

Introduction to GPU Programming

Alessio Sclocco and Stijn Heldens

July 15, 2024

netherlands
eScience center

Research Software Engineer @ Netherlands eScience Center

a.sclocco@esciencecenter.nl

Background:

- 2011-2012 researcher at VU Amsterdam
 - Working on GPUs for radio astronomy
- 2012-2017 PhD at VU Amsterdam
 - "Accelerating Radio Astronomy with Auto-Tuning"
- 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
 - Real-time tracking of social insects
- 2017-now Research Software Engineer at the Netherlands eScience Center
 - Radio astronomy, climate modeling, biology, natural language processing, high-energy physics, medical imaging



- Stijn Heldens (s.heldens@esciencecenter.nl)
 - Research Software Engineer (RSE)
 - Research interests in HPC include:
GPU programming, parallel algorithms,
distributed systems, and scalable programming models.
 - background:
 - 2022-now, Research Software Engineer at Netherlands eScience Center
 - 2018-2022, PhD on scalable distributed programming models
 - 2016-2017, Researcher at University of Twente
 - 2015-2016, Researcher at Delft University of Technology
 - 2012-2015, MSc Computer science at VU University Amsterdam



Schedule

- 13:00 – 13:15 Course introduction
- 13:15 – 13:25 Introduction to GPU programming
- 13:25 – 13:40 Overview of GPU programming models
- 13:40 – 13:55 Introduction to CUDA programming
- 13:55 – 14:10 Hands-on exercise

- 14:10 – 14:25 Break

- 14:25 – 14:40 Host code
- 14:40 – 14:50 CUDA memories part 1
- 14:50 – 15:10 Hands-on exercise
- 15:10 – 15:20 CUDA memories part 2
- 15:20 – 15:50 Hands-on exercise

- 15:50 - 16:05 Break

- 16:05 – 16:25 CUDA Program execution
- 16:25 – 16:35 Optimizing a GPU application - example
- 16:35 – 16:40 Closing

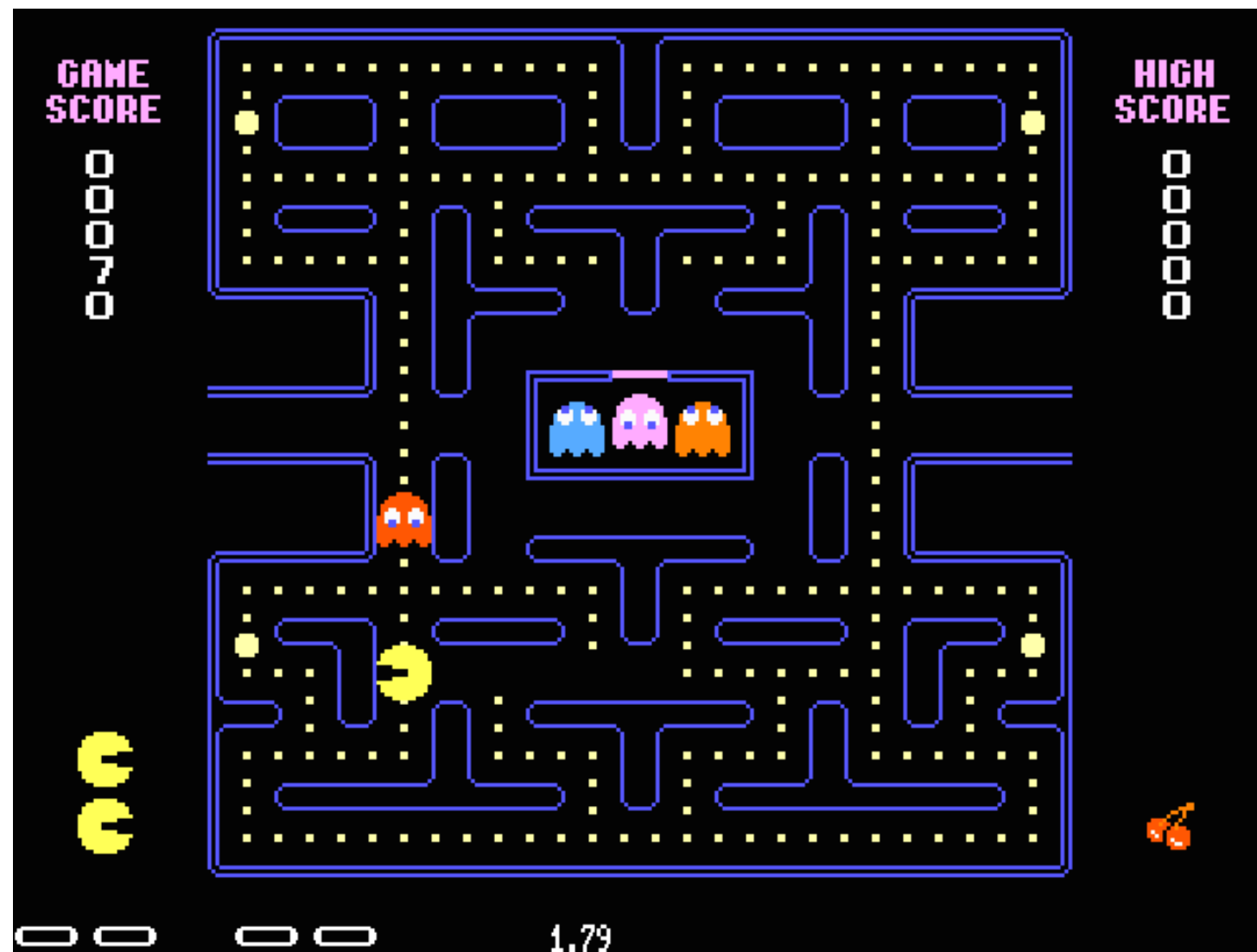
- Get your own copy of the slides so you can read along and click on links
 - See:
- Our slides are sometimes very wordy, this is intentional, so they may serve as a reference that you can read again later
- In code samples on the slides, we sometimes abbreviate the code a bit to save space

Introduction to GPU Computing

netherlands
eScience center

- Graphics Processing Unit –
The computing chip on a graphics card
- GPGPU – General Purpose computing on GPUs
 - Using GPUs for more than graphics

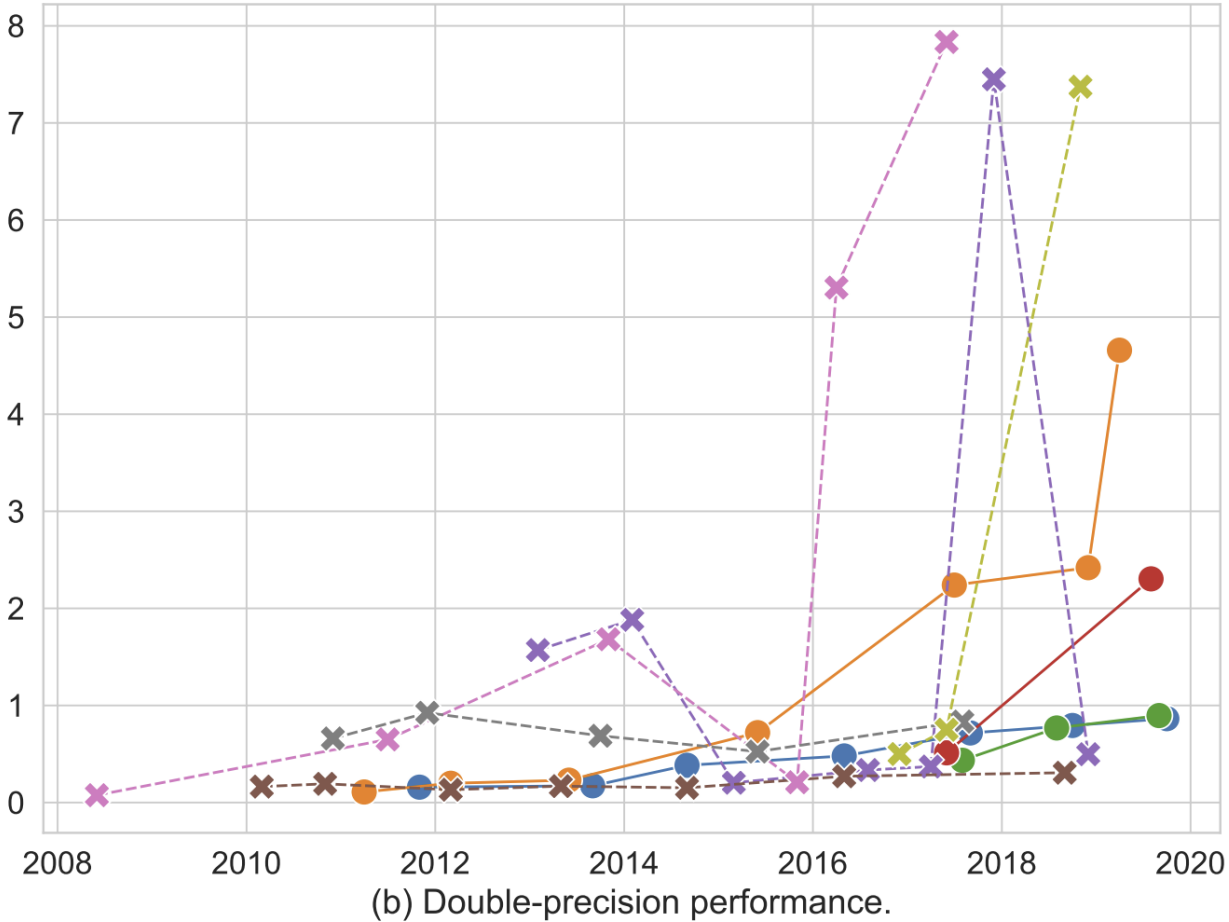
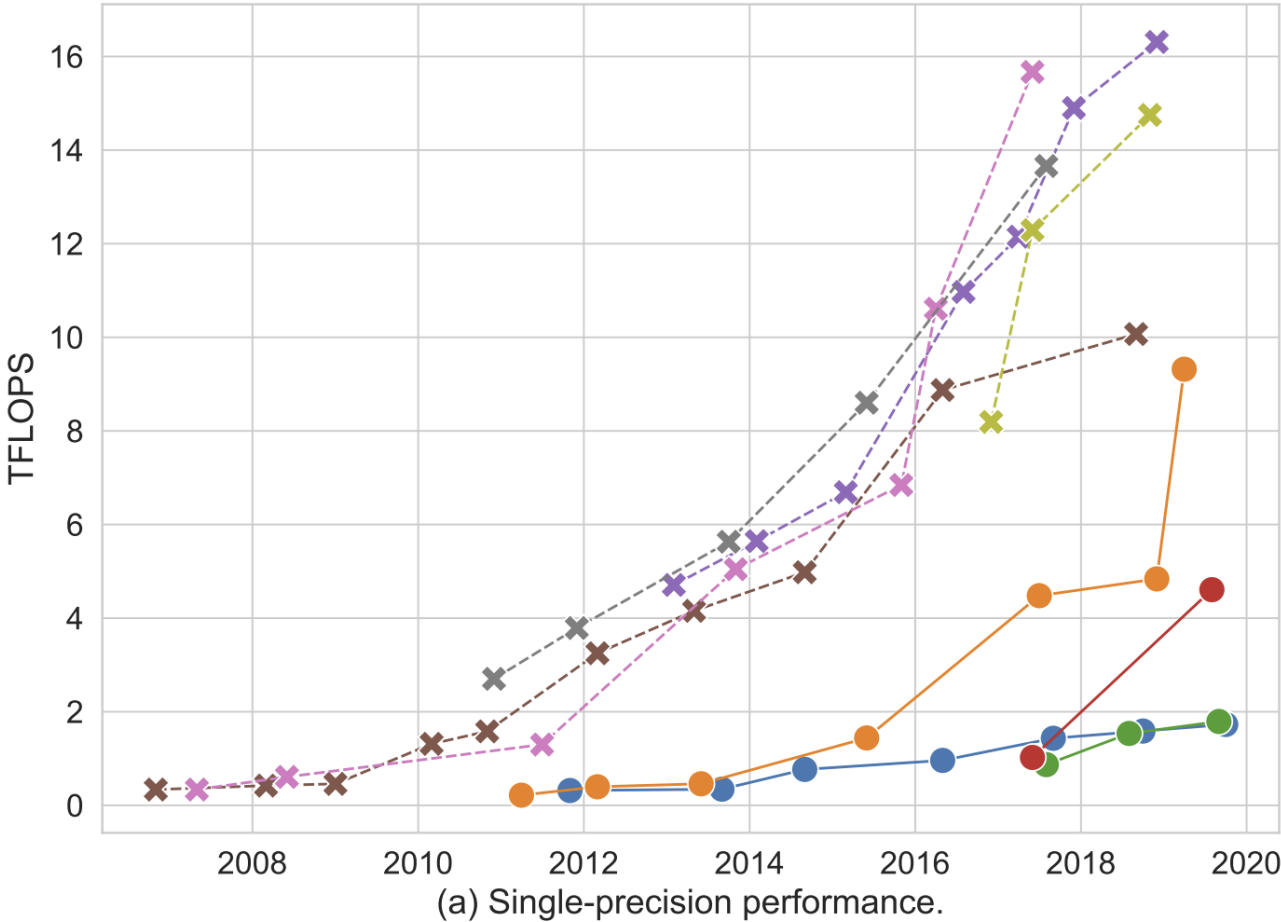
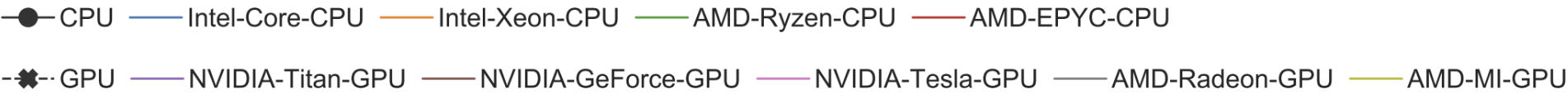




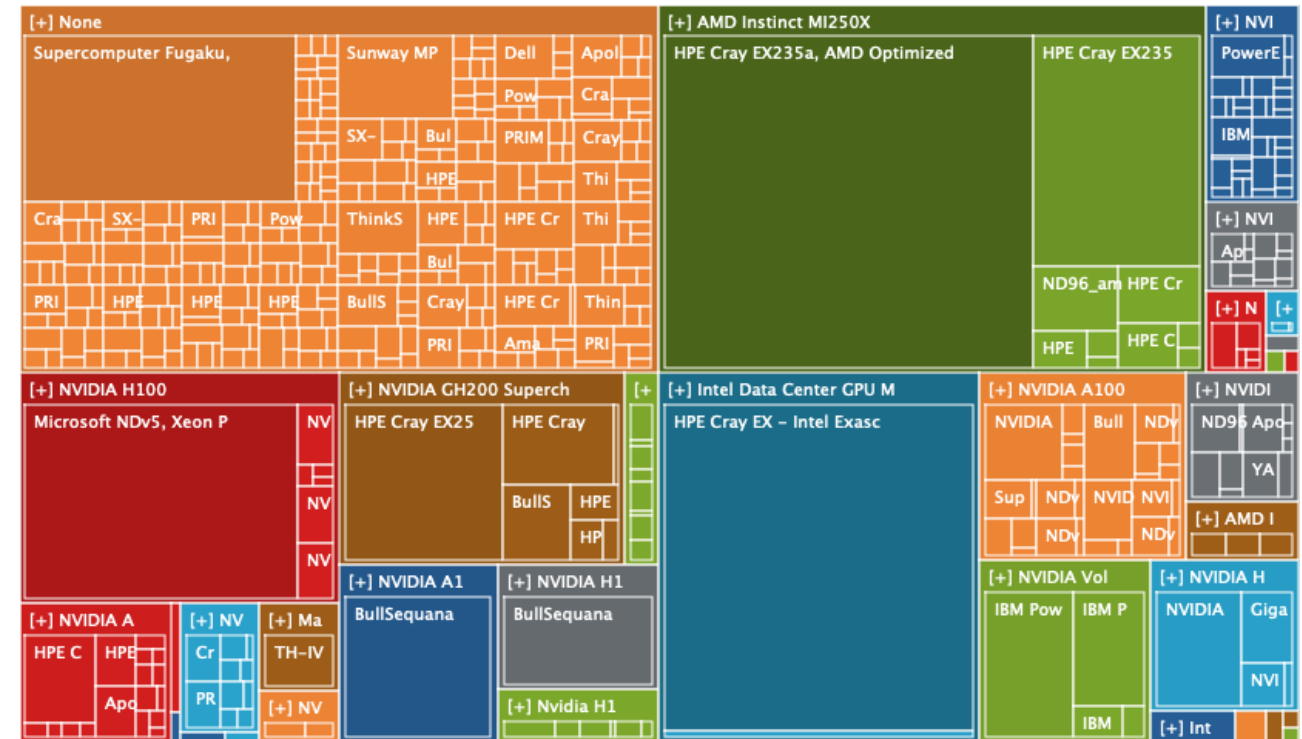




Performance Comparison: CPUs vs GPUs



- Graphics Processing Units (GPUs) are widely used to accelerate HPC applications
- On the June 2024 edition of the Top500 list:
 - The fastest supercomputer (Frontier) have GPUs
 - The two (known) exascale machines (Frontier, Aurora) have GPUs
 - The top 3 systems have GPUs
 - 9 of the top 10 systems have GPUs



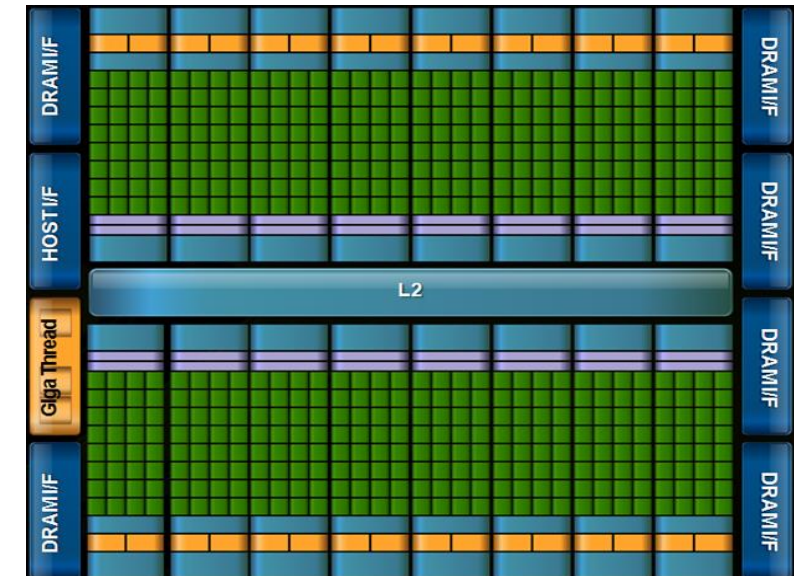
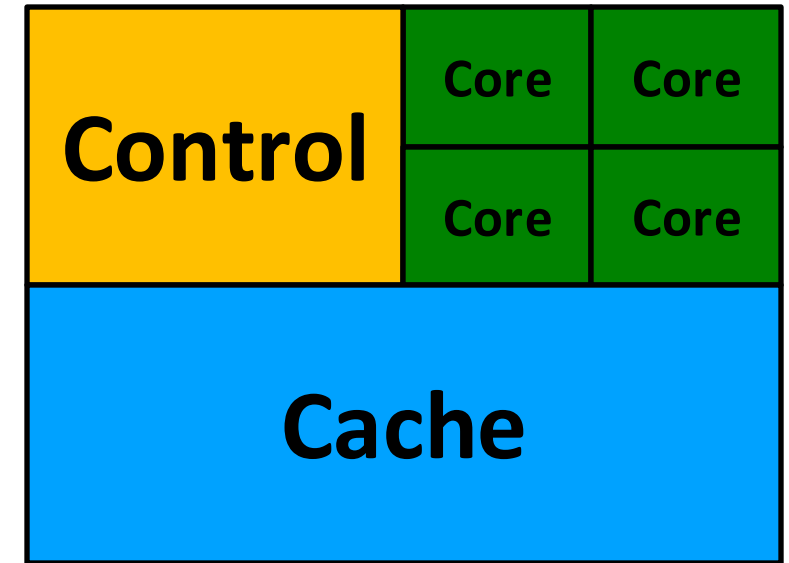
The World's Fastest Supercomputer: Frontier

Number 1 in TOP500 list (Jun 2024)

- Peak compute performance:
 - 1.6 ExaFLOPS =
 1.6×10^{18} floating point operations per second
- 9472 nodes with:
 - 1 CPU (AMD EPYC 64C)
 - 4 GPUs (AMD Instinct MI250X)



- 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

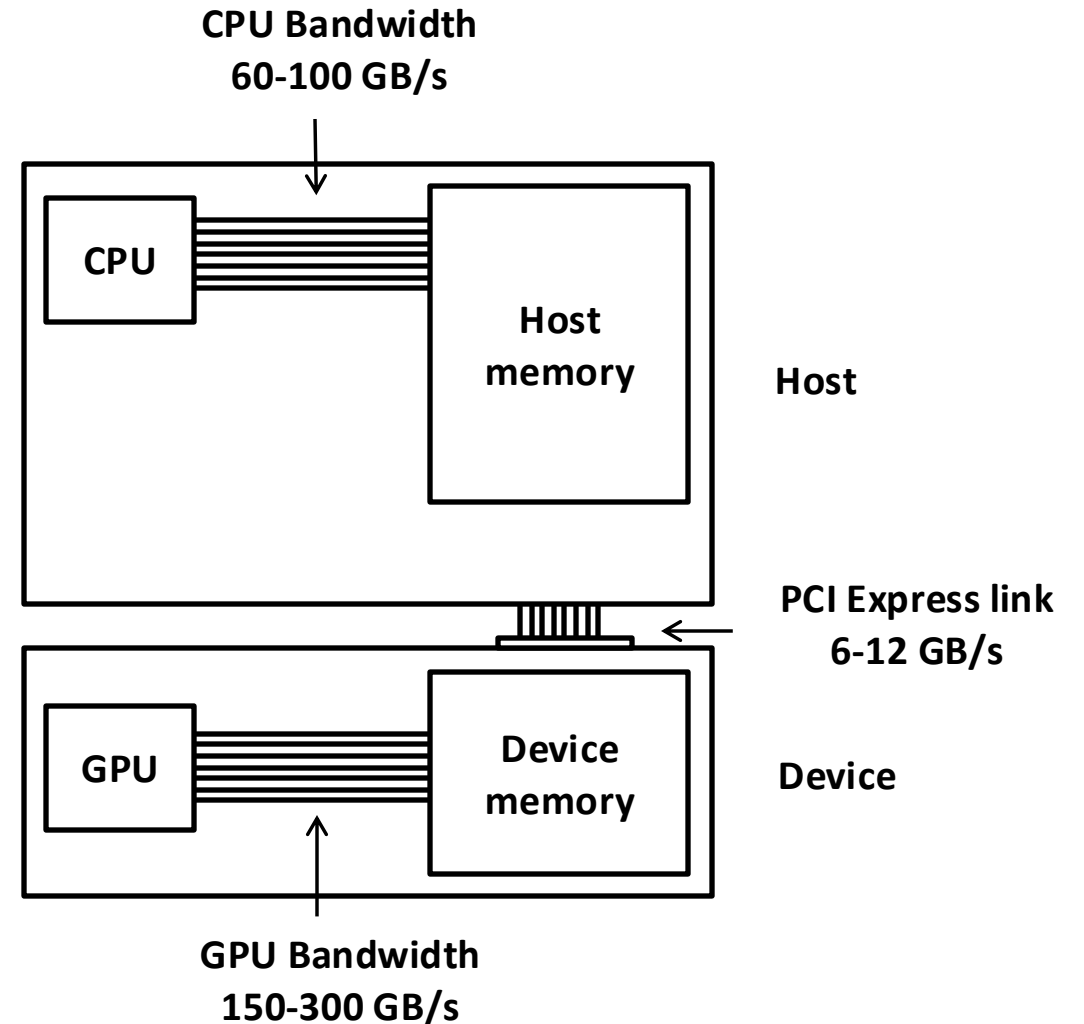


The computer architecture is very different:

- 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

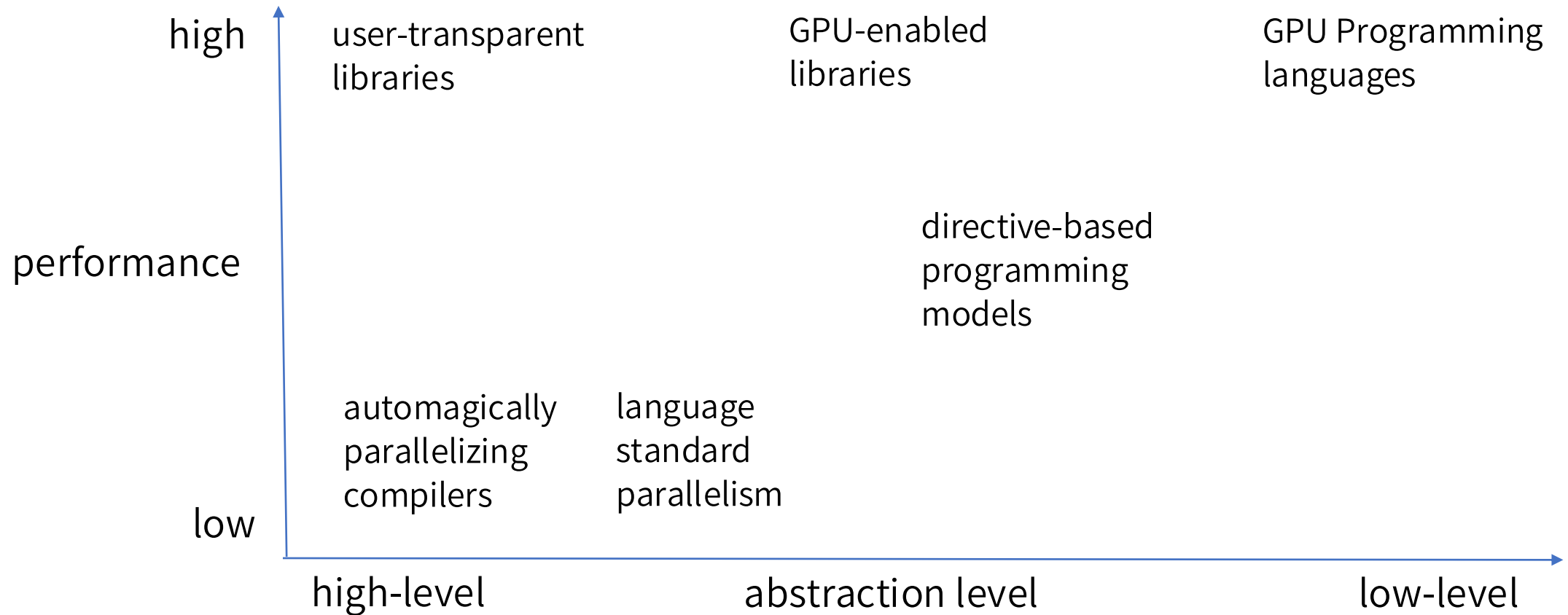
GPUs are on separate devices:

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

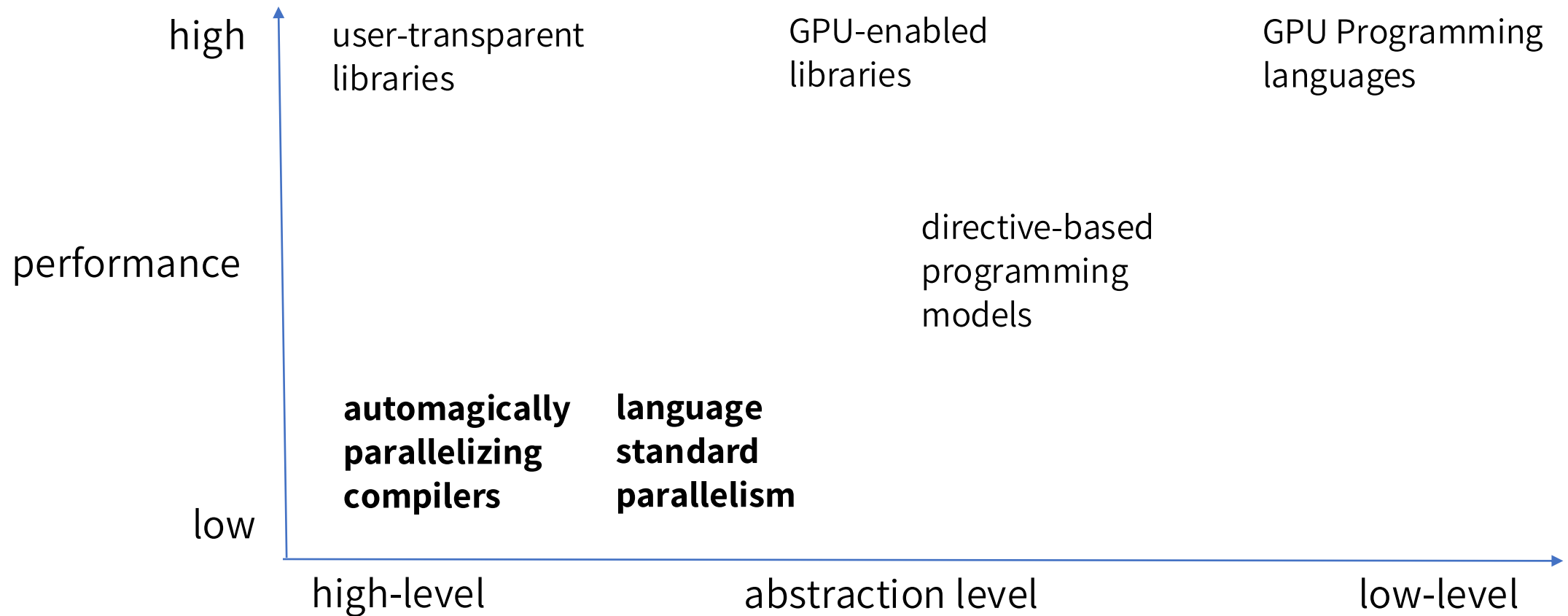


Overview of GPU Programming Models

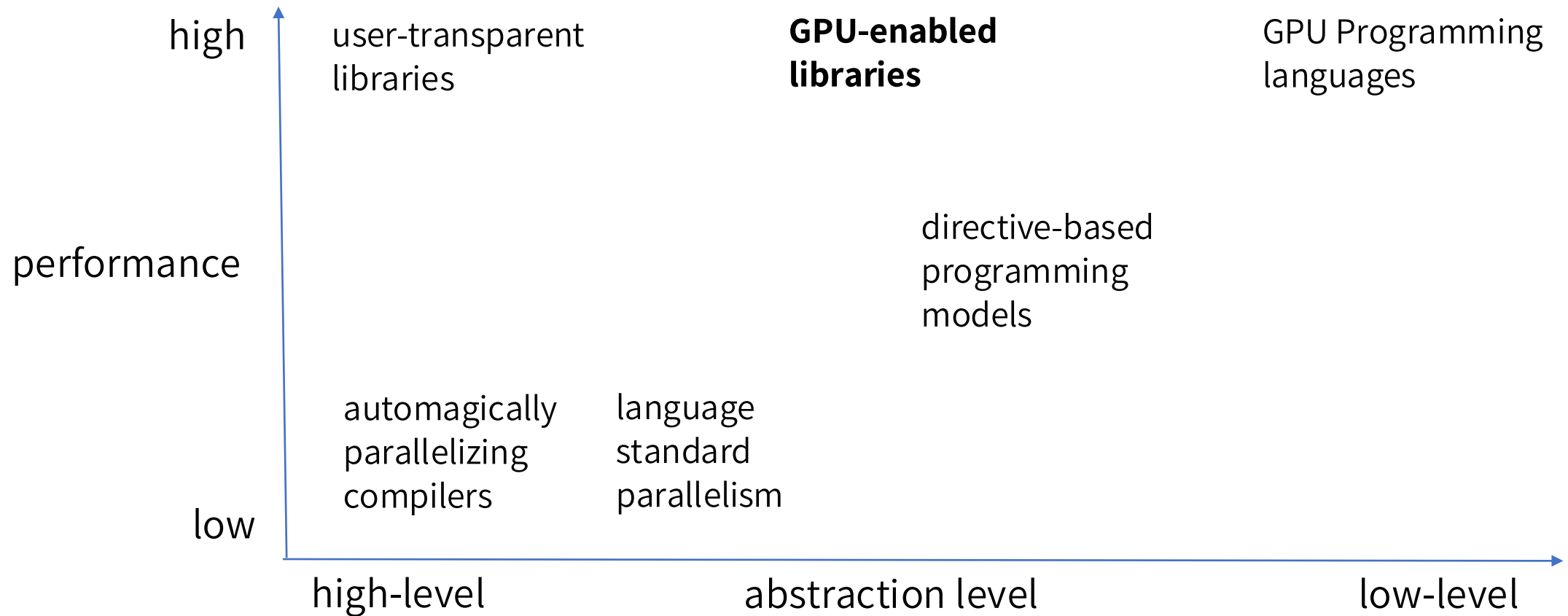
GPU Programming techniques



GPU Programming techniques



GPU Programming techniques



- 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

- 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

- **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>

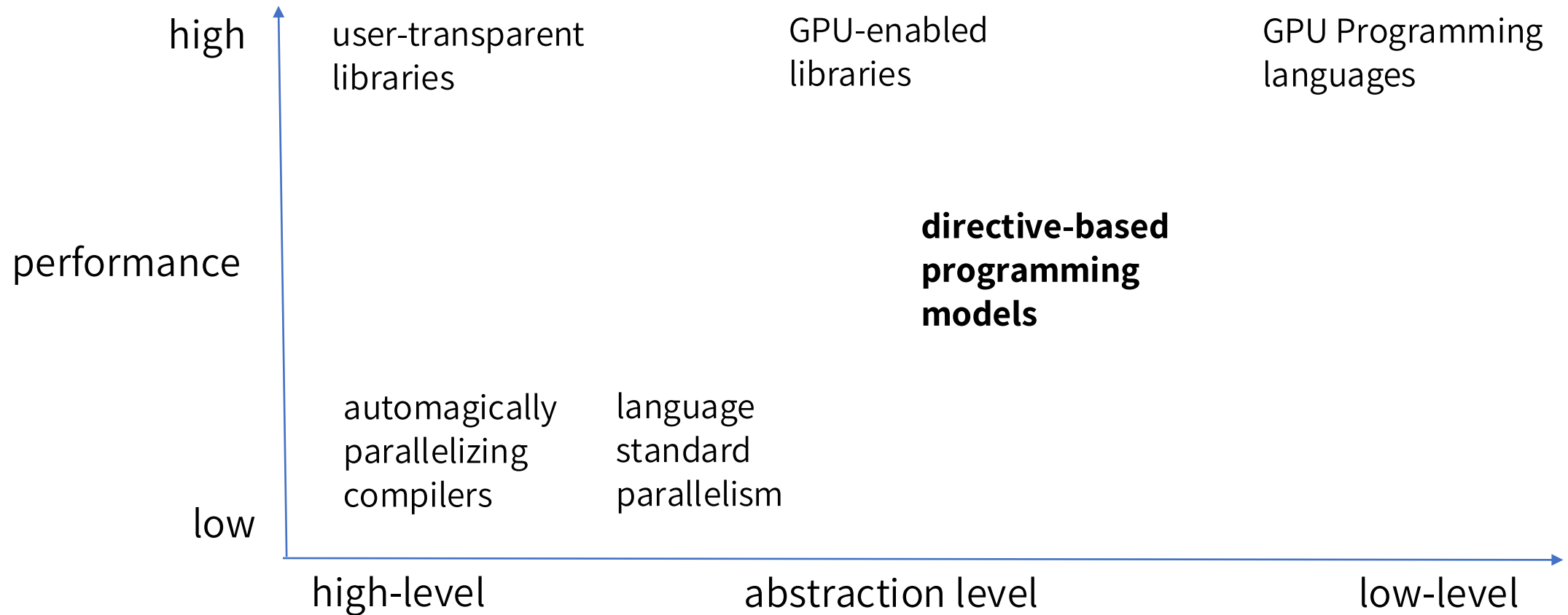
Examples on how to use libraries from AMD's ROCm platform are included in the documentation: <https://rocmdocs.amd.com>

Intel's [OneAPI toolkit](#) includes many numerical libraries and code samples

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 some C++ code that calls the library function and that can be called from another language, e.g. Fortran.

GPU Programming techniques



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

- **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 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

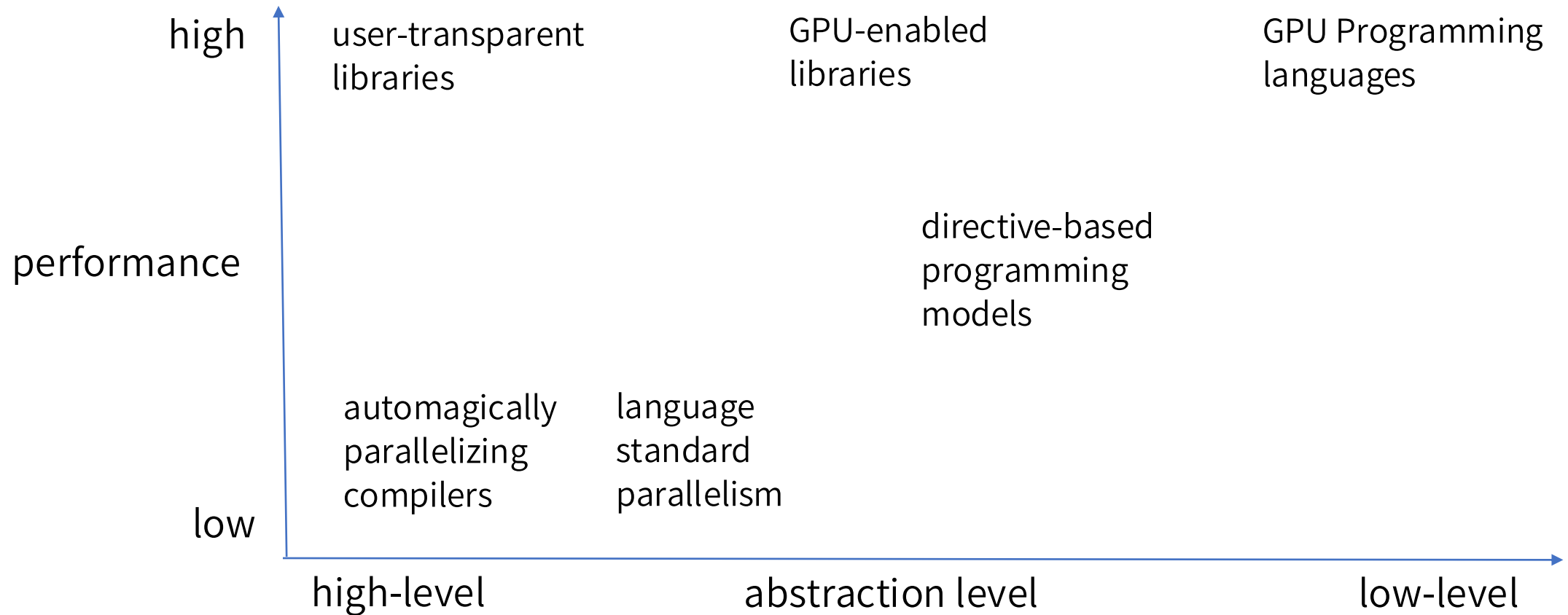
- From the OpenACC specification:

“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.

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

GPU Programming techniques: Summary



Introduction to CUDA Programming

netherlands
eScience center

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, reads 'a' once, 3 writes to a_new
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;
}
```

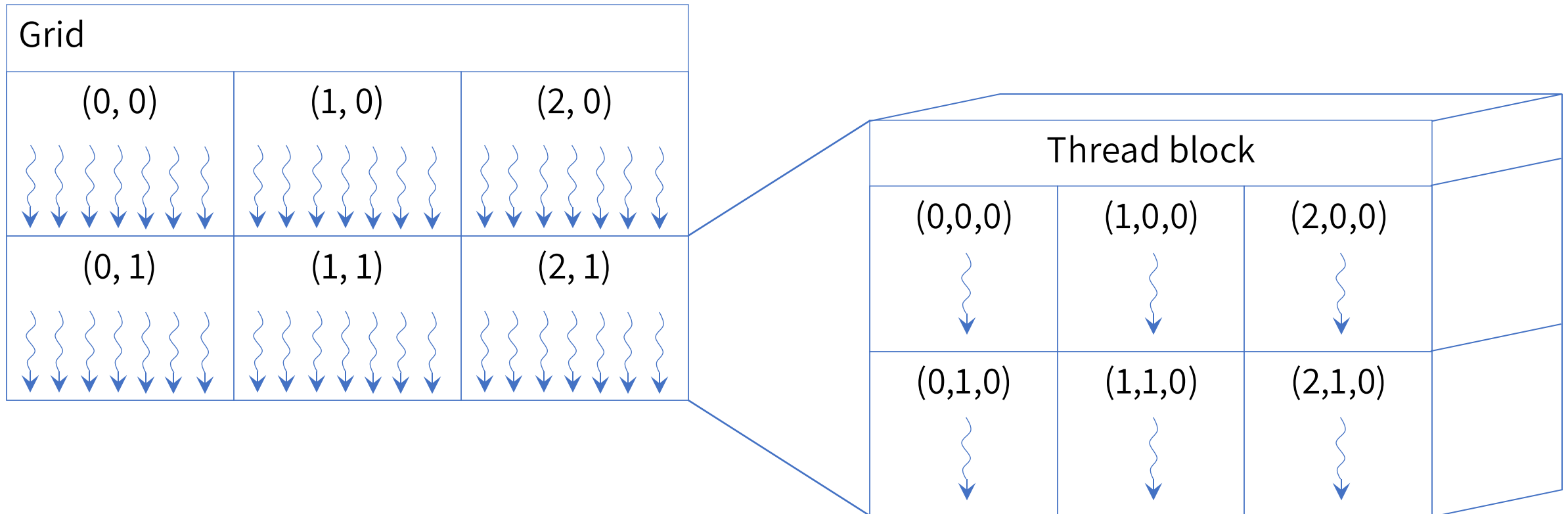


```
//almost the same, but with 3 reads for every 1 write
for (int i=1; i<N-1; i++) {
    a_new[i] = 0.25*a[i-1] + 0.5*a[i] + 0.25*a[i+1];
}
```

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

- The programming language for kernels is **CUDA**.
 - 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

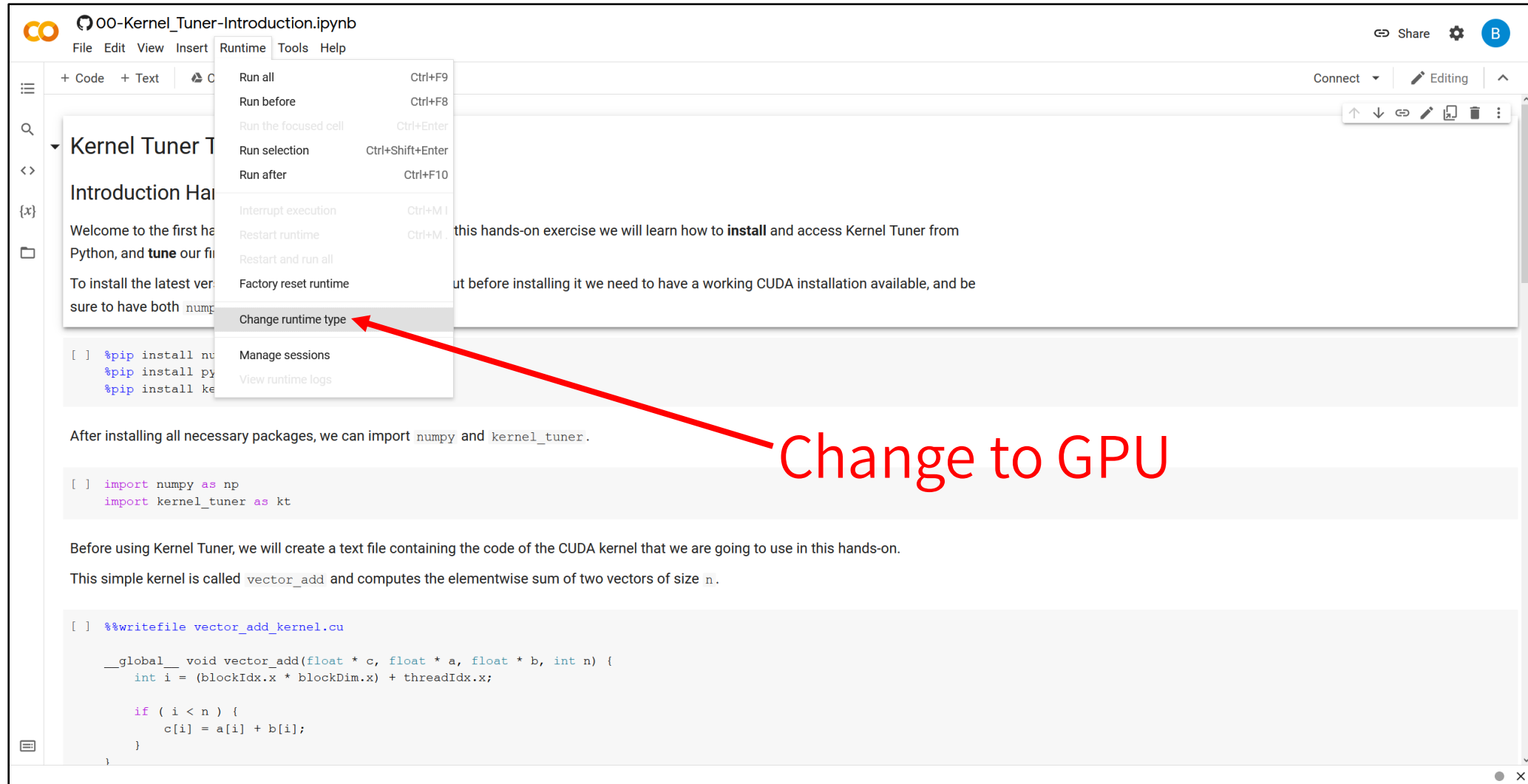
- Threads are grouped in **thread blocks**, allowing you to work on problems larger than the maximum thread block size
- Thread blocks are also numbered, using the built-in variable `blockIdx.xyz` containing the index of each block within the grid.
- Total number of threads created is always a multiple 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.xyz`

First Hands-on

netherlands
eScience center

- Make sure you understand everything in the code, and complete the exercise!
- [Open](#) the notebook in your Google Colab and work there
- 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
- The notebook is located here
 - https://github.com/benvanwerkhoven/gpu-course/blob/master/notebooks/vector_add.ipynb

Change runtime type in Colab



00-Kernel_Tuner-Introduction.ipynb

File Edit View Insert Runtime Tools Help

Run all Ctrl+F9
Run before Ctrl+F8
Run the focused cell Ctrl+Enter
Run selection Ctrl+Shift+Enter
Run after Ctrl+F10
Interrupt execution Ctrl+M
Restart runtime Ctrl+M
Restart and run all
Factory reset runtime
Change runtime type
Manage sessions
View runtime logs

Kernel Tuner T
Introduction Ha
Welcome to the first ha
Python, and **tune** our fi
To install the latest ver
sure to have both num
this hands-on exercise we will learn how to **install** and access Kernel Tuner from
but before installing it we need to have a working CUDA installation available, and be

```
[ ] %pip install nu  
%pip install py  
%pip install ke
```

After installing all necessary packages, we can import `numpy` and `kernel_tuner`.

```
[ ] import numpy as np  
import kernel_tuner as kt
```

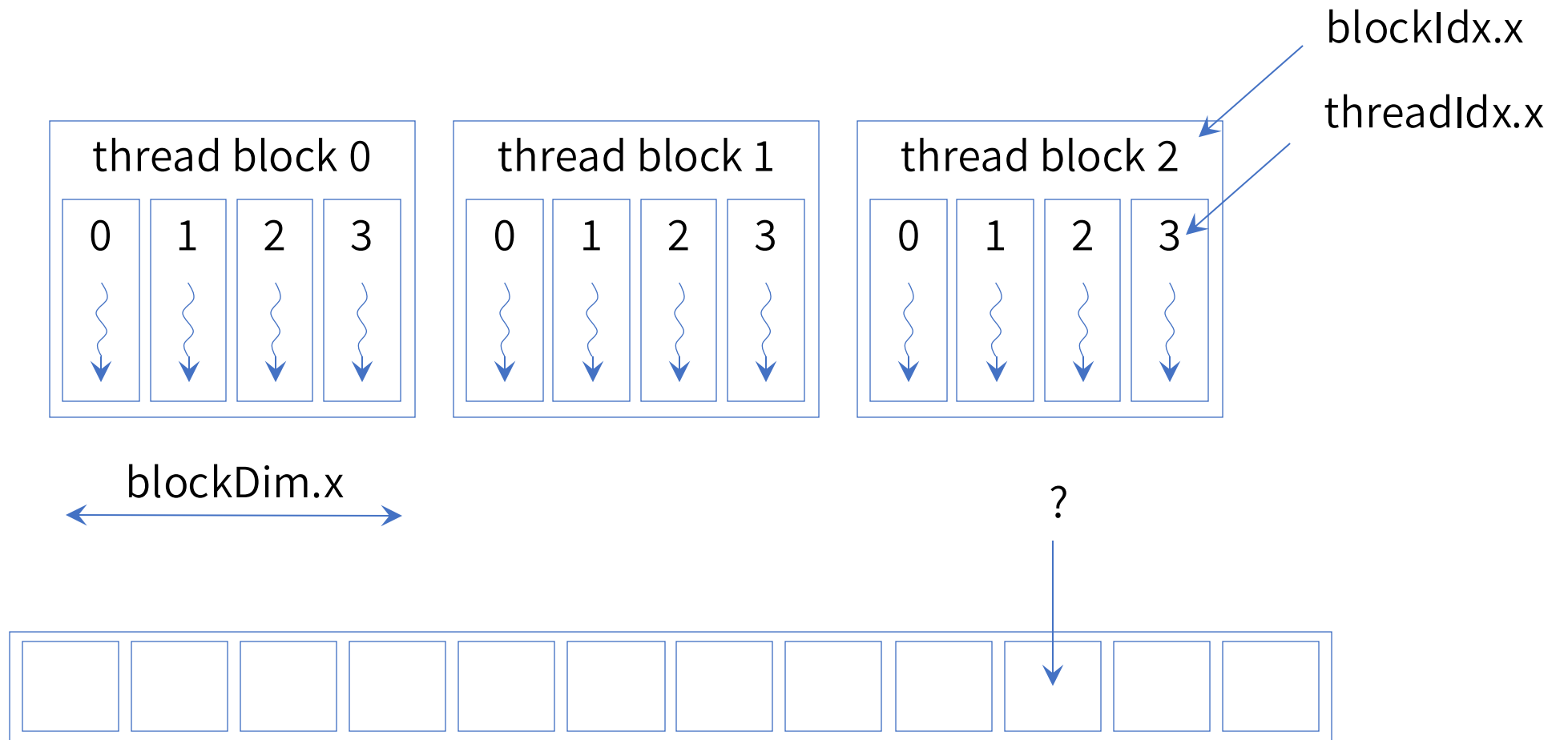
Before using Kernel Tuner, we will create a text file containing the code of the CUDA kernel that we are going to use in this hands-on.

This simple kernel is called `vector_add` and computes the elementwise sum of two vectors of size `n`.

```
[ ] %%writefile vector_add_kernel.cu  
  
__global__ void vector_add(float * c, float * a, float * b, int n) {  
    int i = (blockIdx.x * blockDim.x) + threadIdx.x;  
  
    if ( i < n ) {  
        c[i] = a[i] + b[i];  
    }  
}
```

Change to GPU

Hint



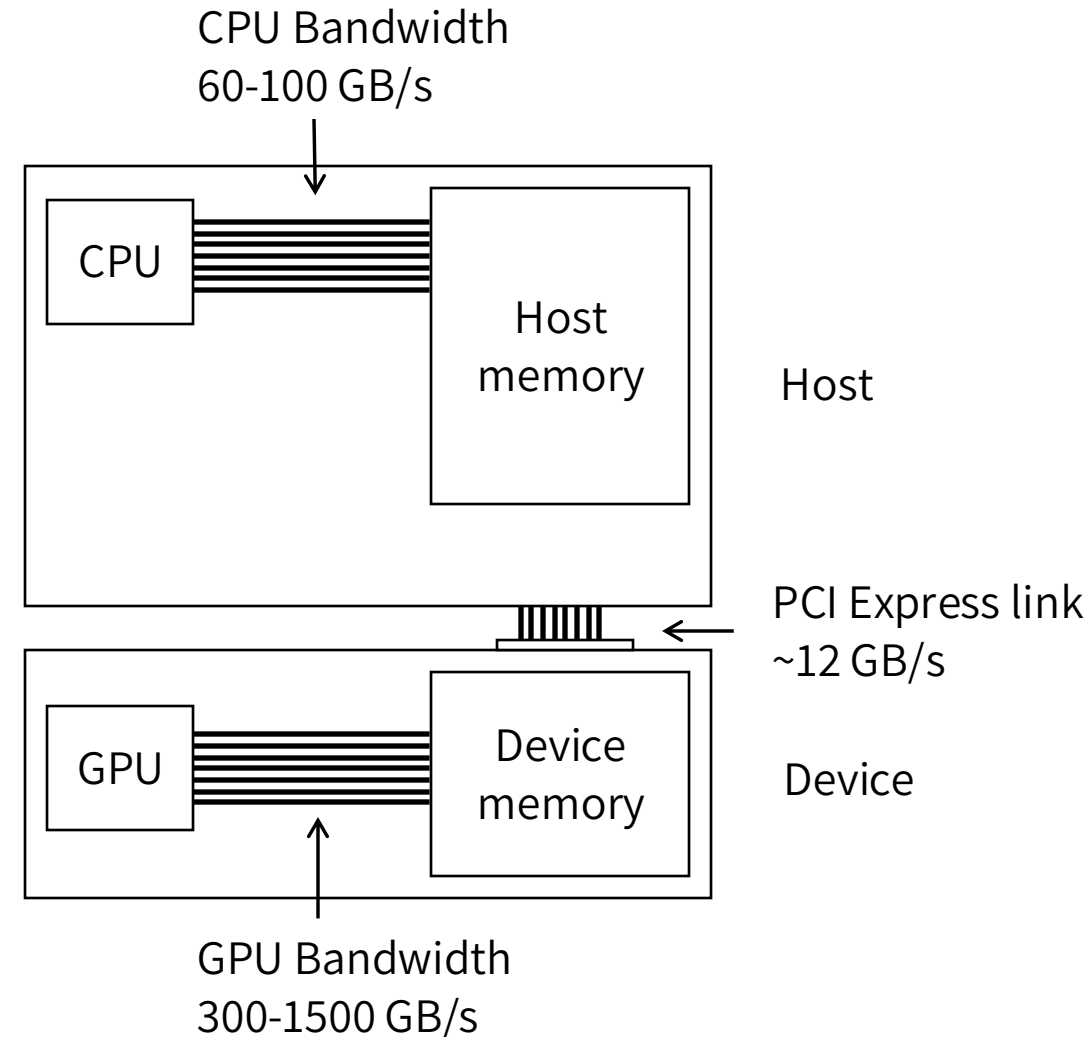
A close-up photograph of a white ceramic coffee cup with a handle on the left. A stream of dark brown coffee is being poured from above into the cup, creating a dynamic splash and ripples on the surface of the liquid already inside. The background is a warm, out-of-focus wooden surface. The text "Coffee break" is centered over the cup in a white, sans-serif font.

Coffee break

Host Code

netherlands
eScience center

- 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
- The **device** code runs on the GPU
 - Runs threads that execute your kernel code



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

- 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

- 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

- 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

- 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`

- `cudaError_t cudaMallocManaged(void** devPtr, size_t size, unsigned int flags = cudaMemAttachGlobal)`
- Allocates memory that will be managed automatically
 - Data is automatically moved between host and device memory
 - Manual transfers from/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();
```

- 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()
```

- 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()
```


CUDA Memories (part 1)

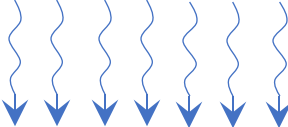
CUDA memory hierarchy

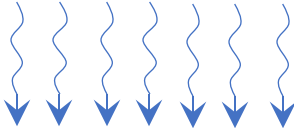
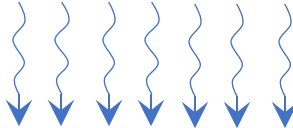
Registers

Shared memory

Global memory
Constant memory

Thread


Thread
Block


| Grid | |
|---|---|
| (0, 0) | (1, 0) |
|  |  |

- 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

- 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

```
__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`

Second Hands-on

netherlands
eScience center

- Make sure you understand everything in the code, and complete the exercise!
- [Open](#) the notebook in your Google Colab and work there
- 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](#)
- The notebook is located here
 - <https://github.com/benvanwerkhoven/gpu-course/blob/master/notebooks/pnpoly.ipynb>


CUDA Memories (part 2)

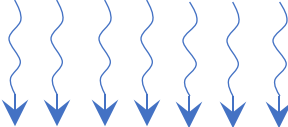
CUDA memory hierarchy

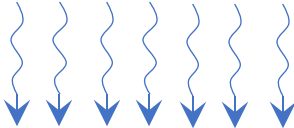
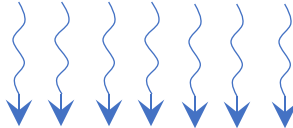
Registers

Shared memory

Global memory
Constant memory

Thread


Thread
Block


| Grid | |
|---|---|
| (0, 0) | (1, 0) |
|  |  |

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 (threadIdx.x < NUM_BINS)
        sh_output[threadIdx.x] = 0;                    //initialize shared memory
    __syncthreads();                                    //wait for all threads

    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

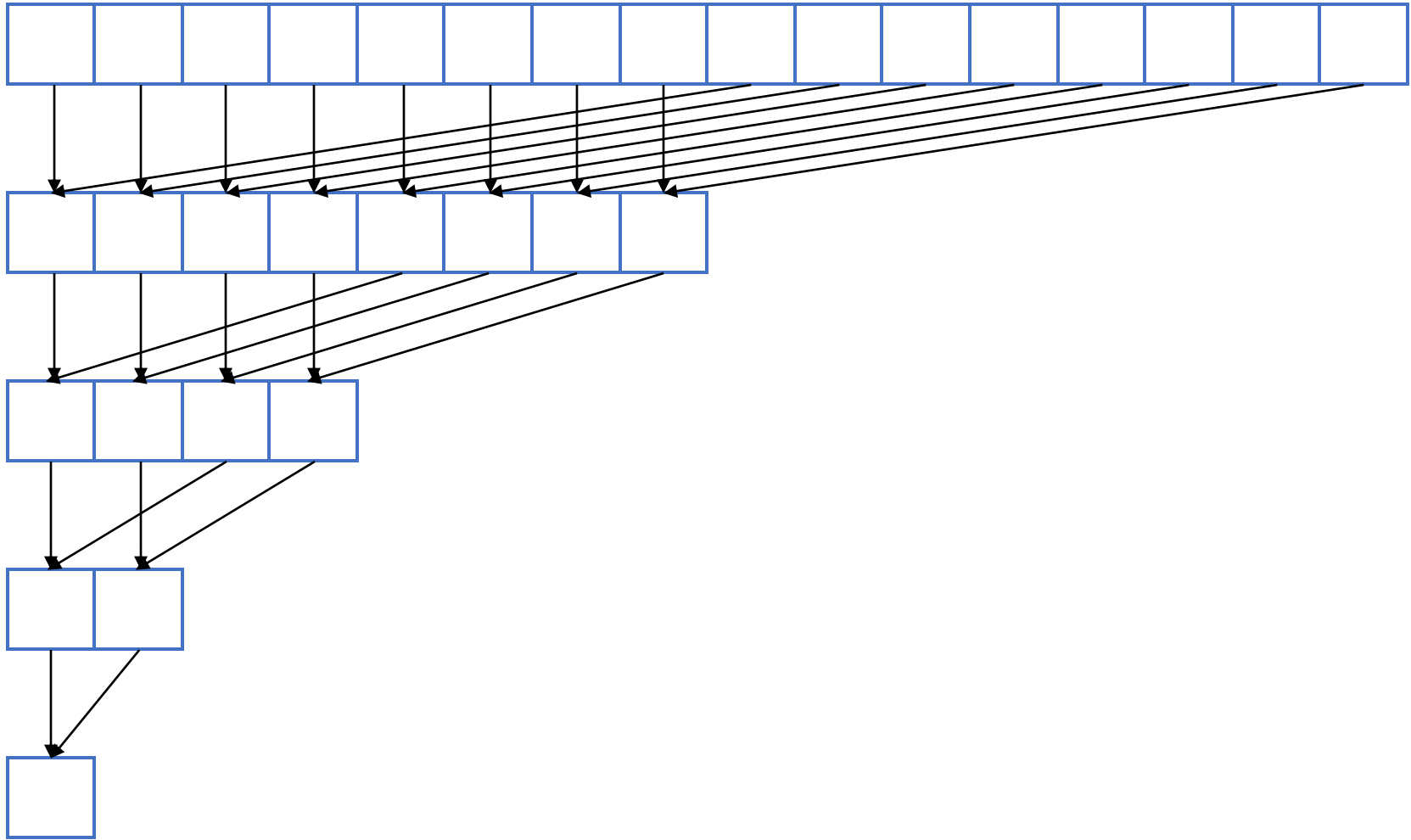
    i = threadIdx.x + blockIdx.y * block_size_y;
    j = threadIdx.y + blockIdx.x * block_size_x;
    if (j < w && i < h) {
        output[j*h+i] = sh_mem[threadIdx.x][threadIdx.y];    //store to global using shared memory
    }
}
```


Third Hands-on

netherlands
eScience center

- 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!
- [Open](#) the notebook in your Google Colab and work there
- 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
- The notebook is located here
 - <https://github.com/benvanwerkhoven/gpu-course/blob/master/notebooks/reduction.ipynb>

Hint – Parallel Summation

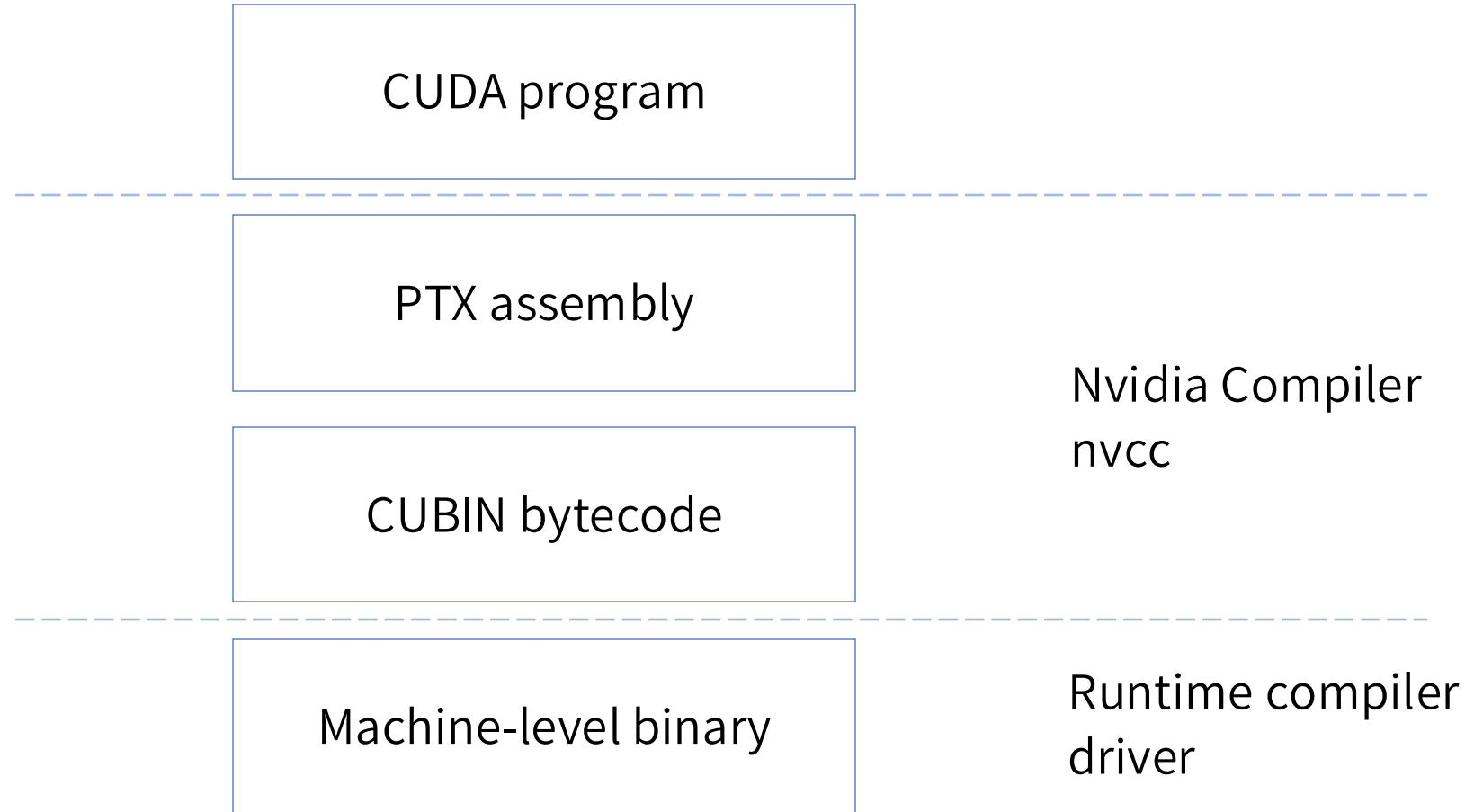


A close-up photograph of a white ceramic coffee cup with a handle on the left. A stream of dark brown coffee is being poured from above into the cup, creating a dynamic splash and ripples on the surface of the liquid already in the cup. The background is a warm, out-of-focus wooden surface. The text "Coffee break" is centered over the cup in a white, sans-serif font.

Coffee break

CUDA Program Execution

netherlands
eScience center



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

- 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

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

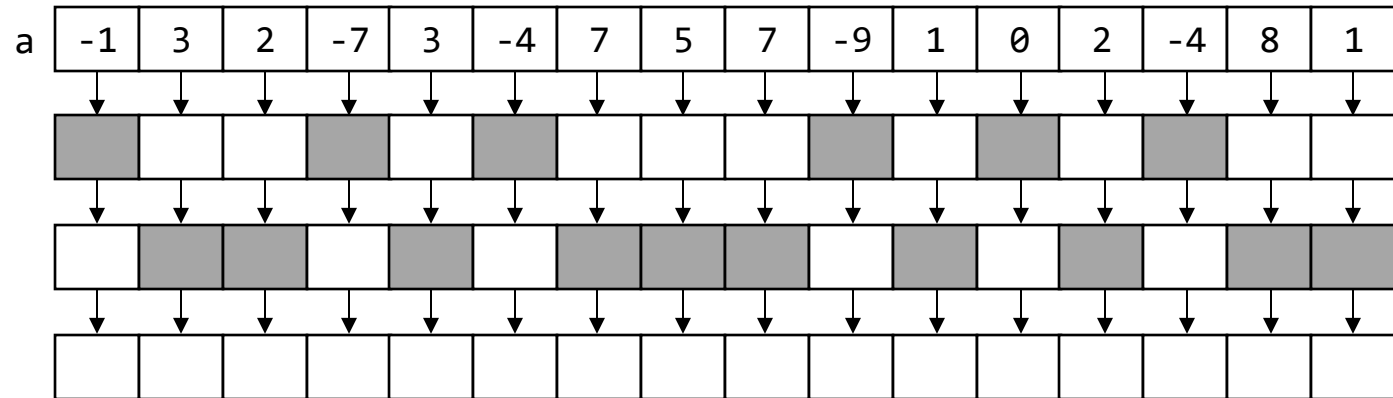
```
mad.f32    %f1, %f2, %f3, %f1;          // c += a*b;
```
- The same instruction, but on different data
- What about control flow instructions? (if, else, for, while)
 - All threads in the warp execute all live paths, with some threads predicated

```
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

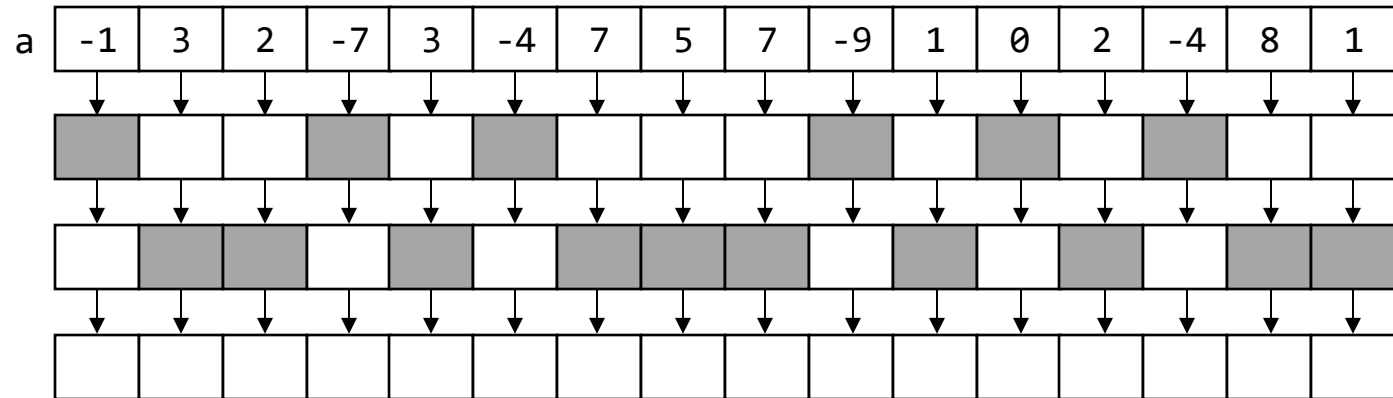
```
if (threadIdx.x < 32)
```
- The Volta architecture replaces predication with a per-thread program counter and call stack. The same performance recommendations apply, however.

Reducing Branch Divergence

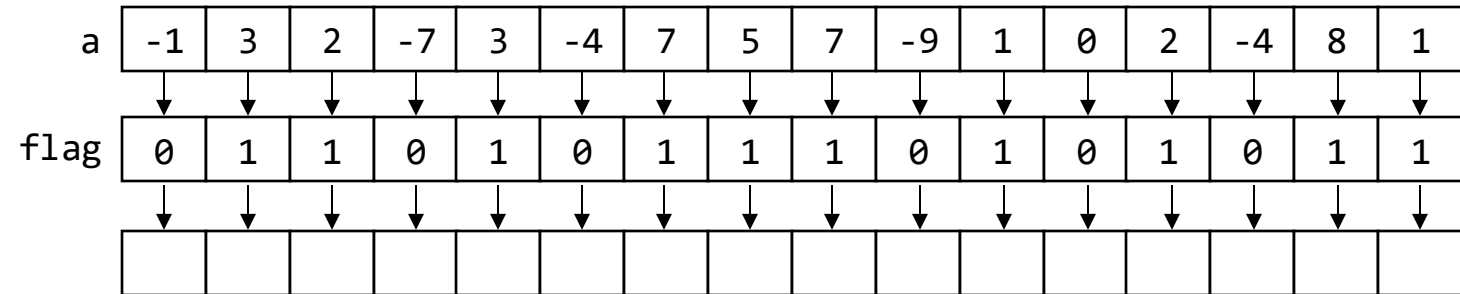
```
if a>0
  c = a*a+b
else
  c = a*a-b
```



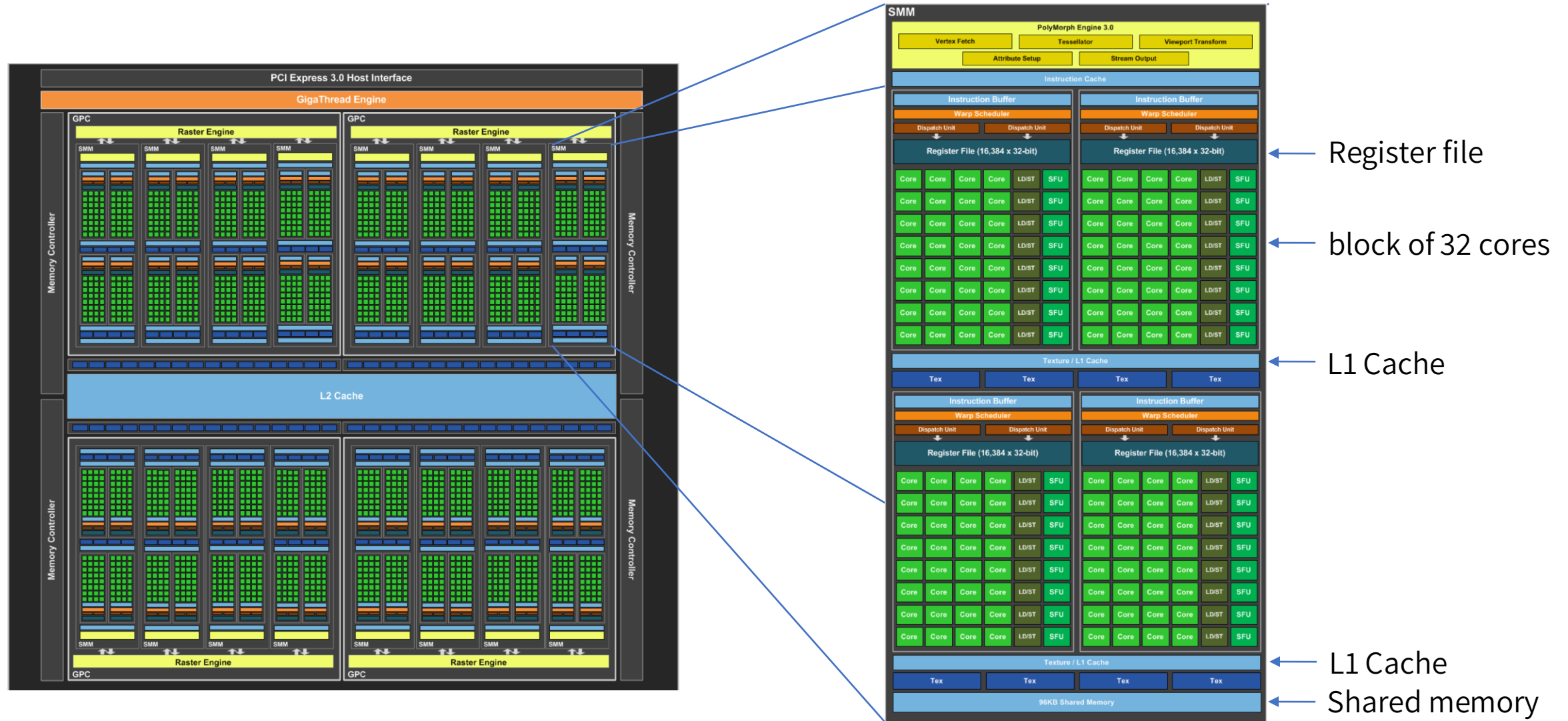
```
if a>0
    c = a*a+b
else
    c = a*a-b
```



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



Maxwell Architecture



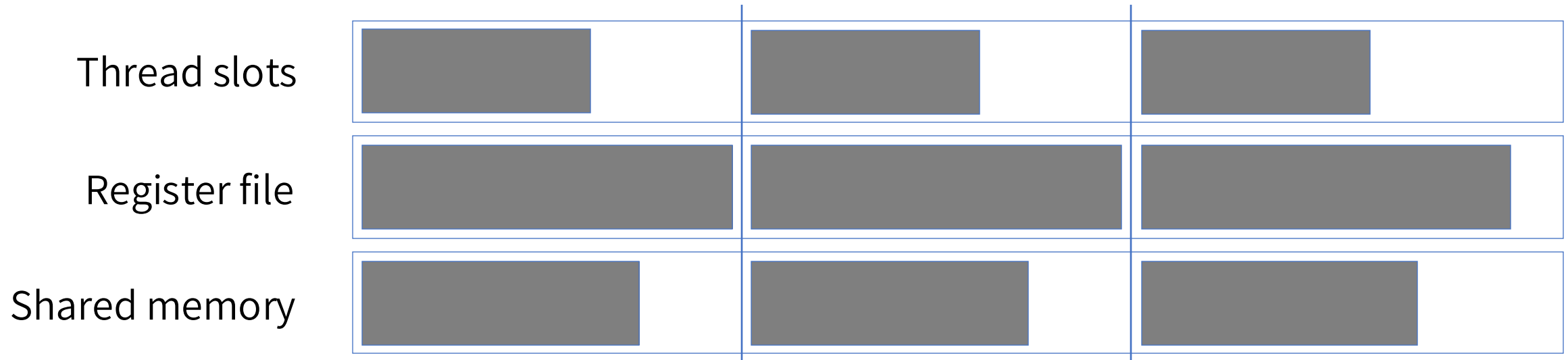
Turing architecture

- 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

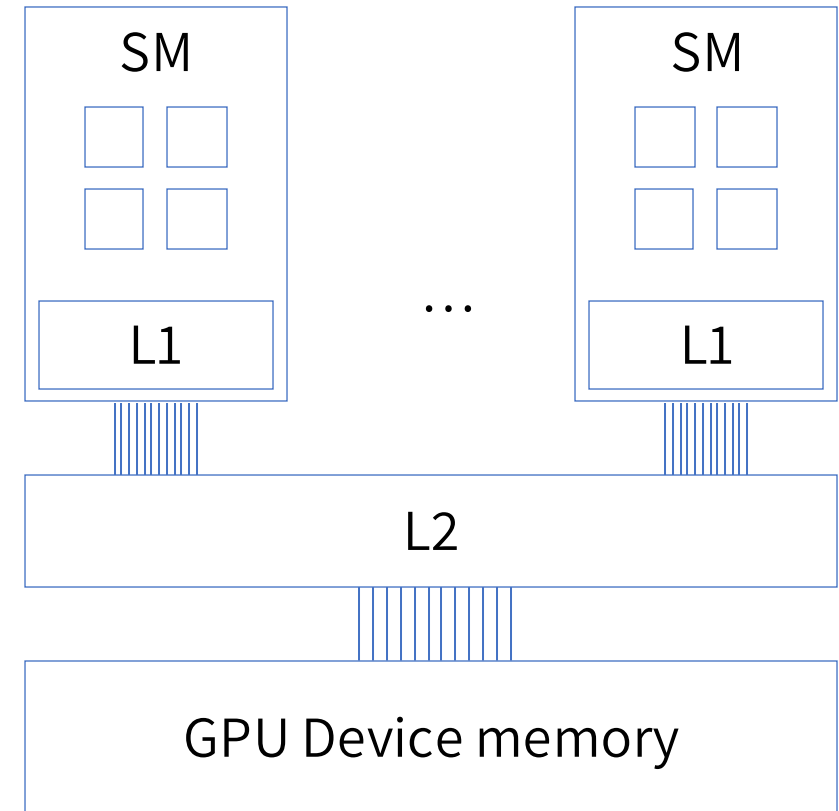


Resource partitioning

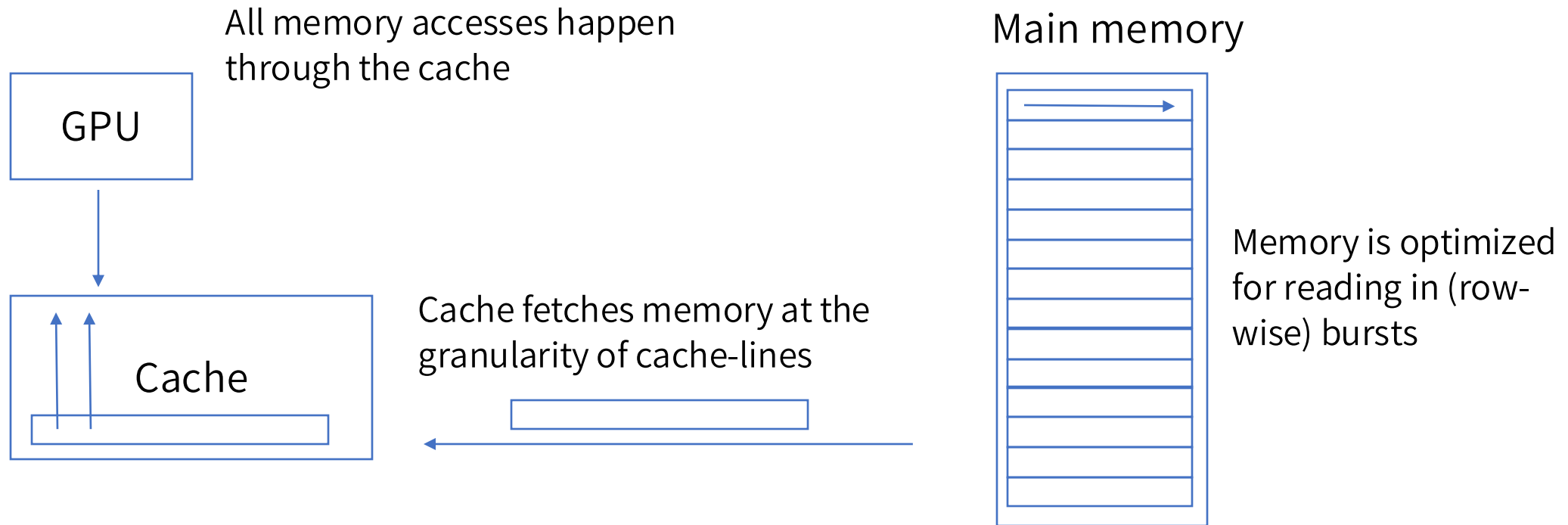
- The GPU consists of several (1 to 80) *streaming multiprocessors* (SMs)
- The SMs are fully independent
- Each SM contains several resources: Register file, Shared memory, Thread Slots, and Thread Block slots
- SM resources are dynamically partitioned among the thread blocks that execute concurrently 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

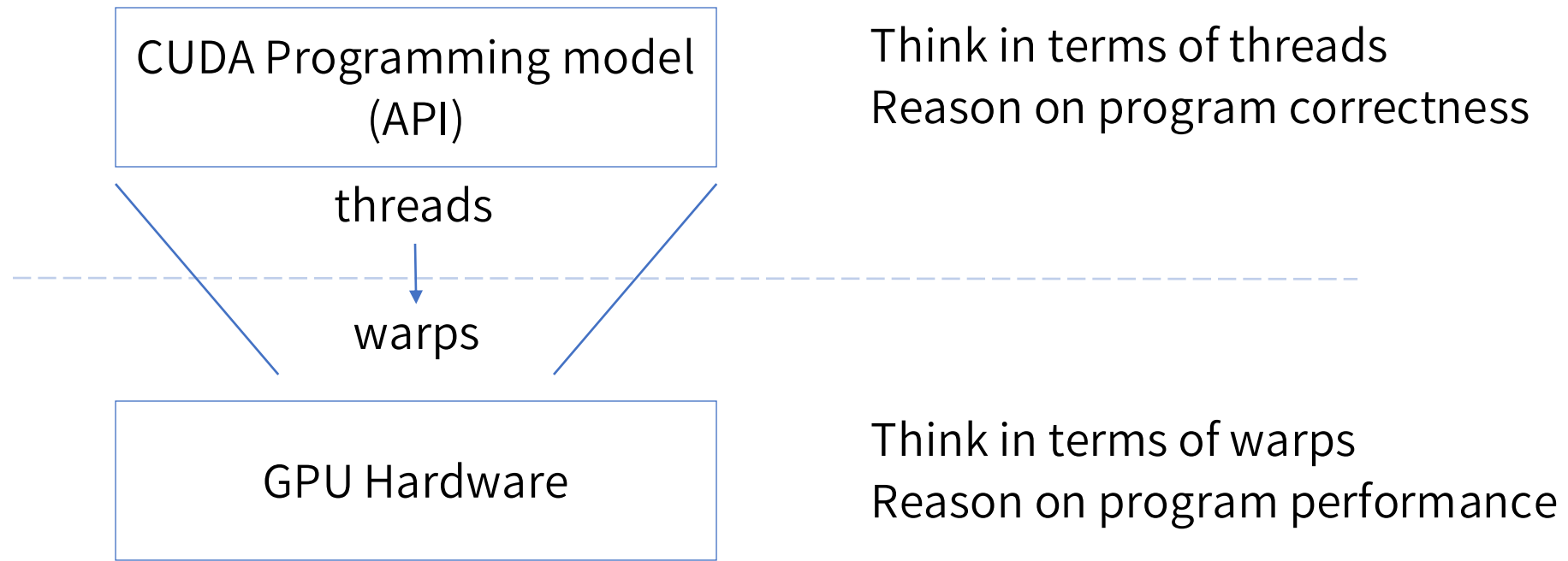


The memory hierarchy is optimized for certain access patterns



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 the data
- 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)



Optimizing a GPU Application

Gijs van den Oord, Alessio Sclocco, Bart van Stratum, Menno Veerman, Georges-Emmanuel Moulard, David Guibert, Erwan Raffin, Ben van Werkhoven, Chiel van Heerwaarden

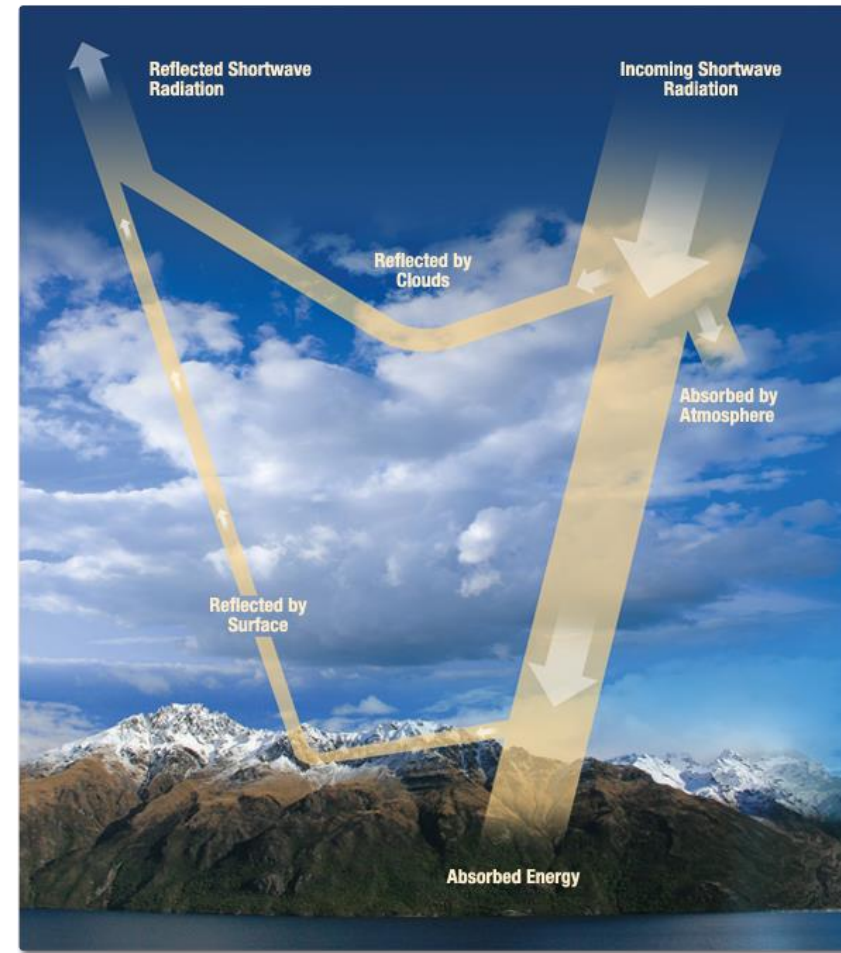
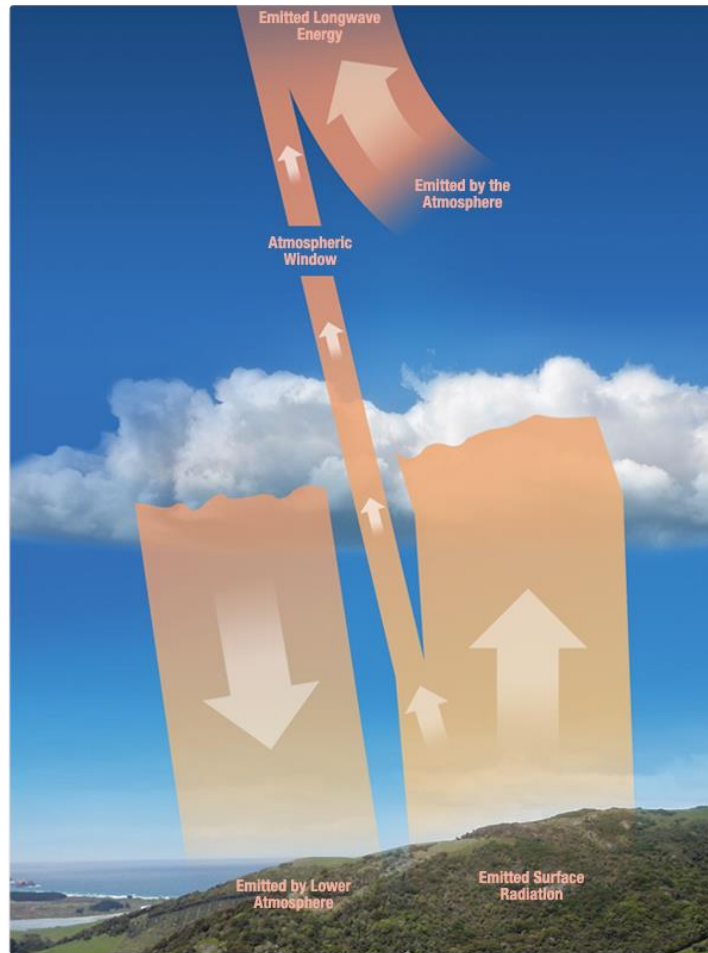
netherlands
eScience center



ESIWACE2 has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 823988

Radiative transfer code

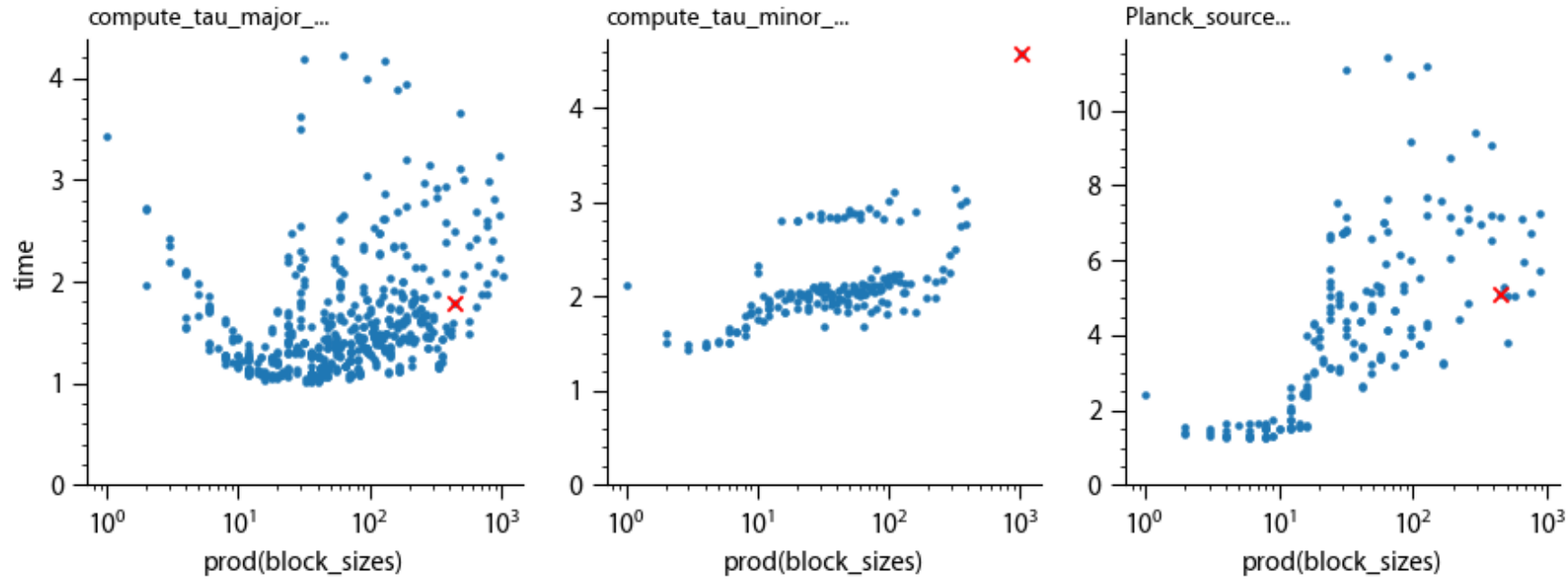
- Radiative Transfer for Energetics (RTE) + Rapid Radiative Transfer Model for GCM-Parallel (RRTMGP)
- Used by many large-scale atmospheric and earth system models for climate simulations



- The GPU version implemented in CUDA and the original CPU version in Fortran performed roughly similar
- Two main performance issues:
 - Many calls to CUDA kernels required GPU memory allocations and frees
 - The CUDA kernels were not yet optimized for performance

- Issue: Several kernels require temporary data structures in GPU memory, which are allocated and freed every time these kernels are called
- Solution: Using a **memory pool**, we can reuse GPU memory allocations
- 2nd solution: As of CUDA 11.3, CUDA supports `cudaMallocAsync` to allocate from a memory pool managed by the CUDA runtime

- Red x marks the originally used thread block dimension
- Tuning revealed underlying performance issues

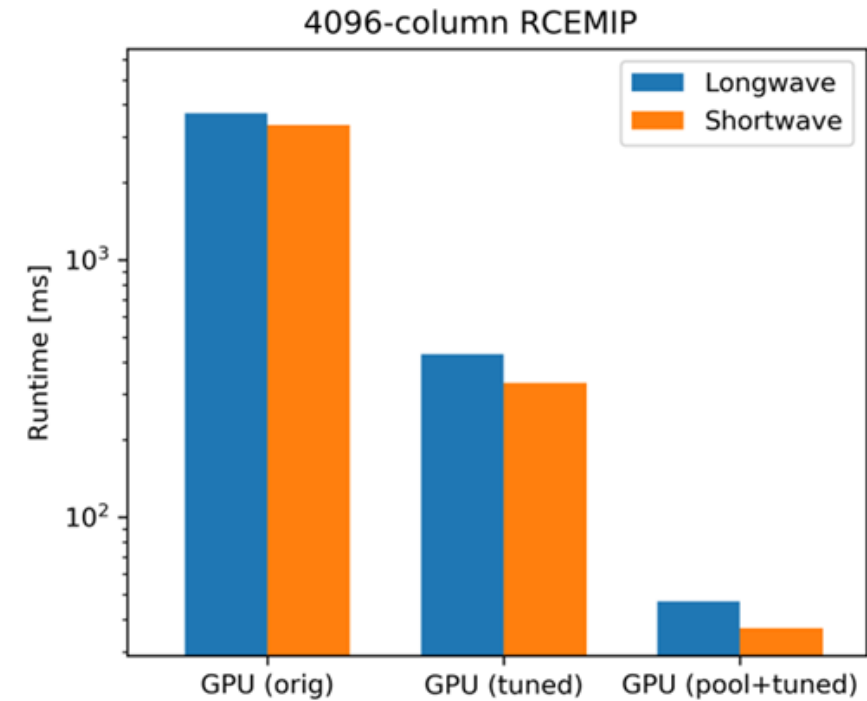


Code optimizations that we applied in various kernels:

- Reordering the parallelization and mapping to threads
- Selecting the thread block dimensions
- Fusing kernels
- Removing the need to store data in GPU memory (reducing the need for allocs/frees in GPU memory)
- Reducing register usage
- Partial Loop unrolling
- Moving data to shared memory for efficient reuse

Used Kernel Tuner to automatically select optimal combination of all of the above

- Improved performance by roughly two orders of magnitude
 - ~ 10x by using a memory pool
 - ~ 10x by optimizing and auto-tuning CUDA kernels



Closing Remarks

- Read these slides again
 - And follow the links to the documentation!
- Play with the notebooks
 - If you do not have access to GPUs, try [Google Colab](#), they have GPU support
- Check the C code for the exercises
- Check out Kernel Tuner (https://github.com/KernelTuner/kernel_tuner)
- If you have questions, contact us
 - a.sclocco@esciencecenter.nl
 - s.heldens@esciencecenter.nl

- *Optimization Techniques for GPU Programming*
P. Hijma, S. Heldens, A. Sclocco, B. van Werkhoven, and H.E. Bal
ACM Computing surveys 2022
<https://dl.acm.org/doi/abs/10.1145/3570638>
- *Lessons Learned in a Decade of Research Software Engineering GPU Applications*
B. van Werkhoven, W.J. Palenstijn, A. Sclocco
International Conference on Computational Science (ICCS 2020)
https://doi.org/10.1007/978-3-030-50436-6_29
- *Kernel Tuner: A search-optimizing GPU code auto-tuner*
B. van Werkhoven
Future Generation Computer Systems 2019
<https://doi.org/10.1016/j.future.2018.08.004>

Acknowledgments

ESiWACE3 is funded by the European Union. This work has received funding from the European High Performance Computing Joint Undertaking (JU) and Spain, Netherlands, Germany, Sweden, Finland, Italy and France, under grant agreement No 1010930.

