```
cudaStream_t stream[2];
cudaEvent_t event;
for (i=0; i<2; i++)
    cudaStreamCreate(&stream[i]);
cudaEventCreate(&event);
```



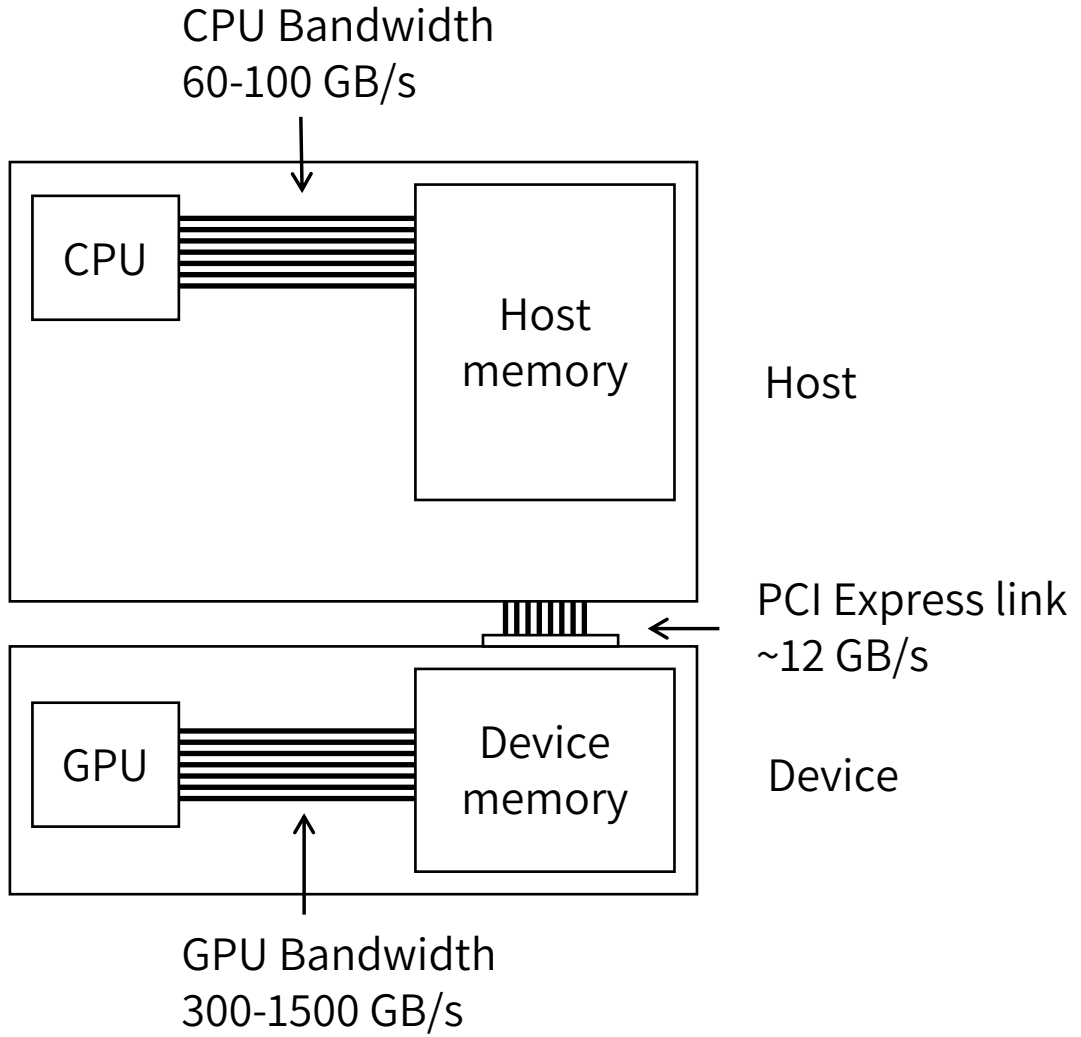

```
kernel_A<<<grid, threads, 0, stream[0]>>>(d_out_A, d_A);
cudaEventRecord(event, stream[0]);
kernel_B<<<grid, threads, 0, stream[1]>>>(d_out_B, d_B);
cudaStreamWaitEvent(stream[1], event, 0);
kernel_C<<<grid, threads, 0, stream[1]>>>(d_out, d_out_A, d_out_B);
```

Asynchronous memory copies

- `cudaError_t cudaMemcpyAsync(void* dst, const void* src, size_t count, cudaMemcpyKind kind, cudaStream_t stream = 0)`
- It is named this way because it *can be* asynchronous with respect to the host
- This means the function call may return before it has been completed:
  - Explicit synchronization is needed before the data can be used on the host
- The operation will be asynchronous, if and only if:
  - A *non-default stream* is used
  - Host memory is allocated as page-locked (pinned) memory using `cudaMallocHost`
- Only if a non-default stream is used may the operation overlap with other operations in other non-default streams

Asynchronous processing: Summary

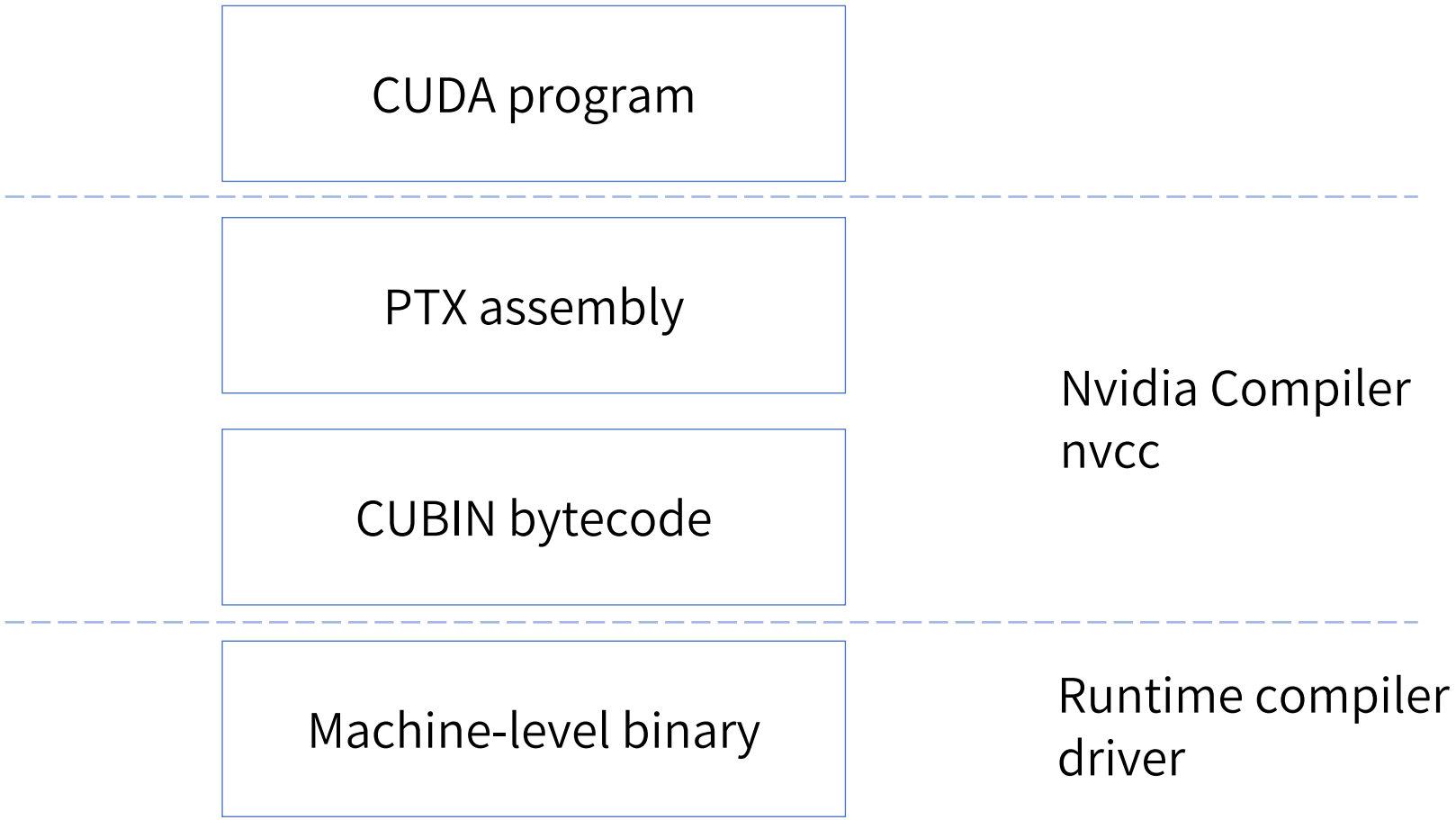
- The *host* and the *device* are different processors and *may* run independently
  - "Async" functions *can be* asynchronous with respect to host
  - Streams are sequentially consistent
  - Operations in different streams may execute out of order
  - Events can be used to synchronize the host or operations in different streams



netherlands eScience center

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 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)
```

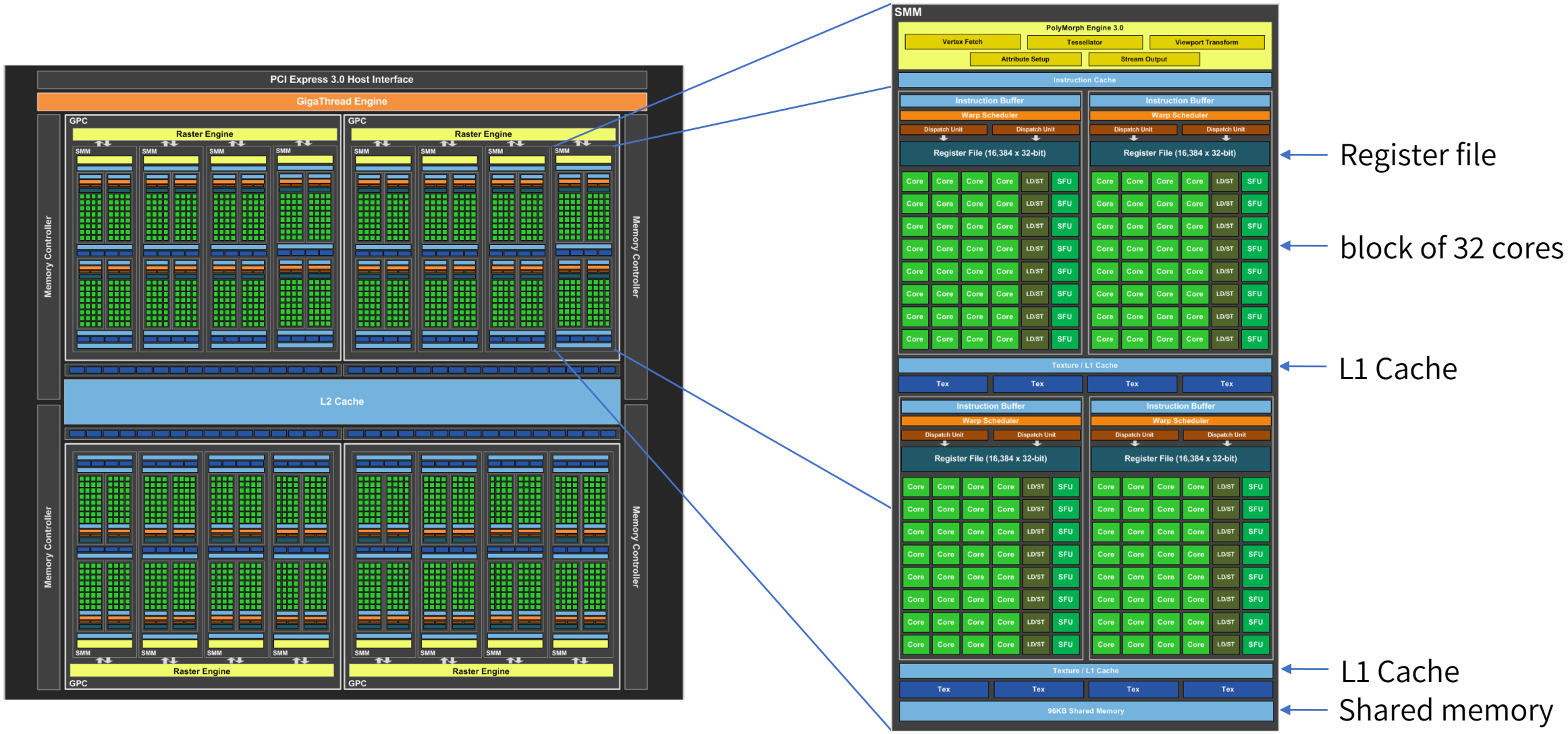
  - This is less efficient, but not always bad.
  - Avoid data-dependent conditional branching if possible

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

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

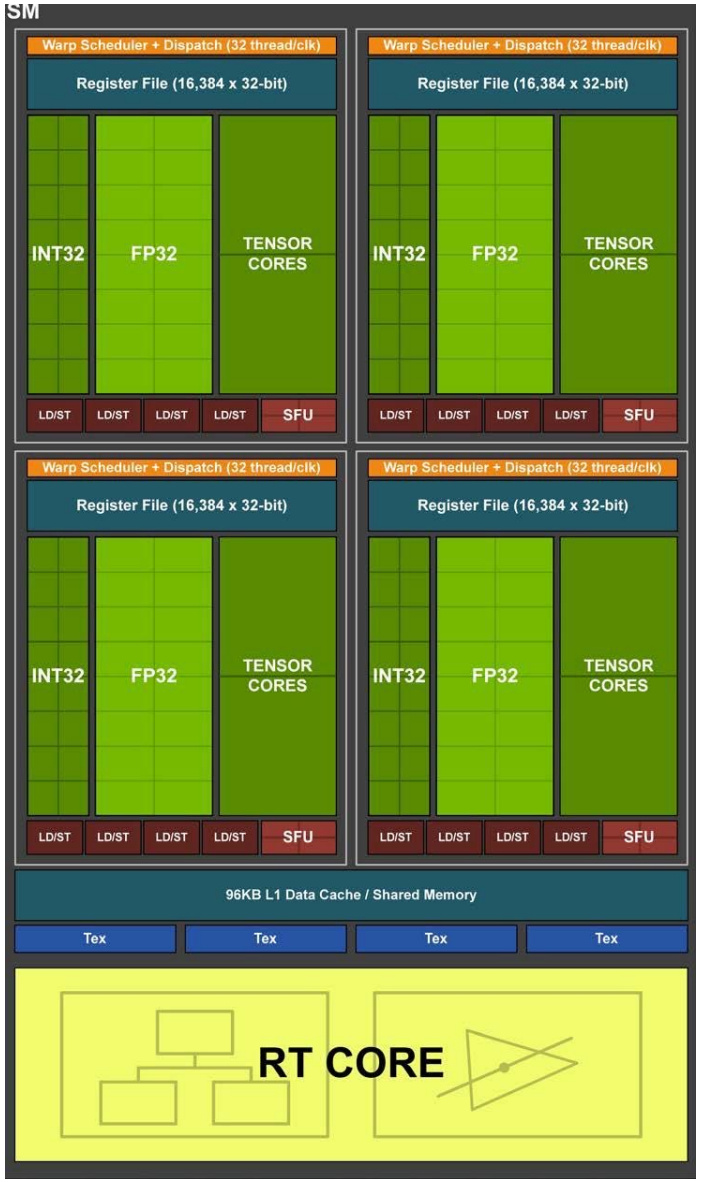
- The Volta architecture replaces predication with a per-thread program counter and call stack. The same performance recommendations apply, however.

Maxwell Architecture



Turing architecture

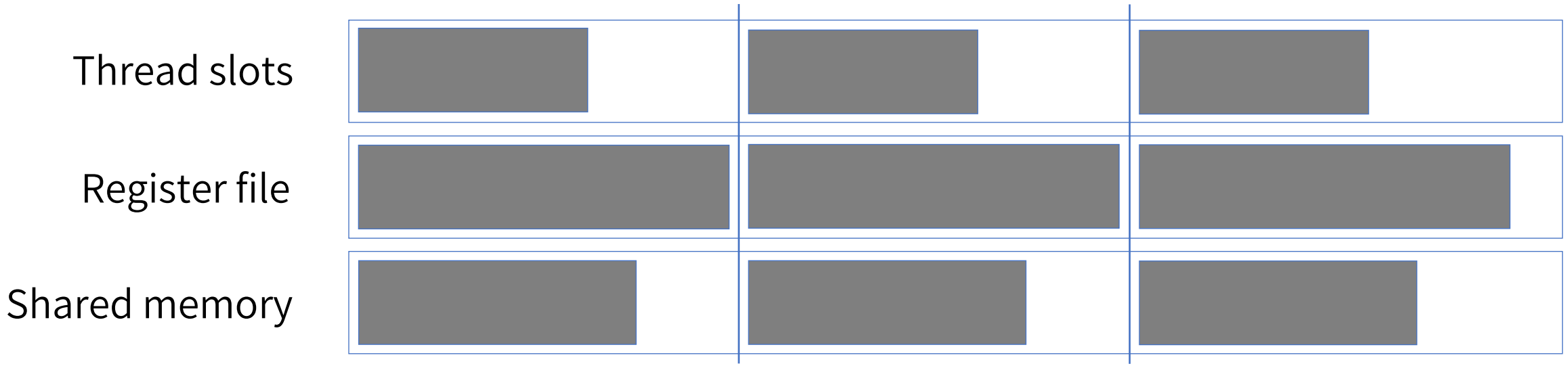
- Features specialized Tensor and RT cores
- Tensor cores can operate on 4/8/16 bit integers and 16 bit half-precision floating points
- RT cores used for Ray-Tracing in graphics workloads



Resource partitioning

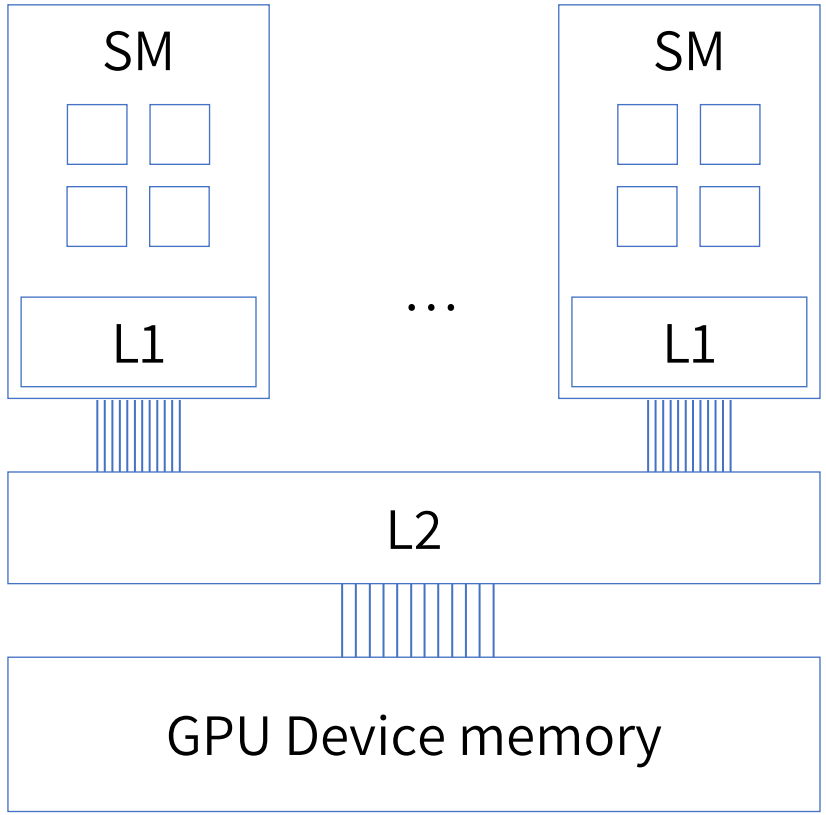
- The GPU consists of several (1 to 68) *streaming multiprocessors* (SMs)
- The SMs are fully independent
- Each SM contains several resources: Register file, Shared memory, Thread Slots, and Thread Block slots

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

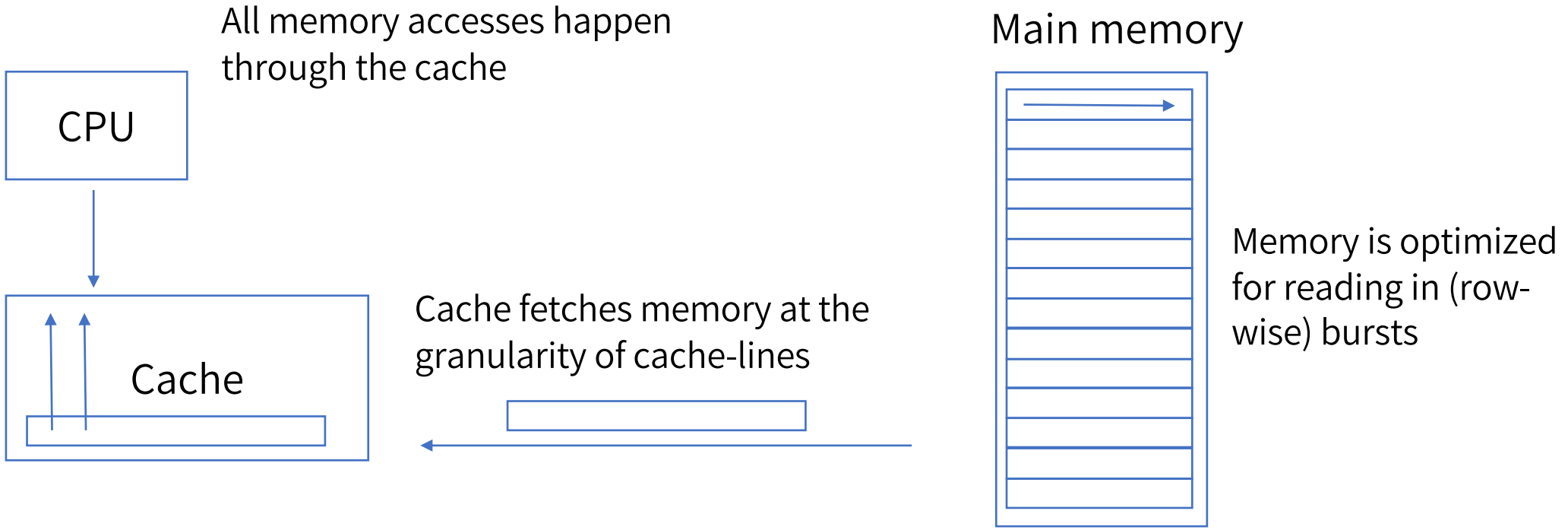


- Global memory is cached at L2, and for some GPUs also in L1

- 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



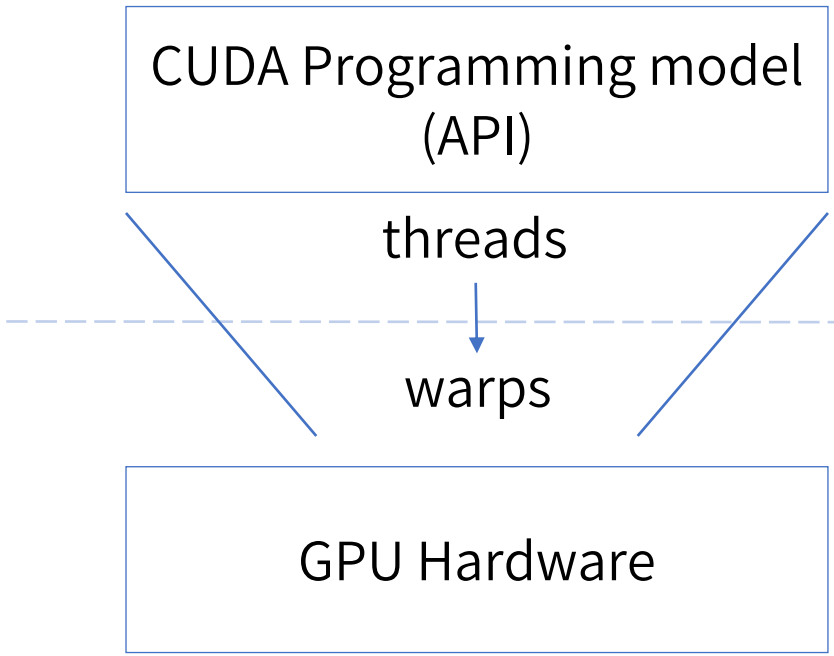
Subsequently accessing values that are adjacent on the same cache line is much faster than when each access requires a new cache line to be fetched

- 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 Kernel Tuner (https://github.com/KernelTuner/kernel_tuner)



Think in terms of threads
Reason on program correctness

Think in terms of warps
Reason on program performance


Coffee break

netherlands eScience center

GPU Optimization techniques

- Modify the kernel code to improve performance or tunability
- Effects on performance can be different on different GPUs or different input data
- You can tune
  - enabling or disabling an optimization
  - the parameters introduced by certain optimizations
- You often need to combine multiple different optimizations with specific tunable parameter values to arrive at optimal performance

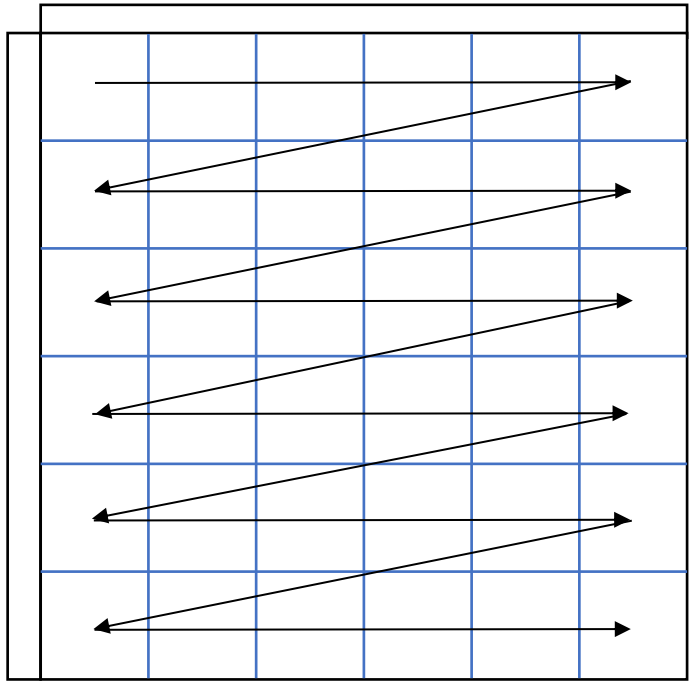
- Coalescing memory accesses
- Host/device communication
- Kernel fusion
- Loop blocking
- Loop unrolling
- Prefetching
- Recomputing values
- Reducing atomics
- Reducing branch divergence
- Reducing redundant work
- Reducing register usage
- Reformatting input data
- Using a specific memory space
- Using warp shuffle instructions
- Varying work per thread
- Vectorization

- Coalescing memory accesses
- Host/device communication
- Kernel fusion
- **Loop blocking**
- **Loop unrolling**
- Prefetching
- Recomputing values
- Reducing atomics
- **Reducing branch divergence**
- Reducing redundant work
- **Reducing register usage**
- Reformatting input data
- Using a specific memory space
- Using warp shuffle instructions
- **Varying work per thread**
- **Vectorization**

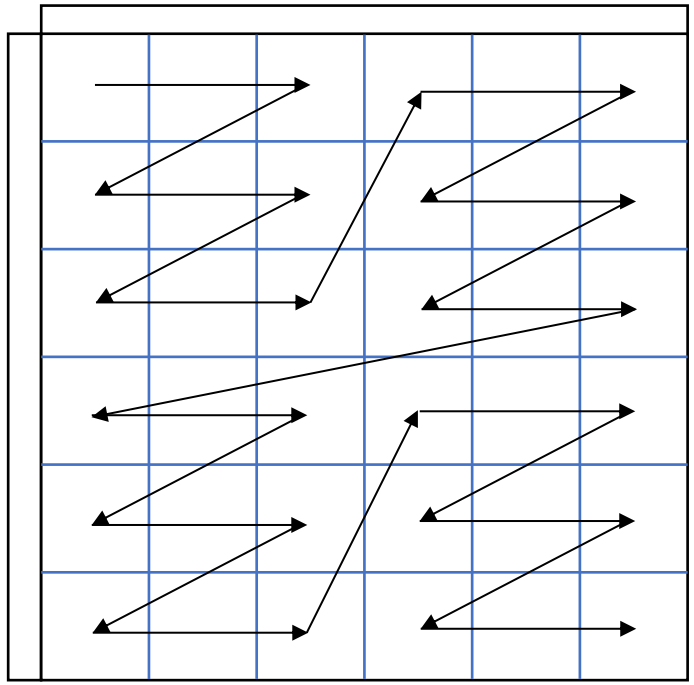
Loop blocking

- Why?
  - Increases spatial / temporal locality
  - Reduces the ‘working set’ of the algorithm
- How?
  - Change the order of computations and data accesses in nested loops
  - Usually nearly doubles the number of for-loops in the code
  - Outer-loops iterate over the blocks
  - Inner-loops iterate within each block

Loop blocking



```
for (int j=0; j<ny; j++) {
    for (int i=0; i<nx; i++) {
        ...[j*nx + i]
    }
}
```



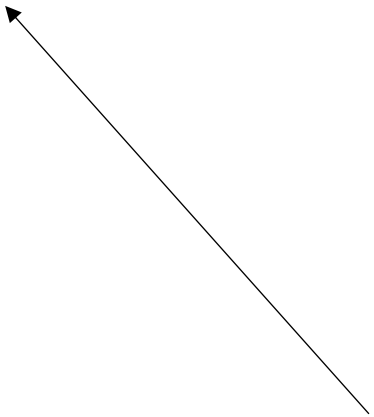
```
for (int j=0; j<ny; j+=nyb) {
    for (int i=0; i<nx; i+=nxb) {

        for (int jb=0; jb<nyb; jb++) {
            for (int ib=0; ib<nxb; ib++) {
                ...[(j+jb)*nx + (i+ib)]
            }
        }
    }
}
```

(Partial) Loop Unrolling

- Why?
  - Increases instruction-level-parallelism
  - Reduces loop overhead instructions
- How?
  - Replicate the contents of a for-loop n times, increase loop counter by n
  - In the early days, only manually or with a code generator
  - Compiler does this now: `#pragma unroll <value>`
  - In CUDA, value has to be an integer constant expression
    - 0 is not allowed and gives an error, 1 means unrolling is disabled

Partial loop unrolling

```
...

#pragma unroll loop_unroll_factor_nlay
for (int ilay=0; ilay<nlay; ++ilay) {
    ...
}
```



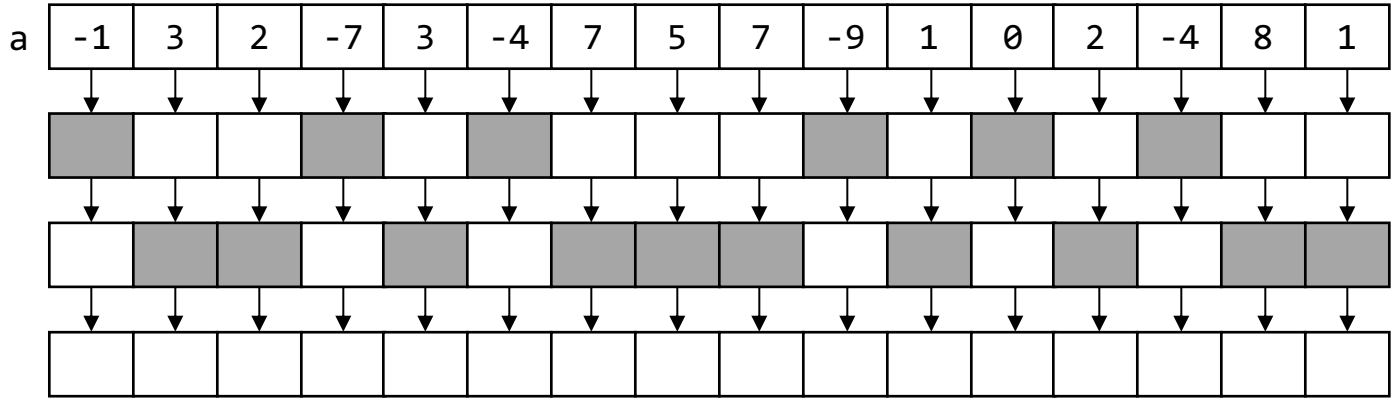
The compiler can unroll this loop if `nlay` is known at compile-time. The `loop_unroll_factor_nlay` parameter should be a divisor of the loop counter, i.e. `nlay`.

Reducing Branch Divergence

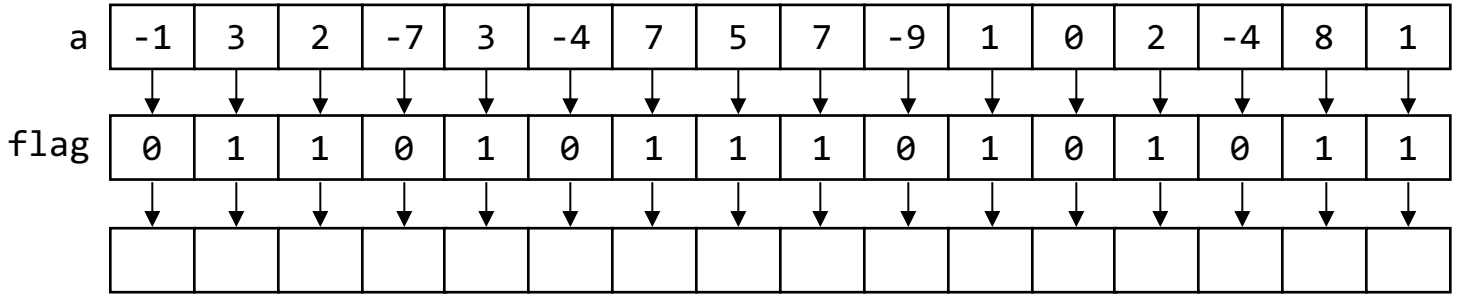
- Why?
  - Threads within the same warp taking different paths will be predicated
  - Improve utilization by reducing the number of predicated instructions
  - May also improve access patterns and memory throughput
- How?
  - By avoiding or removing control-flow structures, if-then-else
  - If you need to use if-then-else, prefer to branch based on the `threadIdx` such that threads within the same warp have the same control flow
  - By reordering or sorting similar tasks based on their control flow needs

Reducing Branch Divergence

```
if a>0
  c = a*a+b
else
  c = a*a-b
```




```
flag = a>0
c = a*a+(2*flag-1)*b
```

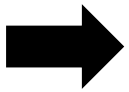


Reducing register usage

- Why?
  - Registers are an important and limited SM resource and are likely to limit occupancy
  - Allows to increase the tunable range of thread block dimensions
- How?
  - Compiling constant values into your code rather than keeping them in registers (e.g. using templates or tunable parameters)
  - Limiting or disabling loop unrolling are very effective ways of reducing register usage
  - In kernels that do many different things, splitting the kernel may help to cut down register usage
  - Enabling register spilling with compiler flag `-maxrregcount=N` or tuning the number of blocks per SM using the kernel `__launch_bounds__()`

Reducing register usage

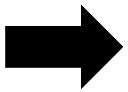
```
template<typename TF> __global__
void some_kernel(const int ncol,
                 const int nlay,
                 const int ngpt,
                 const int top_at_1, TF* flux_dn)
{
    const int icol = blockIdx.x*blockDim.x + threadIdx.x;
    const int igpt = blockIdx.y*blockDim.y + threadIdx.y;
    if ( (icol < ncol) && (igpt < ngpt) )
    {
        if (top_at_1)
        {
            ...
        }
        else
        {
            ...
        }
    }
}
```



```
template<typename TF, int top_at_1> __global__
void some_kernel(const int ncol,
                 const int nlay,
                 const int ngpt,
                 TF* flux_dn)
{
    const int icol = blockIdx.x*blockDim.x + threadIdx.x;
    const int igpt = blockIdx.y*blockDim.y + threadIdx.y;
    if ( (icol < ncol) && (igpt < ngpt) )
    {
        if (top_at_1)
        {
            ...
        }
        else
        {
            ...
        }
    }
}
```

Reducing register usage

```
__global__ void


some_kernel(...)
{
    ...
}
```



```
__global__ void
__launch_bounds__(block_size_x*block_size_y*block_size_z,
                  blocks_per_sm)
some_kernel(...)
{
    ...
}
```

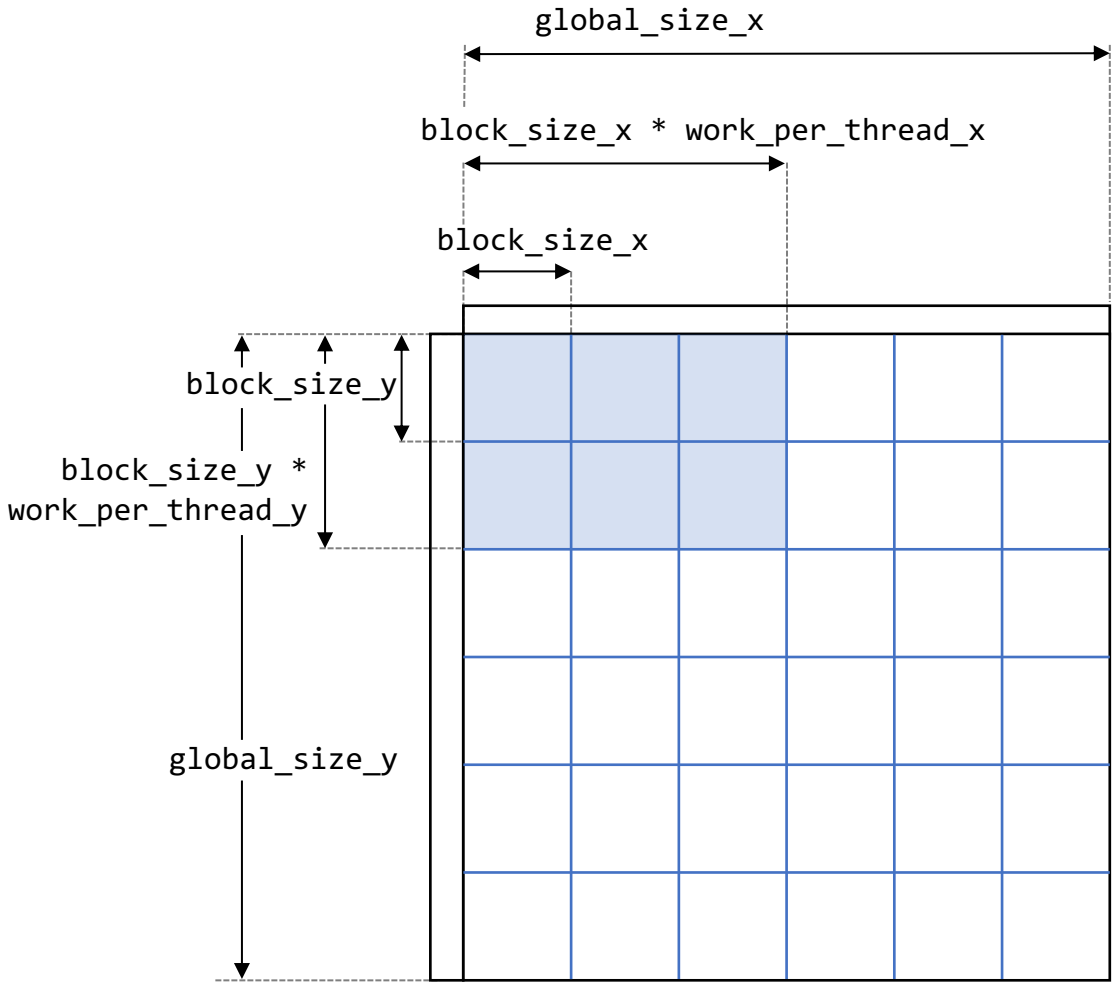
Varying work per thread

- Why?
  - Increasing work per thread often increases data reuse and locality
  - Reduces redundant instructions previously executed by other threads
  - Increases instruction-level parallelism and possibly increases register usage
- How?
  - Reduce number of threads blocks in total, but increase the work per thread block
  - Bring down number of threads within the block, but keep the amount of work equal

Varying work per thread

```
...
#pragma unroll
for (kb = 0; kb < block_size_x; kb++) {

    sum[i][j] += sA[ty][kb] * sB[kb][tx];

}
```

```
...
#pragma unroll
for (kb = 0; kb < block_size_x; kb++) {

    #pragma unroll
    for (int j = 0; j < work_per_thread_x; j++) {
        sum[i][j] += sA[ty][kb] * sB[kb][tx + j * block_size_x];
    }

}
```



Vectorization

- Why?
  - Reduces the instructions needed to fetch data from global memory
  - Improves memory throughput
  - Often also increases work per thread and instruction-level parallelism
  - May increase register usage
- How?
  - Using wider data types (e.g. `float2` or `float4` instead of `float`)
  - Vector length can be tuned

Vectorization

```
#if (vector==1)
#define floatvector float
#elif (vector == 2)
#define floatvector float2
#elif (vector == 4)
#define floatvector float4
#endif

__global__ void sum_floats(float *sum_global, floatvector *array, int n) {

    int x = blockIdx.x * block_size_x + threadIdx.x;
    if (x < n/vector) {
        floatvector v = array[x];

        ...
    }
...
```

- Coalescing memory accesses
- Host/device communication
- Kernel fusion
- **Loop blocking**
- **Loop unrolling**
- Prefetching
- Recomputing values
- Reducing atomics
- **Reducing branch divergence**
- Reducing redundant work
- **Reducing register usage**
- Reformatting input data
- Using a specific memory space
- Using warp shuffle instructions
- **Varying work per thread**
- **Vectorization**

- *Optimization Techniques for GPU Programming*
  Pieter Hijma, Stijn Heldens, Alessio Sclocco, Ben van Werkhoven, and Henri Bal
  ACM Computing surveys 2022
  https://dl.acm.org/doi/abs/10.1145/3570638

netherlands eScience center

Auto-tuning

To maximize GPU code performance, you need to find the best combination of:

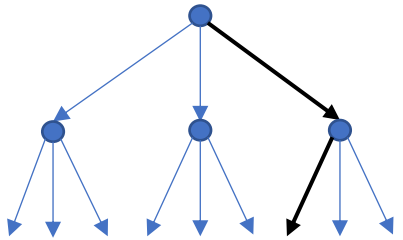
- Different mappings of the problem to threads and thread blocks
- Different data layouts in different memories (shared, constant, …)
- Different ways of exploiting special hardware features
- Thread block dimensions
- Code optimizations that may be applied or not
- Work per thread in each dimension
- Loop unrolling factors
- Overlapping computation and communication
- …

Problem:

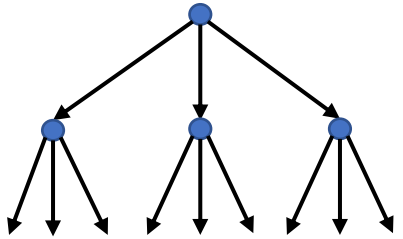
- Creates a very large design space

Manual optimization versus auto-tuning

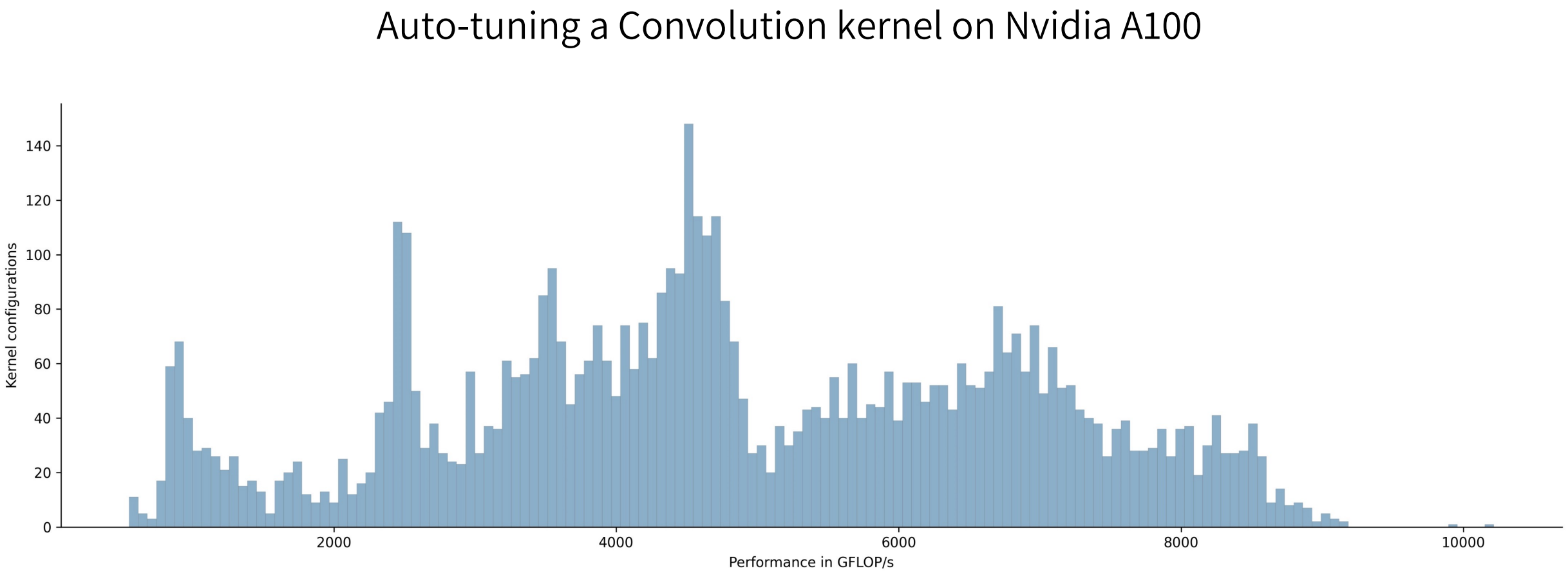
- Optimizing code manually you iteratively:
  - Modify the code
  - Run a few benchmarks
  - Revert or accept the change



- With auto-tuning you:
  - Write a templated version of your code or a code generator
  - Benchmark the performance of all versions of the code



Large search space of kernel configurations



A Python tool for testing and tuning GPU kernels

Easy to use:
- Can be used directly on existing kernels and code generators
- Inserts no dependencies in the kernels or host application
- 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 codeUnit testing GPU code
- …

https://github.com/KernelTuner/kernel_tuner

Minimal example

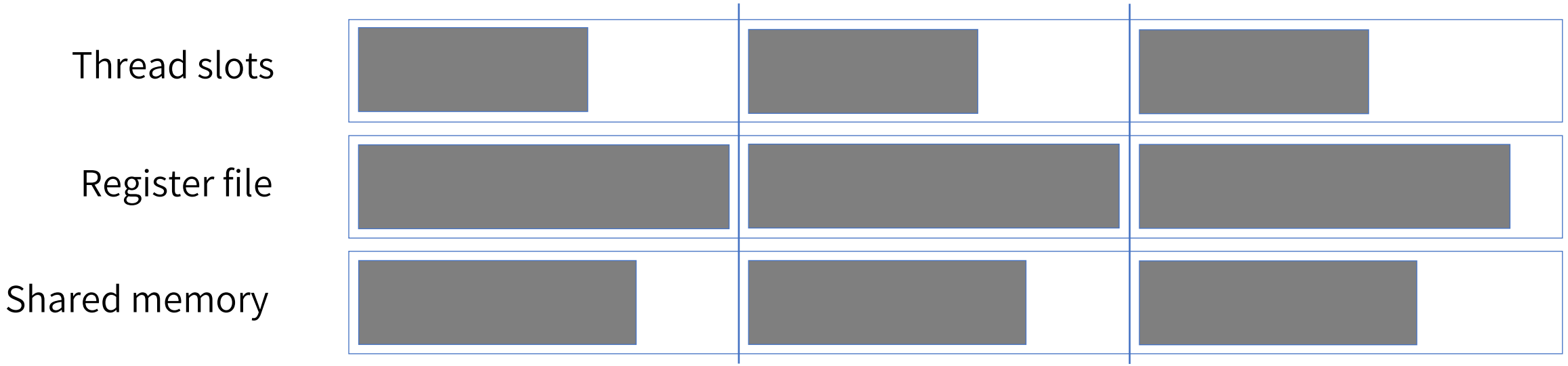
```
import numpy
from kernel_tuner import tune_kernel

kernel_string = """
__global__ void vector_add(float *c, float *a, float *b, int n) {
    int i = blockIdx.x * block_size_x + threadIdx.x;
    if (i<n) {
        c[i] = a[i] + b[i];
    }
}"""

n = numpy.int32(1e7)
a = numpy.random.randn(n).astype(numpy.float32)
b = numpy.random.randn(n).astype(numpy.float32)
c = numpy.zeros_like(b)
args = [c, a, b, n]
tune_params = {"block_size_x": [32, 64, 128, 256, 512]}

tune_kernel("vector_add", kernel_string, n, args, tune_params)
```

Choosing thread block dimensions

- Almost all GPU kernels can be written in a form that allows for varying thread block dimensions
- Usually, changing thread block dimensions affects performance, but not the result
- The question is, how to determine the optimal setting?

Why do thread block dimensions matter so much?

- The GPU consists of several (1 to 80) *streaming multiprocessors* (SMs)
- The SMs are fully independent and contain 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*



Specifying thread block dimensions

- In Kernel Tuner, you specify the possible values for thread block dimensions of your kernel using special tunable parameters:
  - `block_size_x, block_size_y, block_size_z`
- For each, you may pass a list of values this parameter can take:
  - `params["block_size_x"] = [32, 64, 128, 256]`

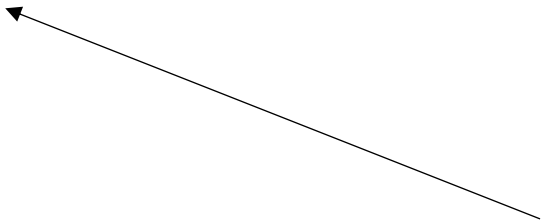
- You can use different names for these by passing the `block_size_names` option using a list of strings
- Note: when you change the thread block dimensions, the number of thread blocks used to launch the kernel generally changes as well

Access thread block dimensions at compile-time

- Kernel Tuner automatically inserts a block of `#define` statements to set values for `block_size_x`, `block_size_y`, and `block_size_z`
- You can use these values in your code to access the thread block dimensions as compile-time constants
- This is generally a good idea for performance, because
  - Loop conditions may use the thread block dimensions, fixing the number of iterations at compile-time allows the compiler to unroll the loop and optimize the code
  - Shared memory declarations can use the thread block dimensions, e.g. for compile-time sizes multi-dimensional data

Vector add example

```
import numpy
from kernel_tuner import tune_kernel

kernel_string = """
__global__ void vector_add(float *c, float *a, float *b, int n) {
    int i = blockIdx.x * block_size_x + threadIdx.x;
    if (i<n) {
        c[i] = a[i] + b[i];
    }
}"""

n = numpy.int32(1e7)
a = numpy.random.randn(n).astype(numpy.float32)
b = numpy.random.randn(n).astype(numpy.float32)
c = numpy.zeros_like(b)
args = [c, a, b, n]
tune_params = {"block_size_x": [32, 64, 128, 256, 512]}

tune_kernel("vector_add", kernel_string, n, args, tune_params)
```



Notice how we can use `block_size_x` in our `vector_add` kernel code, while it is actually not defined (yet)

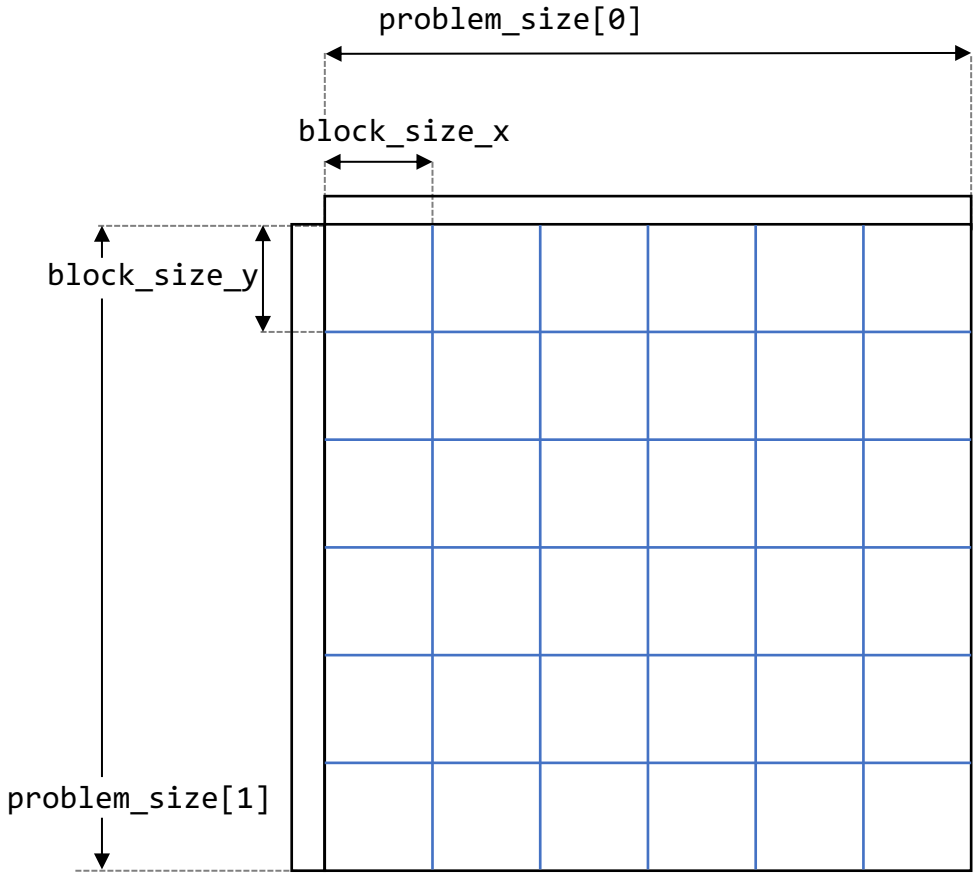
Vector add example

```
import numpy
from kernel_tuner import tune_kernel

kernel_string = """
__global__ void vector_add(float *c, float *a, float *b, int n) {
    int i = blockIdx.x * block_size_x + threadIdx.x;
    if (i<n) {
        c[i] = a[i] + b[i];
    }
}"""

n = numpy.int32(1e7)
a = numpy.random.randn(n).astype(numpy.float32)
b = numpy.random.randn(n).astype(numpy.float32)
c = numpy.zeros_like(b)
args = [c, a, b, n]
tune_params = {"block_size_x": [32, 64, 128, 256, 512]}

tune_kernel("vector_add", kernel_string, n, args, tune_params)
```

n is the number of elements in our array, the number of thread blocks depends on both n and `block_size_x`

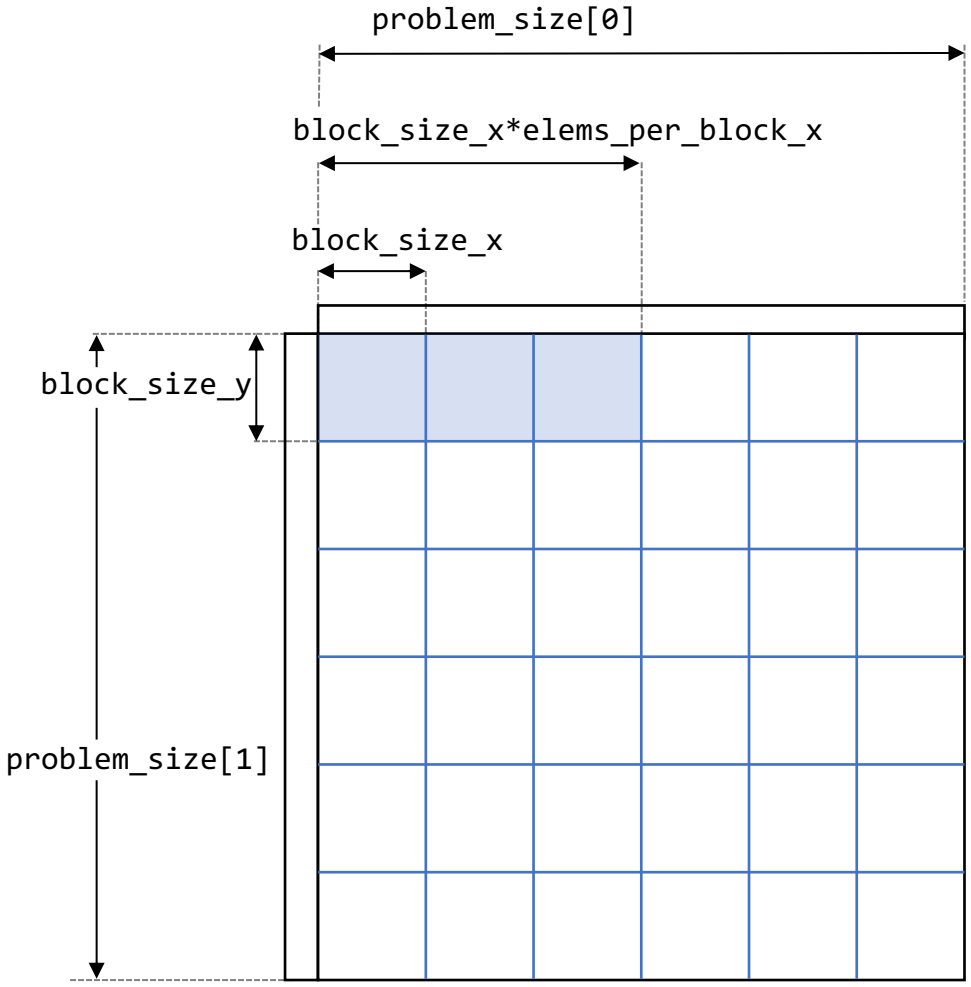
Specifying grid dimensions

- You specify the `problem_size`
- `problem_size` describes the dimensions across which threads are created
- By default, the grid dimensions are computed as:
  - `grid_size_x = ceil(problem_size_x / block_size_x)`



Grid divisor lists

- Other parameters, or none at all, may also affect the grid dimensions
- Grid divisor lists control how `problem_size` is divided to compute the grid size
- Use the optional arguments:
  - `grid_div_x`, `grid_div_y`, and `grid_div_z`
- You may disable this feature by explicitly passing empty lists as grid divisors, in which case `problem_size` directly sets the grid dimensions



- The problem size is usually a single integer or a tuple of integers
- Use strings to derive `problem_size` from a tunable parameter
- May also be a (lambda) function that takes a dictionary of tunable parameters and returns a tuple of integers
- For example, `reduction.py`:

```
size = 800_000_000
tune_params["block_size_x"] = [32, 64, 128, 256, 512, 1024]
tune_params["num_blocks"] = [32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768]
problem_size = "num_blocks"
grid_div_x = []
```

- By default, the search space is the Cartesian product of all possible combinations of tunable parameter values
- Example:

```
tune_params["block_size_x"] = [32, 64, 128, 256, 512]
tune_params["tile_size_x"] = [1, 2, 3, 4, 5, 6, 7, 8]
tune_params["loop_unroll_factor_x"] = [1, 2, 3, 4, 5, 6, 7, 8]
```

- However, for some tunable kernels:
  - there are tunable parameters that depend on each other
  - only certain combinations of tunable parameter values are valid

Dependent parameters example

```
tune_params["block_size_x"] = [32, 64, 128, 256, 512]
tune_params["tile_size_x"] = [1, 2, 3, 4, 5, 6, 7, 8]
tune_params["loop_unroll_factor_x"] = [1, 2, 3, 4, 5, 6, 7, 8]
```

- In this example:
  - One parameter controls a loop count: `tile_size_x`
  - Another parameter controls the partial loop unrolling factor of that loop: `loop_unroll_factor_x`
- By default, Kernel Tuner considers search space to be the Cartesian product of all possible combinations of all values for all parameters
- But only configurations in which `loop_unroll_factor_x` is a divisor of `tile_size_x` are valid

Partial loop unrolling example

```
tune_params["block_size_x"] = [32, 64, 128, 256, 512]
tune_params["tile_size_x"] = [1, 2, 3, 4, 5, 6, 7, 8]
tune_params["loop_unroll_factor_x"] = [1, 2, 3, 4, 5, 6, 7, 8]

# define a lambda function that returns True when a configuration is valid
restrict = lambda p: p["loop_unroll_factor_x"] <= p["tile_size_x"] and
                     p["tile_size_x"] % p["loop_unroll_factor_x"] == 0

# pass our lambda function to the restrictions option
tune_kernel(..., restrictions=restrict, ...)
```

Auto-tuning as numerical optimization

- Combine all sets of values S_i for all n tunable parameters to form search space

$$\mathcal{X} = S_1 \times S_2 \times \cdots \times S_n$$

- Let $f(x)$ be the execution time of kernel configuration $x \in \mathcal{X}$

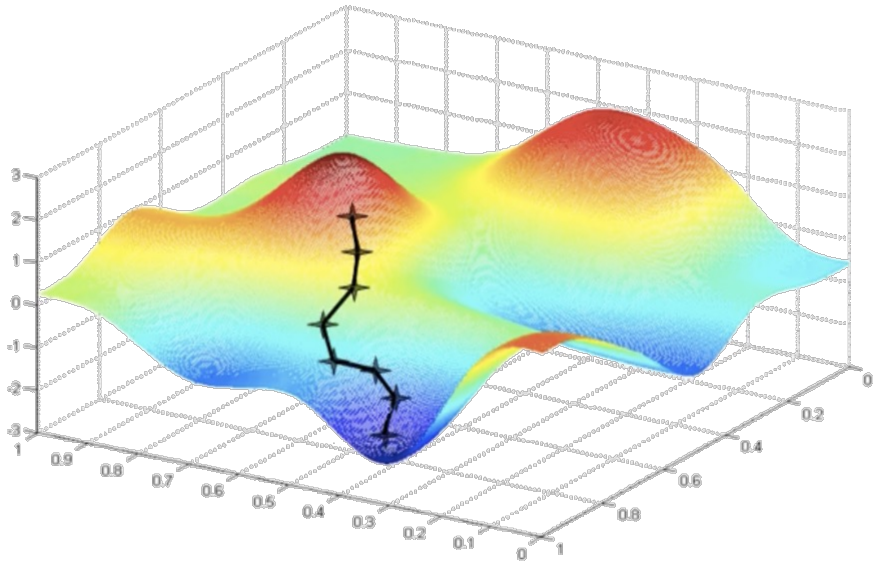
- Treat the problem as a numerical optimization problem

$$x_{opt} = \arg\min f(x)$$

Optimization strategies in Kernel Tuner

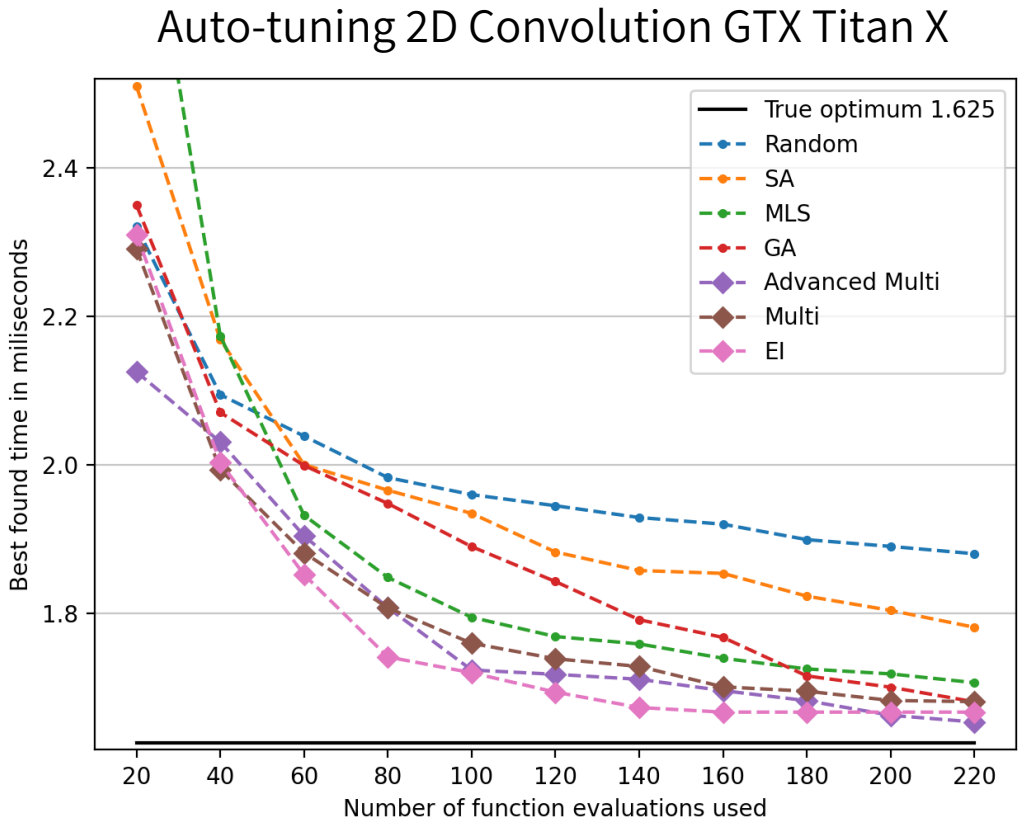
- Local optimization
  - Nelder-Mead, Powell, CG, BFGS, L-BFGS-B, TNC, COBYLA, and SLSQP

- Global optimization
  - Basin Hopping
  - Simulated Annealing
  - Differential Evolution
  - Genetic Algorithm
  - Particle Swarm Optimization
  - Firefly Algorithm
  - Multi-start local search
  - Bayesian Optimization
  - …



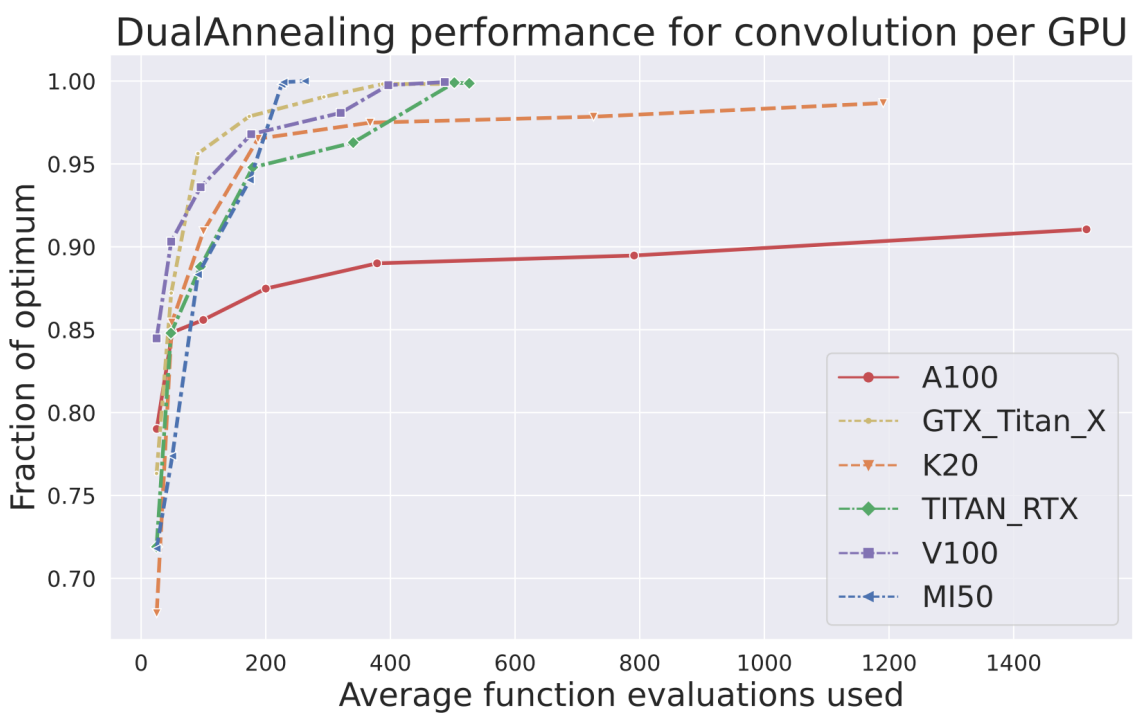
- Challenges to applying Bayesian Optimization:
  - discrete search spaces
  - invalid configurations
  - search space restrictions

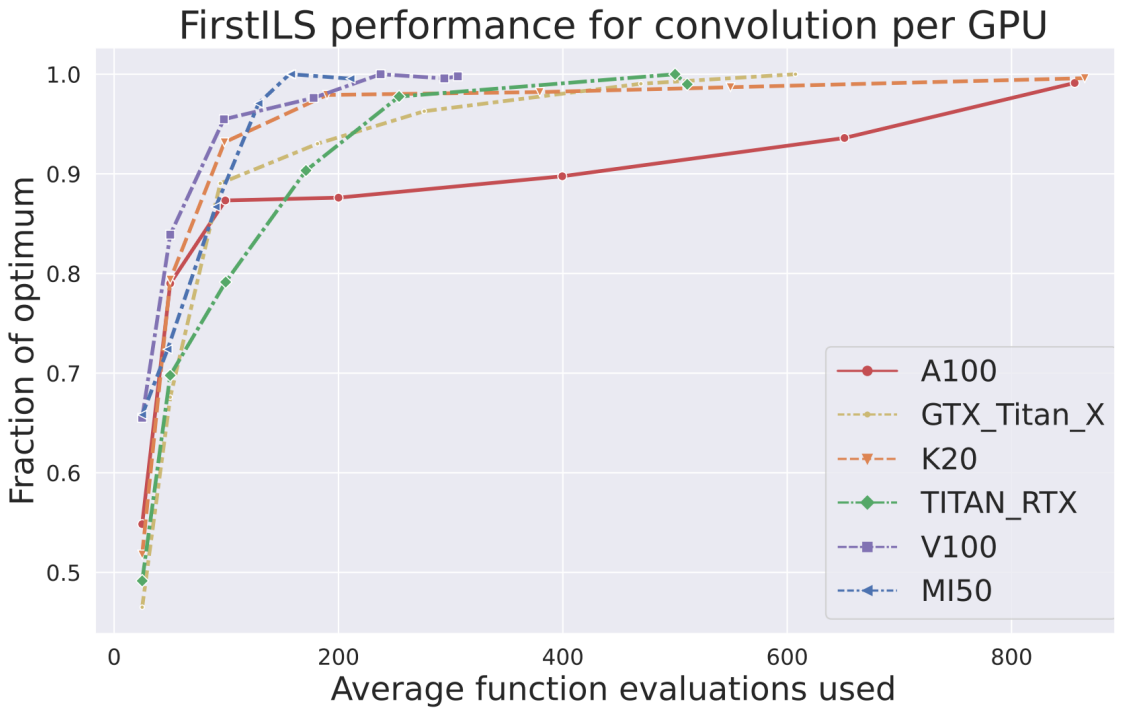
- Outperforms most other methods for most auto-tuning problems, including existing BO packages (e.g. bayes-opt, scikitopt)



Bayesian Optimization for auto-tuning GPU kernels
F.J. Willemsen, R.V. van Nieuwpoort, B. van Werkhoven
PMBS workshop at SC21 2021

Benchmarking optimization algorithm performance



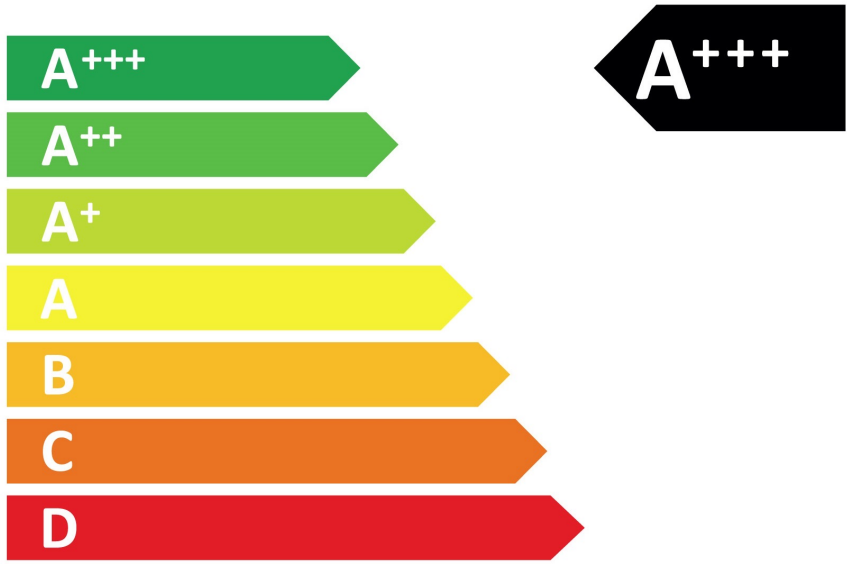


Benchmarking optimization algorithms for auto-tuning GPU kernels
Richard Schoonhoven, Ben van Werkhoven, K. Joost Batenburg
IEEE Transactions on Evolutionary Computation (TEVC) 2022

Minimize energy consumption instead of only the execution time

Support added to Kernel Tuner in 2022:

- Measuring energy consumption during tuning
- Custom objective to optimize for time, energy or any user-defined metric



Going Green: optimizing GPUs for energy efficiency through model-steered auto-tuning
Richard Schoonhoven, Bram Veenboer, Ben van Werkhoven, K. Joost Batenburg
PMBS workshop at SC22 2022

netherlands eScience center

Fourth hands-on

Fourth Hands-on Session: Local Averages

- Select the notebook for the fourth exercise
  - https://github.com/benvanwerkhoven/gpu-course/blob/master/notebooks/local_averages-KT.ipynb

- The kernel is correct, your task is to optimize the code and improve its performance
- You can keep working on this exercise after the course ends

- Hints:
  - Is the memory access pattern optimal?
  - Are threads mapped the right way for the problem?
  - Could vector operations be useful?
  - Is there any data reuse available?

netherlands eScience center

Closing remarks

Going further

- Read these slides again
  - And follow the links to the documentation!
- Play with the notebooks
  - If you do not have access to the DAS, try Google Colab, they have GPU support
- Check the C code for the exercise
- If you have questions, contact us
  - b.vanwerkhoven@esciencecenter.nl
  - a.sclocco@esciencecenter.nl

Kernel Tuner – developed open source

- We are developing Kernel Tuner as an open-source project
- GitHub repository:
  - https://github.com/KernelTuner/kernel_tuner
- License: Apache 2.0
- If you use Kernel Tuner in a project, please cite the paper:
  - B. van Werkhoven, Kernel Tuner: A search-optimizing GPU code auto-tuner, *Future Generation Computer Systems,* 2019

Kernel Tuner Developers

- Alessio Sclocco (eScience Center)
- Stijn Heldens (eScience center)
- Floris-Jan Willemsen (eScience Center)
- Richard Schoonhoven (CWI)
- Willem Jan Palenstijn (CWI)
- Bram Veenboer (Astron)
- Ben van Werkhoven (eScience Center)

- And many more contributors, see GitHub for the full list

Collaborate with us!

We collaborate with researchers from universities and research institutes across the Netherlands on projects in every major discipline.

We also participate in many EU-funded Horizon 2020 projects.

www.esciencecenter.nl