

# NLeSC GPU Workshop

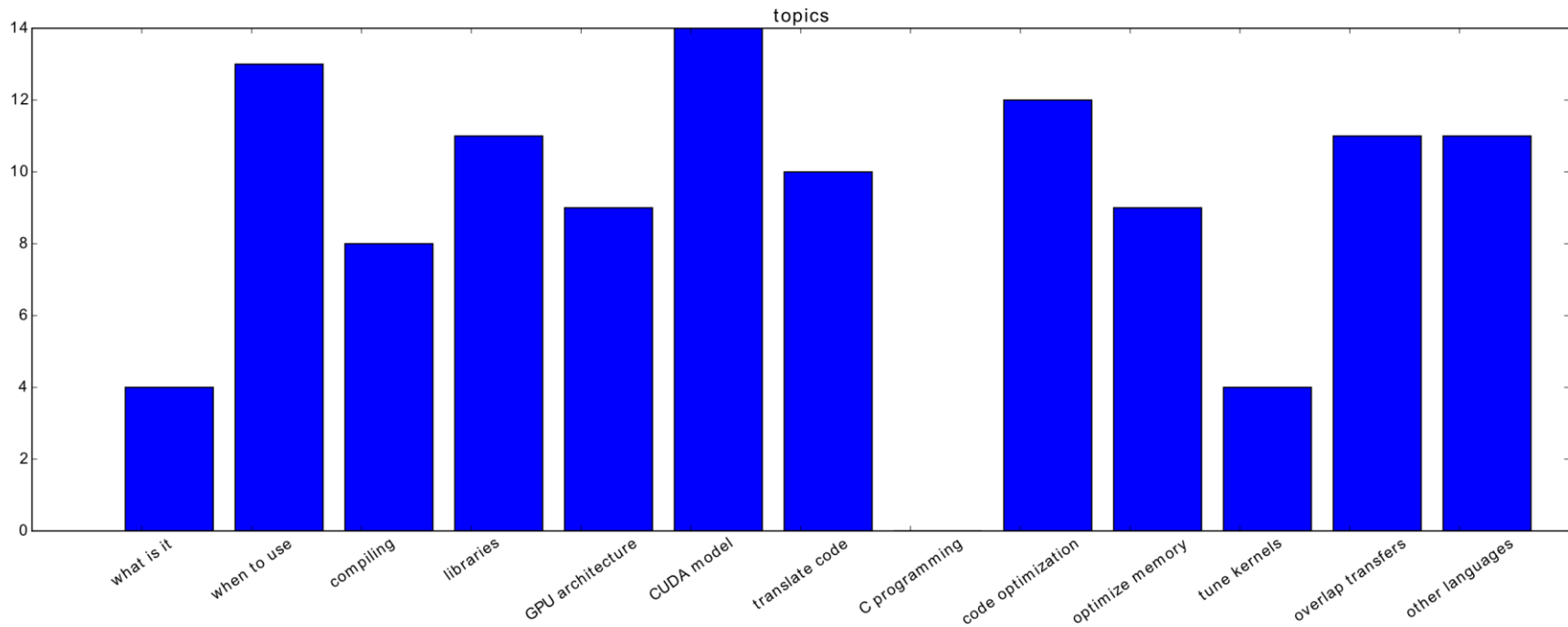
Ben van Werkhoven

netherlands

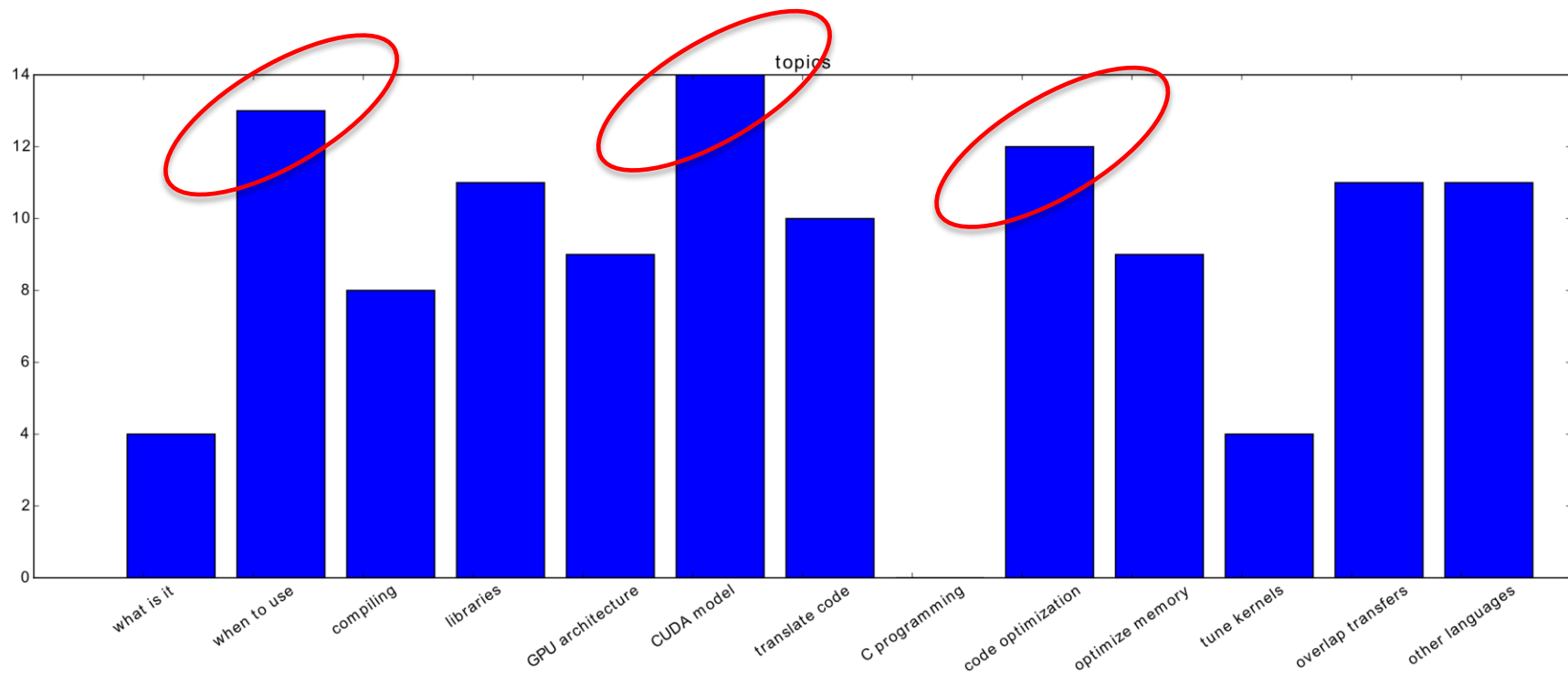
eScience center

by SURF & NWO

# Topics for today



# Topics for today



# Schedule

- **13:00 – 13:10 Getting Started and Intro to GPU Computing**
- **13:10 – 13:30 High-level intro to CUDA Programming Model**
- **13:30 – 13:45 1<sup>st</sup> Hands-on Session**
- **13:45 – 14:15 CUDA Programming model Part 2**
- **14:15 – 15:15 2<sup>nd</sup> Hands-on Session and coffee break**
- **15:15 – 15:30 CUDA Program execution**
- **15:30 – 16:00 Loop optimizations**



# Download the slides!

- **Get your own copy of the slides so you can read along and click on links**
- **My slides are sometimes very wordy, this is intentional, so they may serve as a reference that you can read again later**
- **In code samples in the slides I sometimes leave out '{' and '}' to save space**

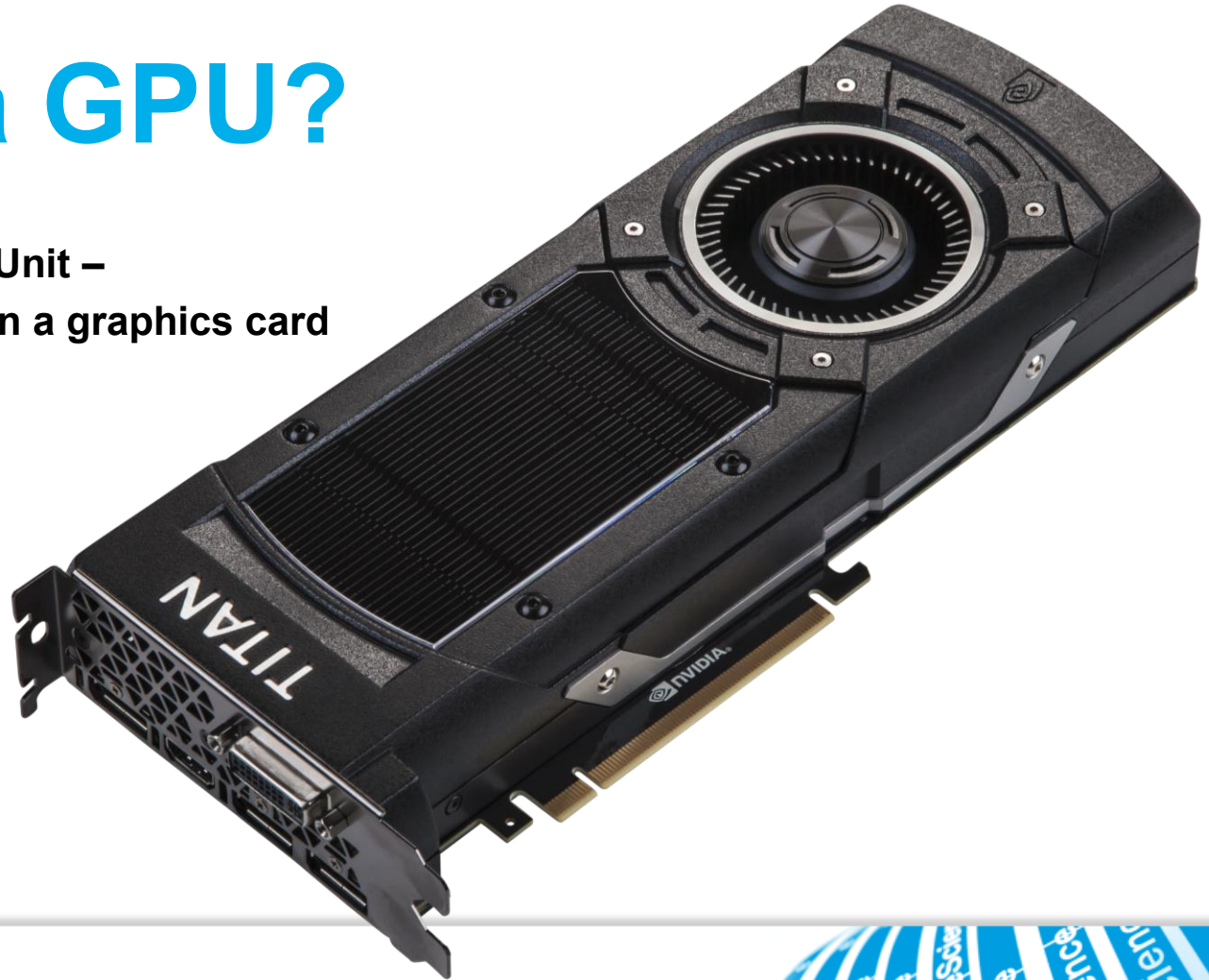


# Introduction to GPU Computing



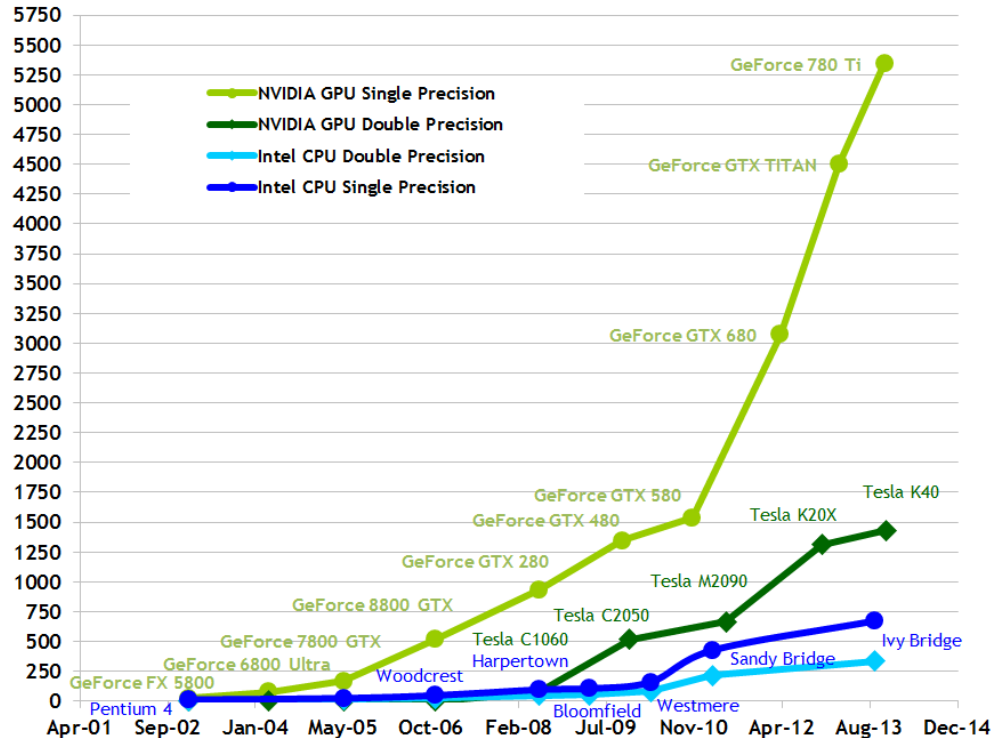
# What is a GPU?

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



# Compute performance

Theoretical GFLOP/s

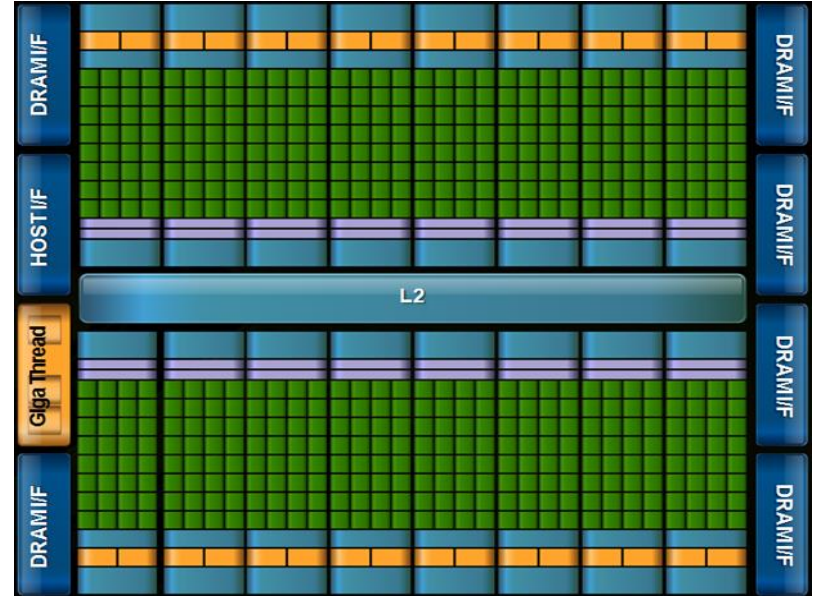
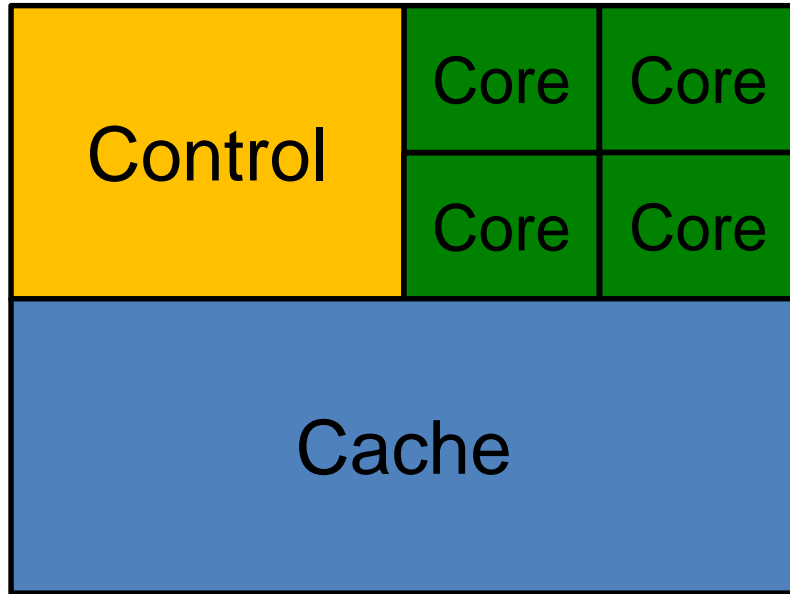


(According to Nvidia)





# CPU vs GPU Hardware



# GPU Computing

- **When:**
  - Thousands or even millions of elements that can be processed in parallel
- **Very efficient for algorithms that:**
  - have high arithmetic intensity (lots of computations per element)
  - have regular data access patterns
  - do not have a lot of data dependencies between elements
  - do the same set of instructions for all elements



# A high-level intro to the CUDA Programming Model



# CUDA Programming Model

Before we start:

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



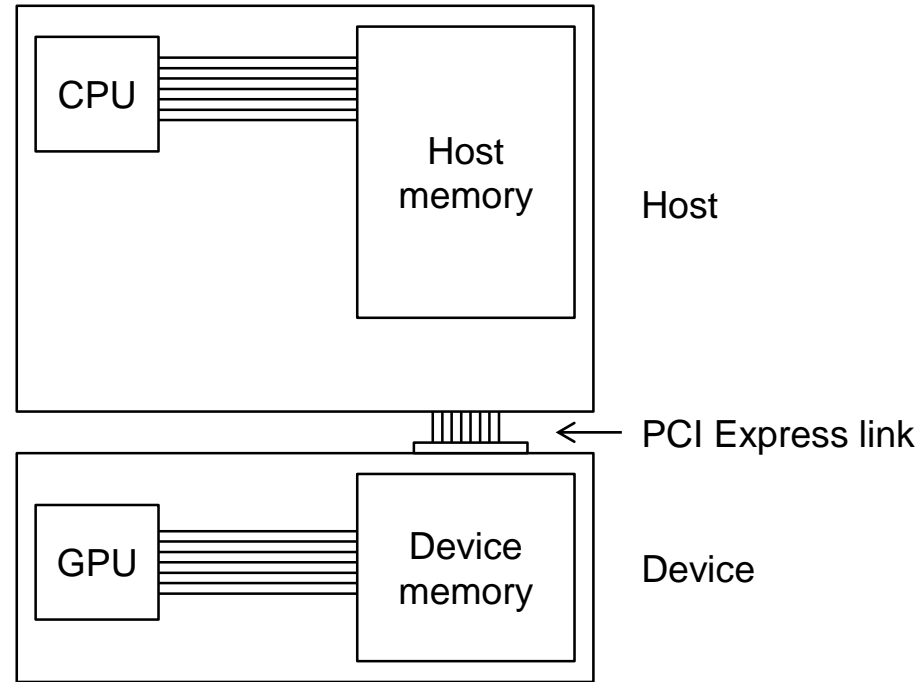
# CUDA Programming Model

- The CUDA programming model separates a program into a host (CPU) and a device (GPU) part.
- The host part: allocates memory and transfers data between host and device memory, and starts GPU functions
- The device part consists of functions that will execute on the GPU, which are called *kernels*
- Kernels are executed by huge amounts of threads at the same time
- The data-parallel workload is divided among these threads
- The CUDA programming model allows you to code for each thread individually



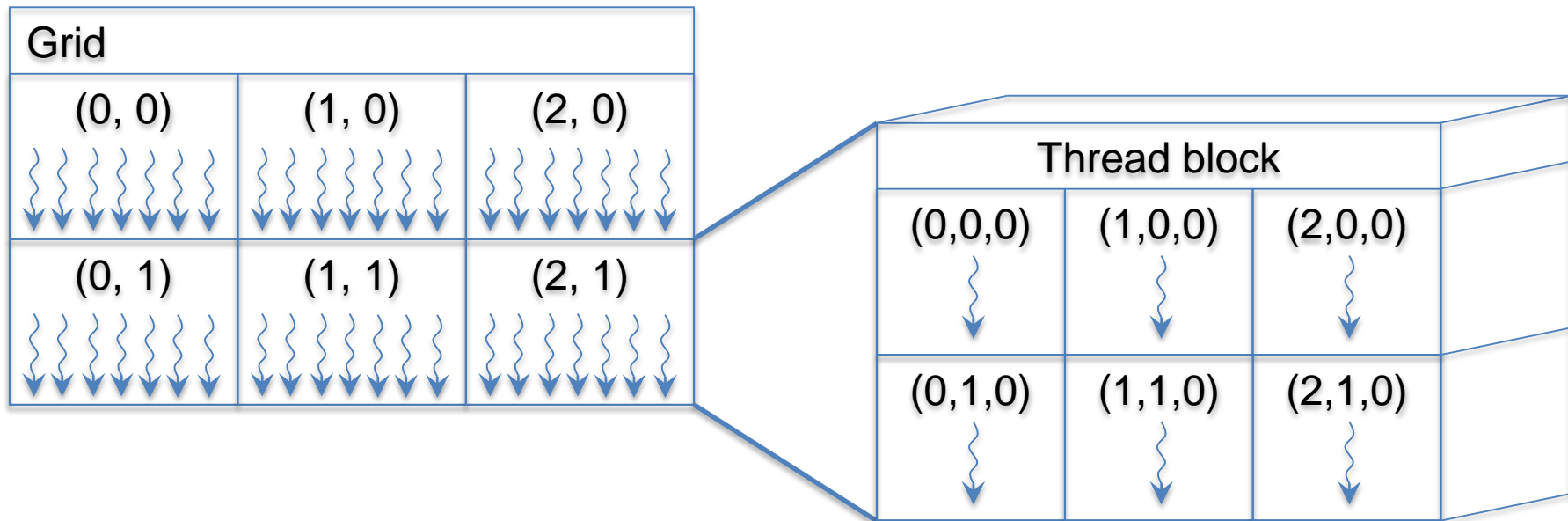
# Data management

- The GPU is located on a separate device
- The host program manages the allocation and freeing of GPU memory
  - In CUDA:
  - `cudaMalloc()`
  - `cudaFree()`
- Host program also copies data between different physical memories
  - In CUDA:
  - `cudaMemcpy()`



# Thread Hierarchy

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





# Threads

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

- **Example:**

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

**Create a single thread block of N threads:**

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

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



# Thread blocks

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



# Starting a kernel

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

- `//create variables to hold grid and thread block dimensions`

```
dim3 threads(x, y, z)
```

```
dim3 grid(x, y)
```

```
//launch the kernel
```

```
vector_add<<<grid, threads>>>(c, a, b);
```

```
//wait for the kernel to complete
```

```
cudaDeviceSynchronize();
```



# First hands-on session

- **Login on the DAS4 or DAS5**
  - **On DAS-4:** `alias gpurun="prun -np 1 -native '-l gpu=GTX480'"`
  - **On DAS-5:** `alias gpurun="srun -N 1 -C TitanX --gres=gpu:1"`
- **Clone the github repository: <https://github.com/benvanwerkhoven/gpu-course>**
- **Change to directory `vector_add`**
- **Compile by typing `make`, run by typing `gpurun vector_add`**
- **Make sure you understand everything in the code**
- **Complete the exercise!**
- **Hints:**
  - **Look at how the kernel is launched in the host program**
  - `threadIdx.x` **is the thread index within the thread block**
  - `blockIdx.x` **is the block index within the grid**
  - `blockDim.x` **is the dimension of the thread block**



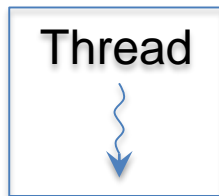
# CUDA Programming model

## Part 2

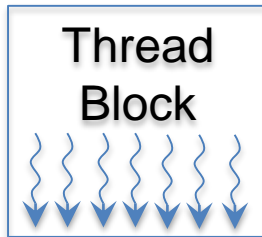


# CUDA memory hierarchy

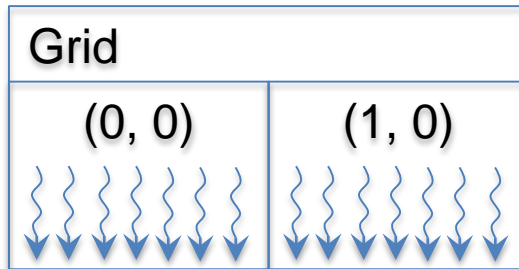
Registers



Shared memory



Global memory  
Constant memory



# CUDA memory spaces

- **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
- **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



# CUDA memory spaces

- **Shared memory**
  - Variables have to be declared using `__shared__` prefix, for example:  
`__shared__ float my_shared_float_array[num_floats];`
  - Not initialized, threads have to fill shared memory with meaningful values
  - Not persistent, after the kernel has finished, value in shared memory are lost
  - Not coherent, `__syncthreads()` is required to make writes visible to other threads within the thread block
- **Constant memory**
  - Statically defined by the host program using `__constant__` prefix, for example:  
`__constant__ float my_constant_float_array[fixed_size];`
  - 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`





# 1D indexing of 2D arrays

- This is crucial to understand the rest of this workshop!
- Idea is to mimic a 2D data structure on what is actually a 1D array in memory
- Row-major accessing all elements in an array:

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

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



# Example application

- 2D Convolution is an image processing operation used for many things

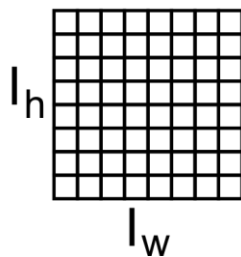
```
//for each pixel in the output image  
for (y=0; y < image_height; y++) {  
  for (x=0; x < image_width; x++) {
```

```
    //for each filter weight
```

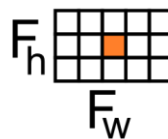
```
    for (i=0; i < filter_height; i++) {  
      for (j=0; j < filter_width; j++) {  
        output[y][x] += input[y+i][x+j] * filter[i][j];
```

```
      }  
    }  
  }  
}
```

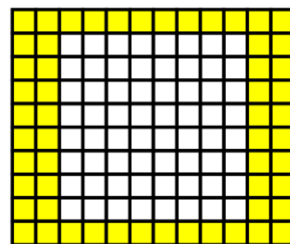
output image



filter



input image



# Example application

## CPU Implementation:

```
//for each pixel in output image
for (y=0; y < image_height; y++)
for (x=0; x < image_width; x++)

//for each filter weight
for (i=0; i < filter_height; i++)
for (j=0; j < filter_width; j++)
output[y][x] += input[y+i][x+j] *
                filter[i][j];
```

## CUDA kernel:

```
//for each output pixel, create a thread
y = blockIdx.y * blockDim.y + threadIdx.y;
x = blockIdx.x * blockDim.x + threadIdx.x;

//for each filter weight
for (i=0; i < filter_height; i++)
for (j=0; j < filter_width; j++)
output[y][x] += input[y+i][x+j] *
                filter[i][j];
```



# CUDA kernel

```
//for each output pixel, create a thread
float y = blockIdx.y * blockDim.y + threadIdx.y;
float x = blockIdx.x * blockDim.x + threadIdx.x;
float sum = 0.0f;    //thread-local register

//for each filter weight
for (i=0; i < filter_height; i++) {
    for (j=0; j < filter_width; j++) {
        sum += input[y+i][x+j] * filter[i][j];
    }
}

output[y][x] = sum;    //store result to global memory
```



# CUDA kernel

```
//for each output pixel, create a thread
float y = blockIdx.y * blockDim.y + threadIdx.y;
float x = blockIdx.x * blockDim.x + threadIdx.x;
float sum = 0.0f;    //thread-local register

//for each filter weight
for (i=0; i < filter_height; i++) {
    for (j=0; j < filter_width; j++) {
        sum += input[(y+i)*input_width+x+j] * filter[i*filter_width+j];
    }
}

output[y*image_width+x] = sum;    //store result to global memory
```



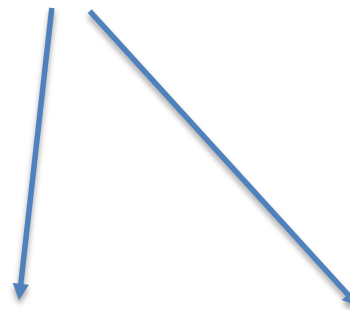
# CUDA kernel

```
//for each output pixel, create a thread
float y = blockIdx.y * blockDim.y + threadIdx.y;
float x = blockIdx.x * blockDim.x + threadIdx.x;
float sum = 0.0f;    //thread-local register
```

```
//for each filter weight
for (i=0; i < filter_height; i++) {
    for (j=0; j < filter_width; j++) {
        sum += input[(y+i)*input_width+x+j] * filter[i*filter_width+j];
    }
}
```

```
output[y*image_width+x] = sum;    //store result to global memory
```

**i and j do not depend on  
threadIdx, so all threads  
access the same values!**

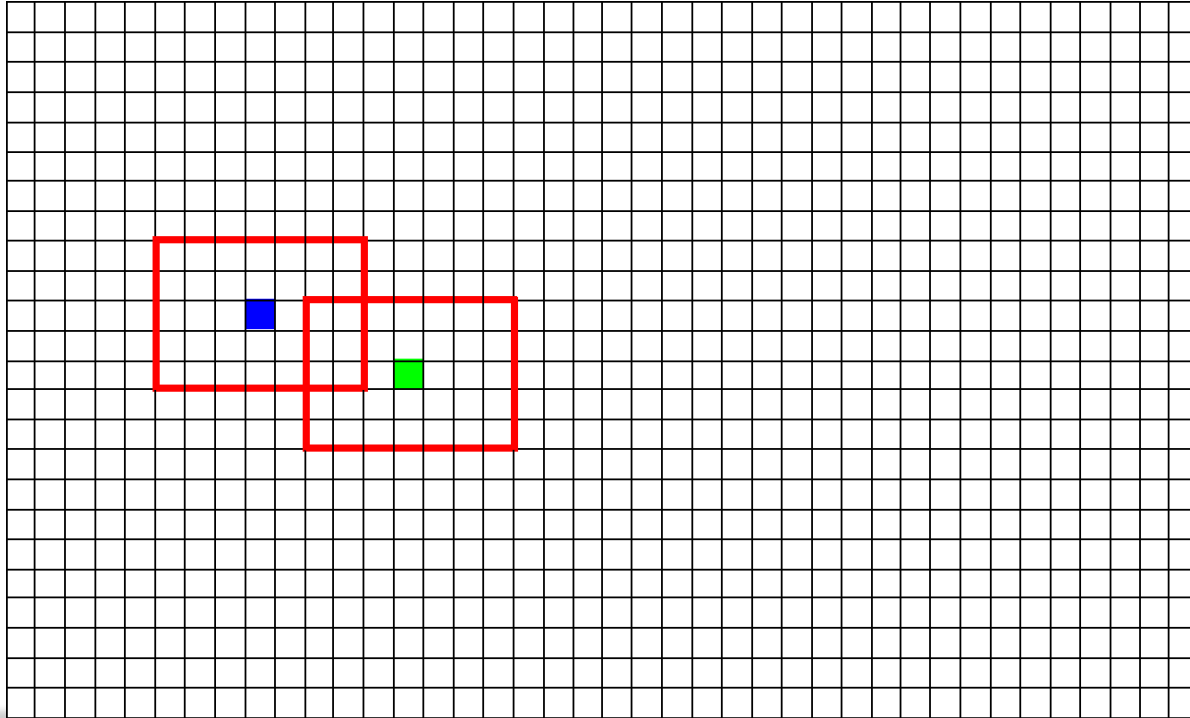


# Second hands-on session

- Go to directory convolution, look at convolution.cu
- Make sure you understand everything in the code
- **Task #1: Store the convolution filter in constant memory space:**
  - Declare a float array of size `filter_height * filter_width` as a global variable using the `__constant__` qualifier, for storing the filter in device memory
  - Change the `cudaMemcpy()` for filter to `cudaMemcpyToSymbol()` and make sure it copies to the right place
  - See [CUDA Runtime documentation on cudaMemcpyToSymbol](#)
  - Make sure the constant memory array is used inside the kernels
- **Task #2: Finalize the `convolution_kernel_shared_mem` kernel code**
  - Write the code that fills shared memory with the values needed by this thread block
  - Rewrite the innermost loop to actually use values in the shared memory
  - Hint: look at how large `sh_input` is

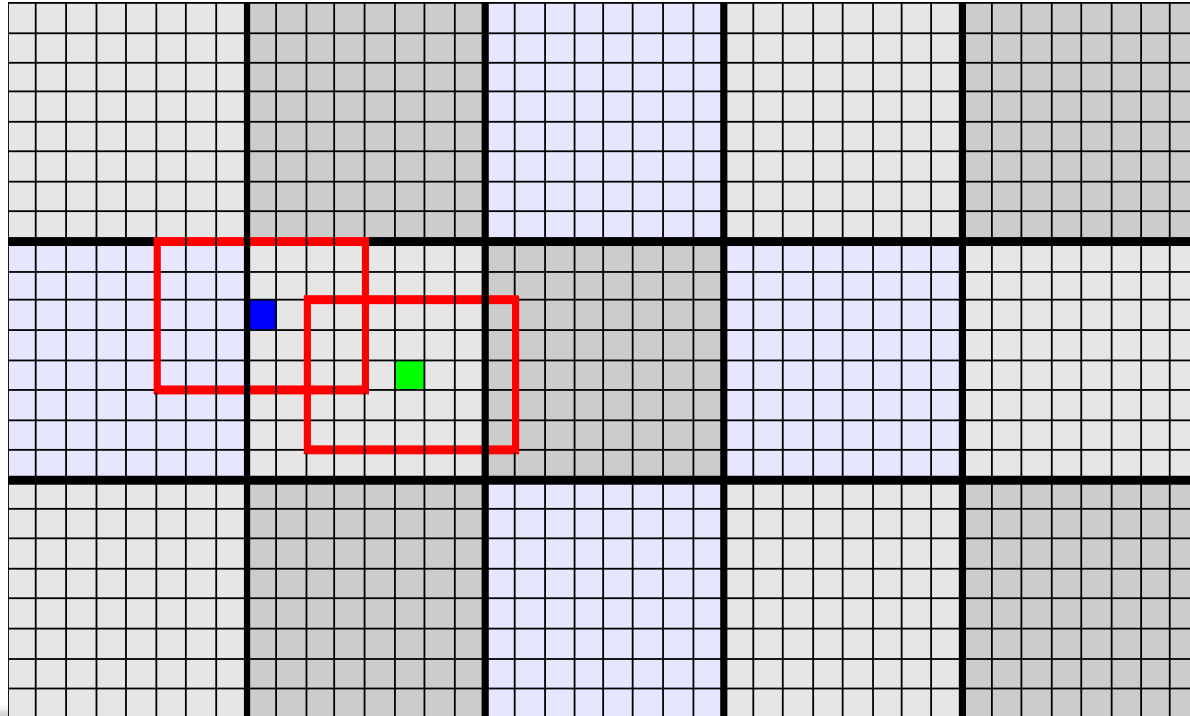


# Hint for Task #2

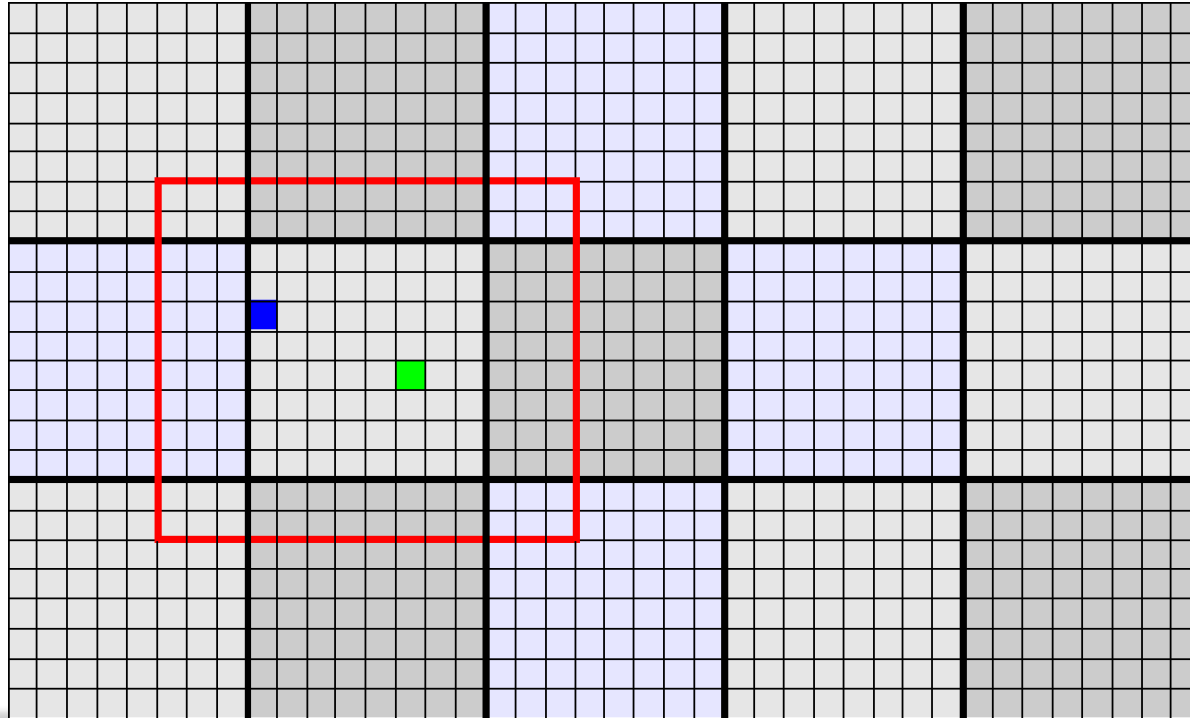




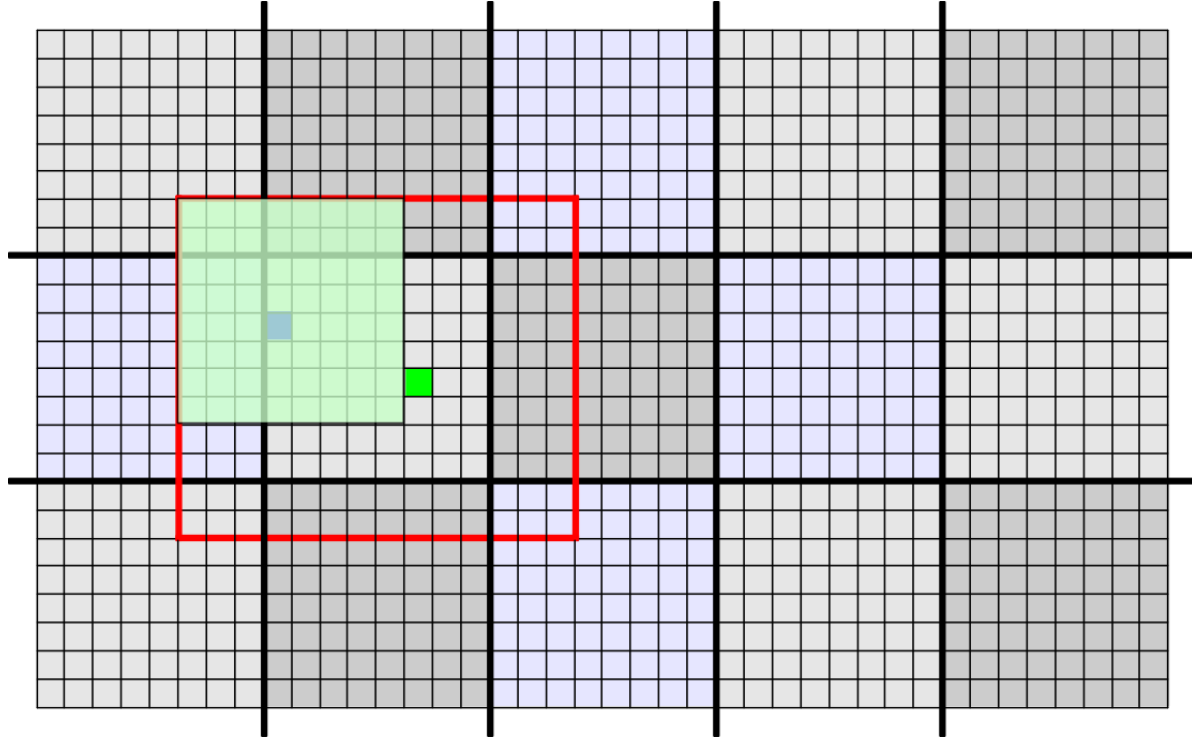
# Hint for Task #2



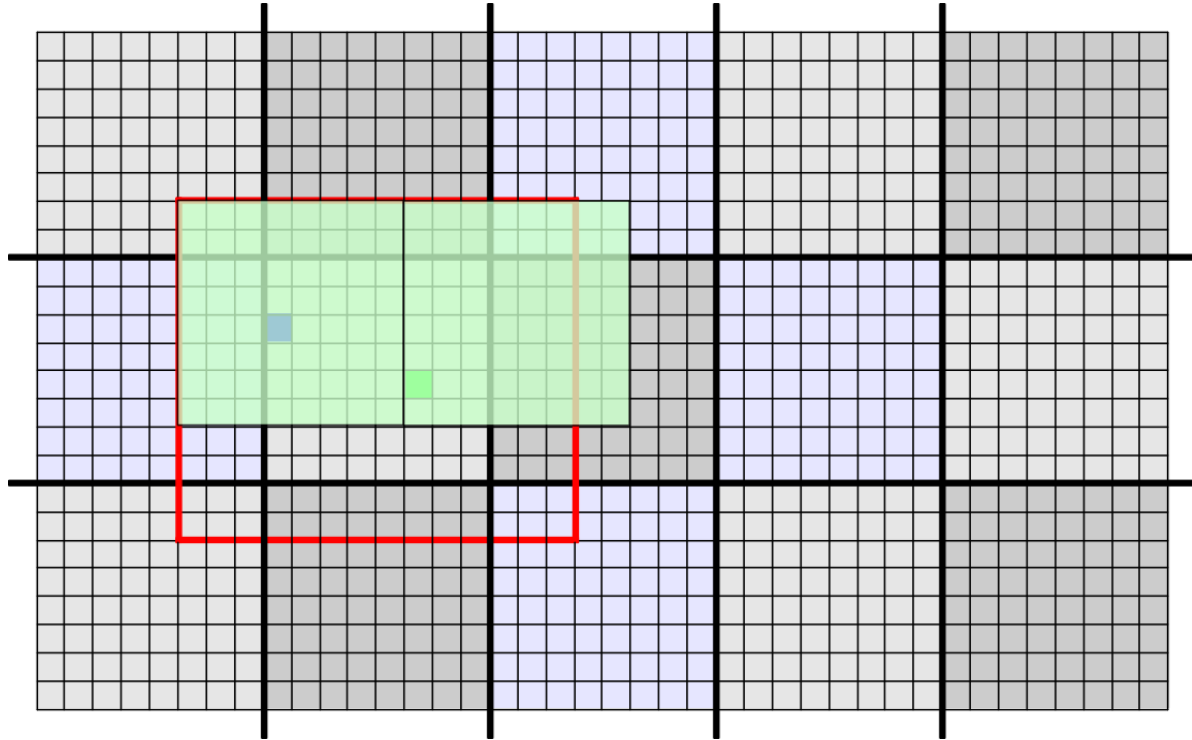
# Hint for Task #2



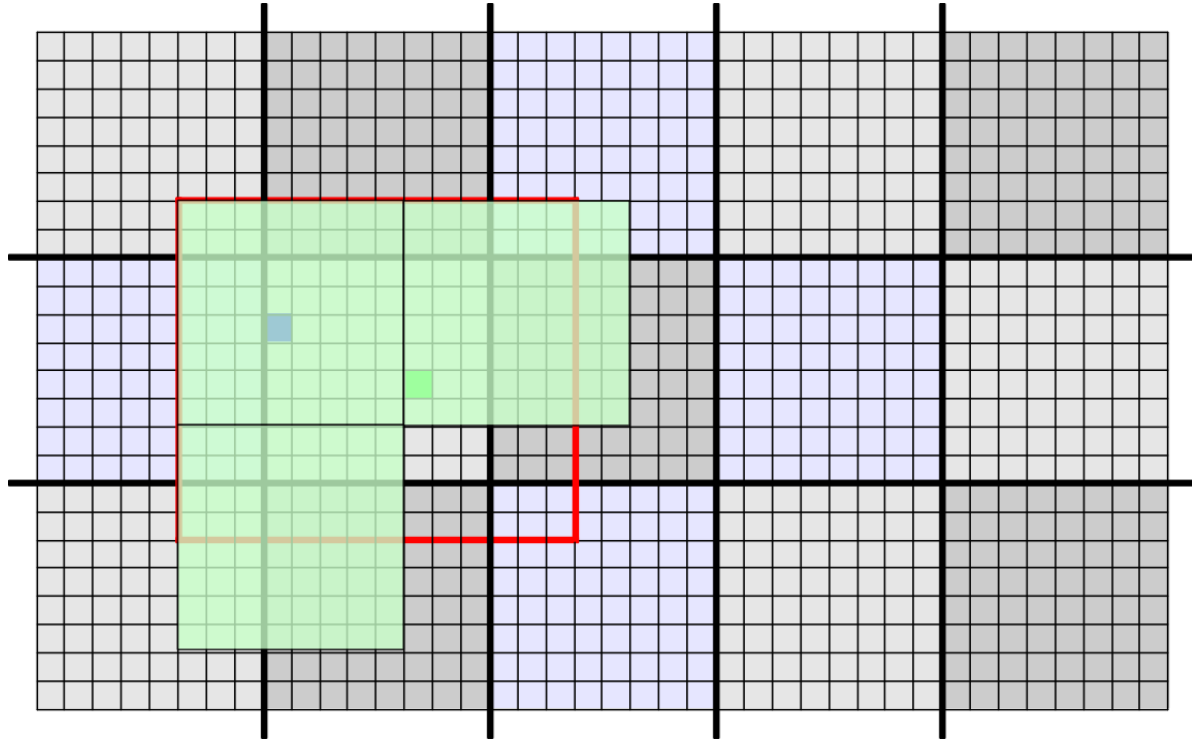
# Hint #2 for Task #2



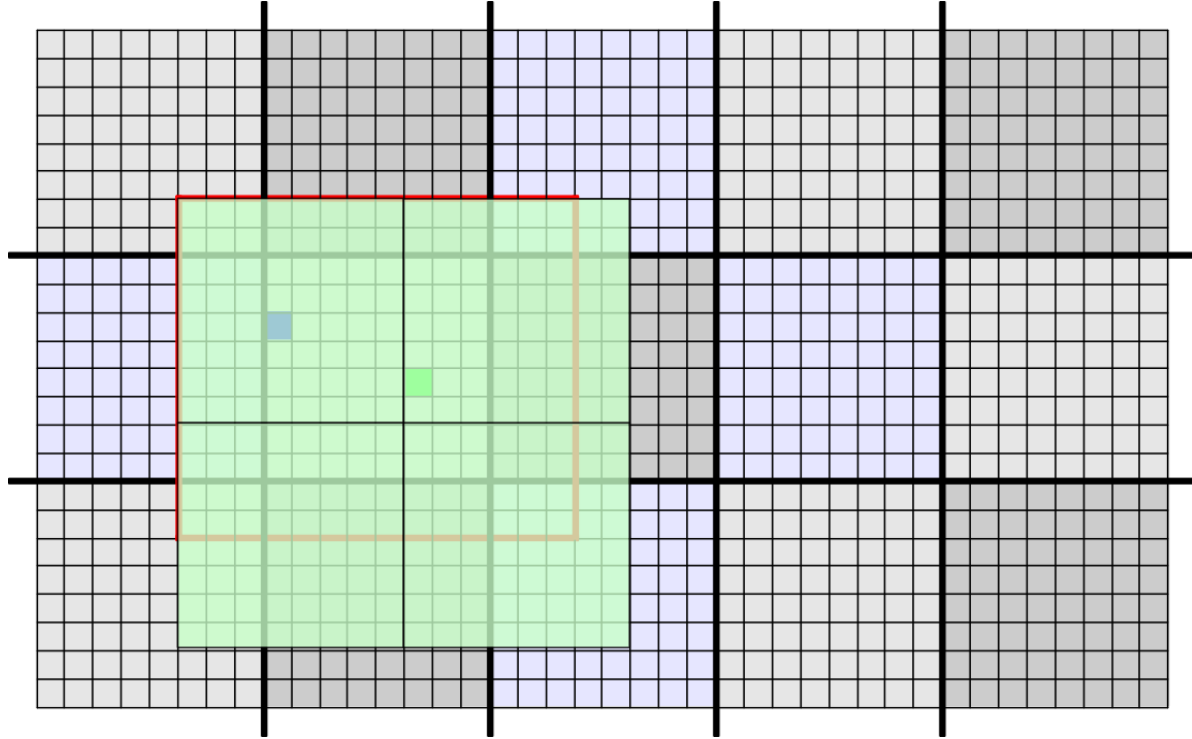
# Hint #2 for Task #2



# Hint #2 for Task #2



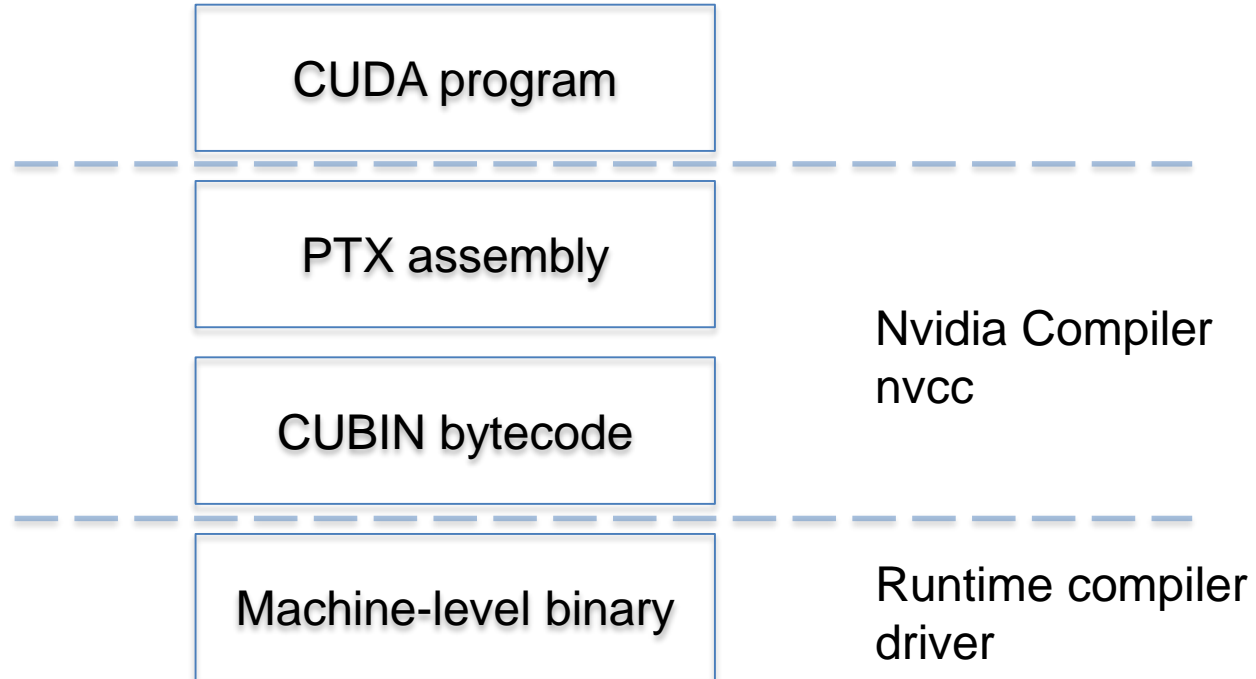
# Hint #2 for Task #2



# CUDA Program execution



# Compilation





# 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

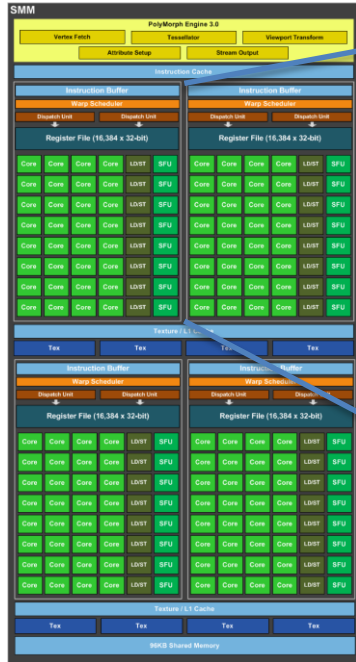


# Predication

- All threads in a warp execute the exact same *instruction* at the same cycle
- The same instruction, but on different data
- What about control flow instructions? (if, else, for, while)
- All threads in the warp execute all paths, with some threads being predicated
- This is less efficient, but not always bad
- Avoid data-dependent conditional branching where possible
- Thread index-dependent branching is harmless, in particular when you respect the warp size



# Maxwell Architecture

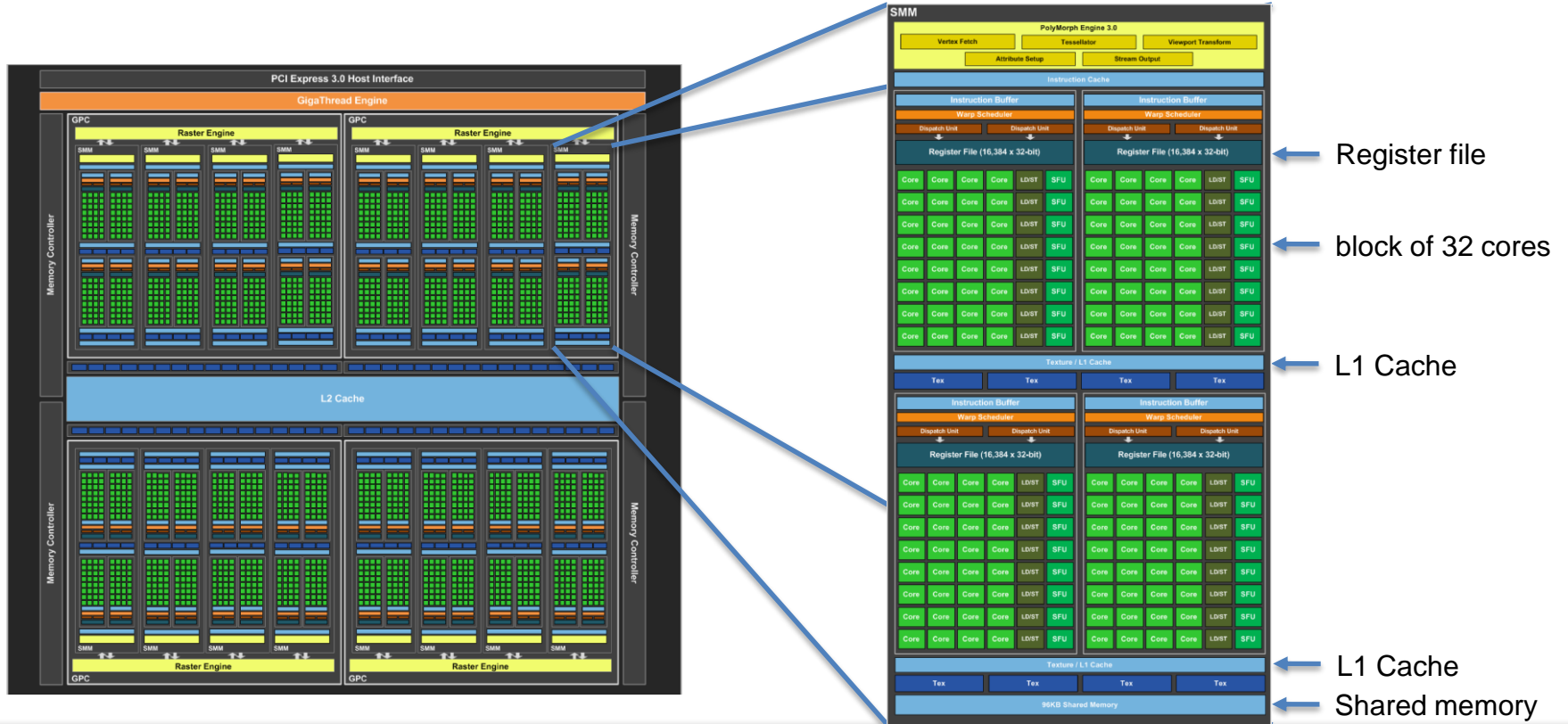


32 core block

Streaming multiprocessor (SM)

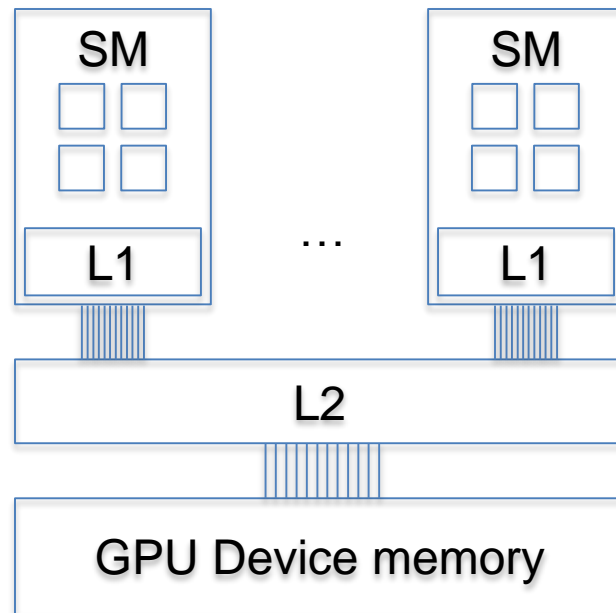


# Maxwell Architecture



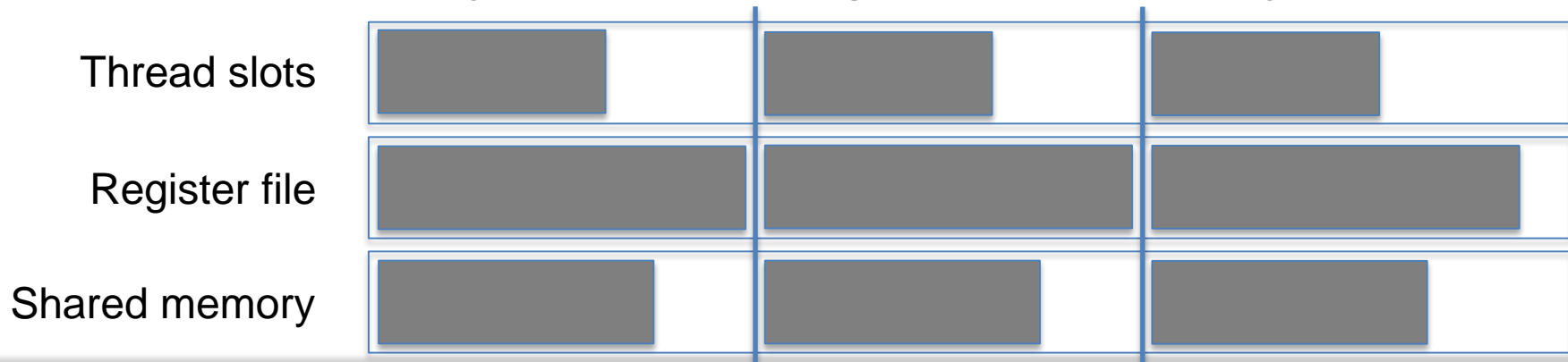
# Global Memory access

- Global memory is cached at L2, and for some GPUs also in L1
- When a thread reads a value from global memory, think about:
  - The total number of values that are access by the warp that 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

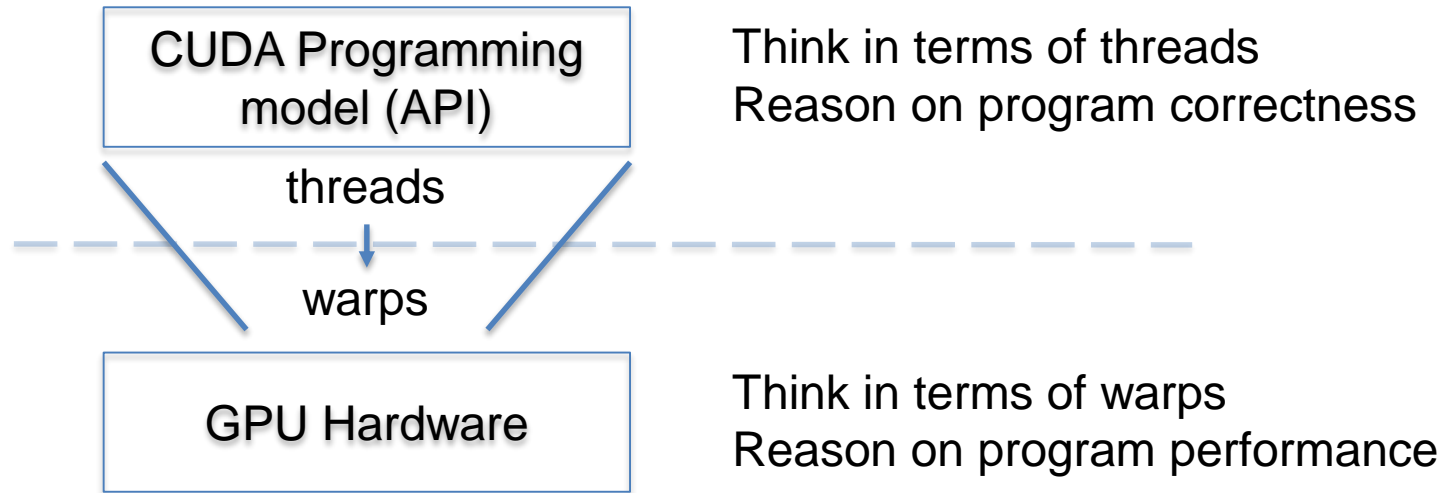


# Resource partitioning

- The GPU consists of several (1 to 16) *streaming multiprocessors* (SMs)
- The SMs are fully independent
- Each SM contains several resources: Thread and Thread Block slots, Register file, and Shared memory
- SM Resources are dynamically partitioned among the thread blocks that execute concurrently on the SM, resulting in a certain *occupancy*



# Overview



# Loop optimizations





# Simple loop manipulations

- **Loop splitting:**

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

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

- **Loop fusion:**

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

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



# Kernel fusion

- **Original code:**

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

```
vec_add(c, a, b);  
vec_add(d, c, a);  
vec_add(e, d, b);
```

- **Inline the vec\_add function and then merge the three loops into one:**

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

- **If c and d are only used here you can save a lot of memory traffic:**

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



# Loop unrolling

- **Example:**

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

- **4 way loop unrolling:**

```
for (i=0; i<N; i+=4)  
    c[i+0] = a[i+0]+b[i+0];  
    c[i+1] = a[i+1]+b[i+1];  
    c[i+2] = a[i+2]+b[i+2];  
    c[i+3] = a[i+3]+b[i+3];
```

- **Result: The number of times we have to increment i and check  $i < N$  is reduced by a factor of four, reducing instruction overhead and increasing instruction-level parallelism**



# Loop re-ordering

- **Key idea:** It is not just the total number of computations that determines the execution time of your application, the order in which the computations happen is just as important!

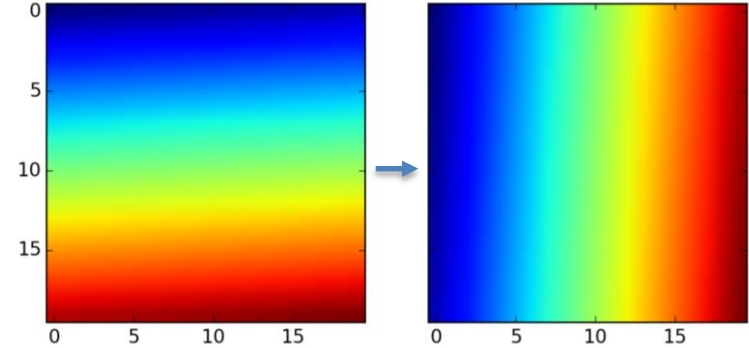
- **Example: Naive Square Matrix-Matrix Multiplication**

```
for (i=0; i<N; i++){  
    for (j=0; j<N; j++){  
        for (k=0; k<N; k++){  
            C[i][j] += A[i][k] * B[k][j];  
        }  
    }  
}
```



# Loop re-ordering

```
for (i=0; i<N; i++)  
  for (j=0; j<N; j++)  
    for (k=0; k<N; k++)  
      C[i][j] += A[i][k] * B[k][j];  
  
for (j=0; j<N; j++)  
  for (i=0; i<N; i++)  
    for (k=0; k<N; k++)  
      C[i][j] += A[i][k] * B[k][j];
```



**Let's call the first order 'ijk', then we can also define orderings: ikj, jki, jik, kij, kji**

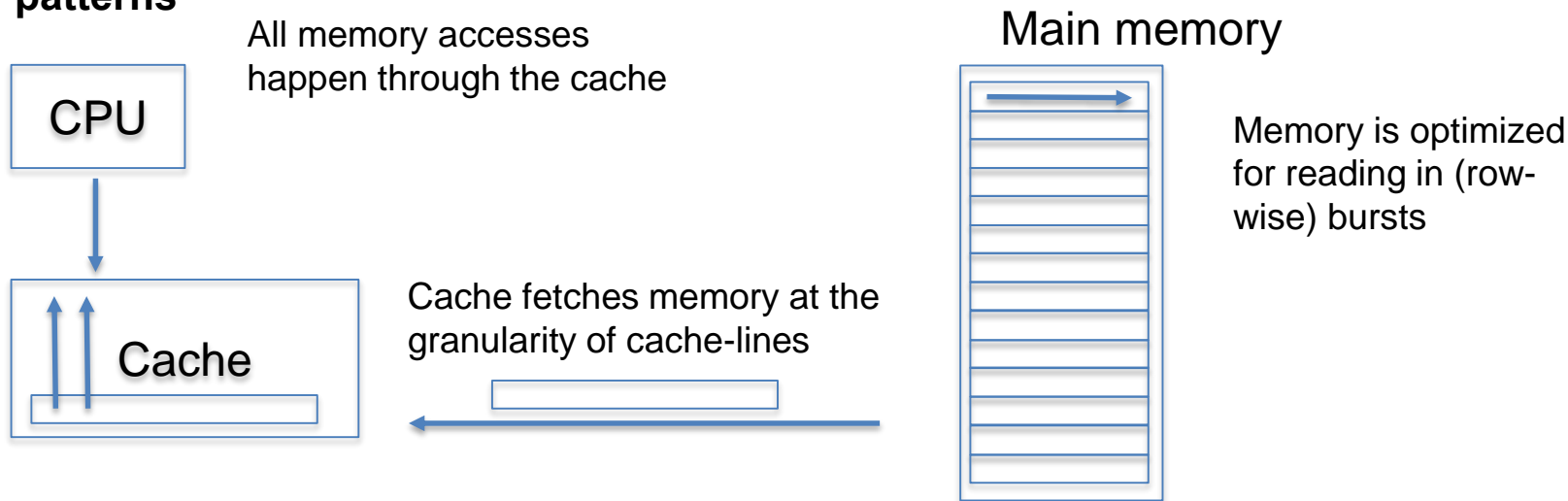
# Loop order matters

- Testing loop orders ijk, ikj, jki, jik, kij, kji on a 1000x1000 matrix on my laptop:
  - ijk loop order took 325.584381 ms
  - ikj loop order took 284.181885 ms
  - jik loop order took 1000.532104 ms
  - jki loop order took 11856.112305 ms
  - kij loop order took 286.160706 ms
  - kji loop order took 3078.023926 ms
- For each loop order the total number of computations is the same
- But the execution times are quite different (up to factor ~40 in this example)
- Try this out yourself! See [loop\\_order.cc](#)



# Why does loop order matter?

- The answer is in that our 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**

# Loop tiling

- **Key idea: Keep the same number of loop iterations, but loop through them in 'blocks' or 'tiles', also referred to as cache-blocking**
- **The goal is to limit the working set of the algorithm to fit inside the cache**
- **For a single loop:**

```
for (i=0; i<N; i++)  
    some_array[i]
```

- **Change to:**

```
//BS is the block size  
for (i=0; i<N/BS; i++)          //loop over the number of blocks  
    for (ib=0; ib<BS; ib++)    //loop within a block  
        some_array[i*BS+ib]    //i block index, ib index within i
```





# Loop tiling in 2D

- **A 2D loop:**

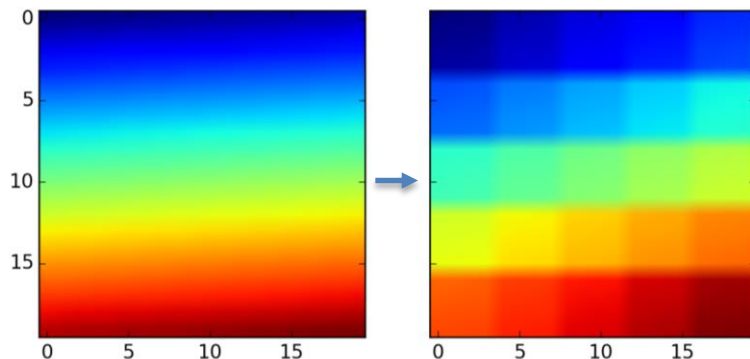
```
for (i=0; i<N; i++)  
    for (j=0; j<N; j++)  
        some_array[i*N+j]
```

- **Changes to:**

```
for (i=0; i<N/BS; i++)  
    for (j=0; j<N/BS; j++)  
        for (ib=0; ib<BS; ib++)  
            for (jb=0; jb<BS; jb++)  
                some_array[(i*B+ib)*N+j*B+jb]
```

//iterate over the blocks

//iterate within a block



# Loop tiled 2D Convolution

```
//for each tile of pixels
for (int y=0; y < image_height/tile_size; y++) {
  for (int x=0; x < image_width/tile_size; x++) {

    //for each pixel in a tile
    for (int yb=0; yb < tile_size; yb++) {
      for (int xb=0; xb < tile_size; xb++) {

        //for each filter weight
        for (int i=0; i < filter_height; i++) {
          for (int j=0; j < filter_width; j++) {
            output[y+yb][x+xb] += input[y+yb+i][x+xb+j] * filter[i][j];
```



# Optimizing Code

- **Auto-tune (automatically exploring the parameter space)**
  - Different loop orderings
  - Different tile sizes, on multiple levels L3, L2, and L1
  - Different number of threads, vector lengths, etc
- **Start with a simple algorithm and keep it for readability and correctness checks**
- **Only optimize when needed**
- **Focus on the bottlenecks first**

