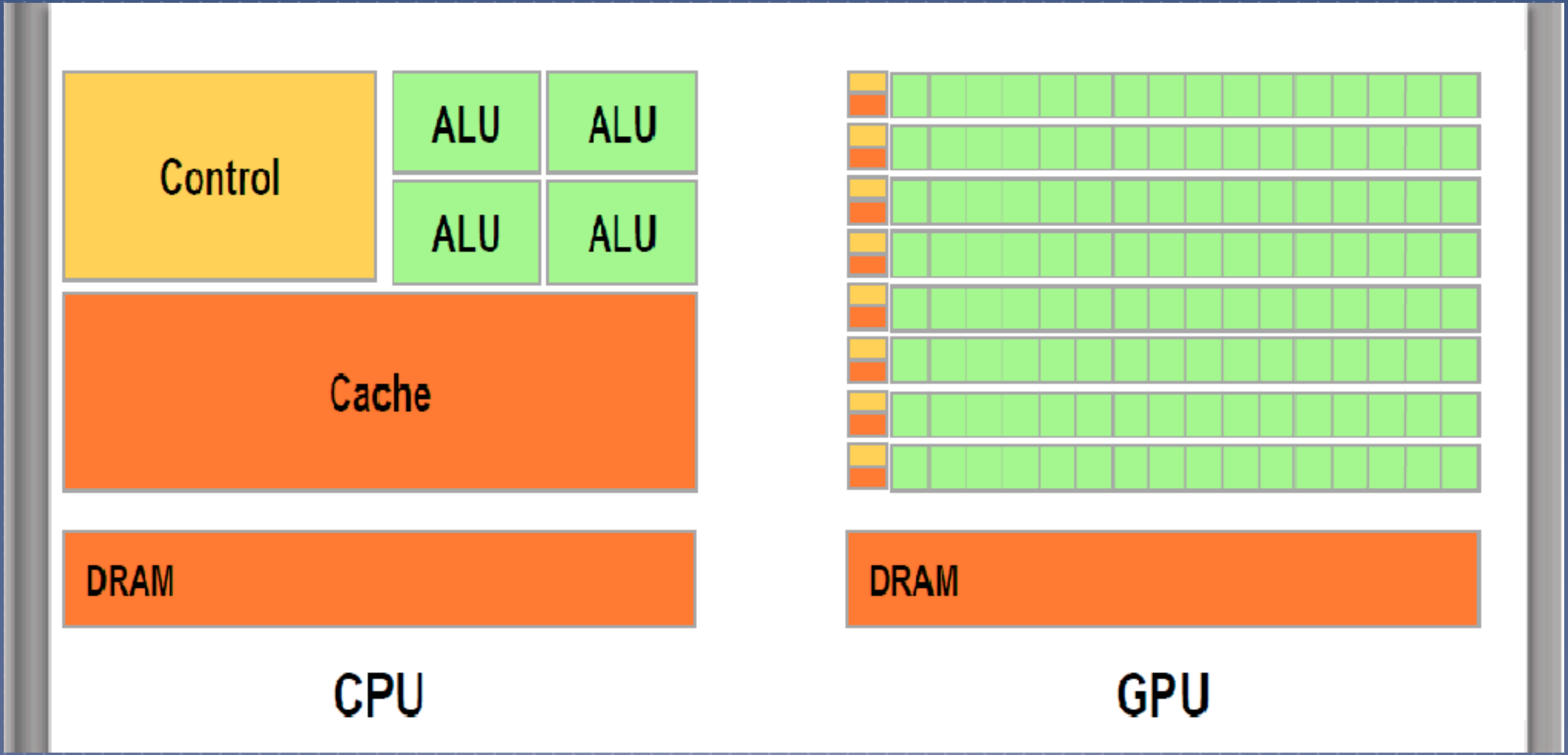
Scientific Computing on Graphics Processing Units

Nicholas Frontiere
ANL/University of Chicago
ATPESC

Overview

- GPU vs. CPU
- CUDA vs. OpenCL (Briefly)
- OpenCL execution and memory framework
- GPU Hardware
- GPU Coding Obstacles and Solutions
  - Lock in Step execution (divergent if's)
  - Memory Latency
  - Coalesced Memory
  - Bank Conflicts
- N-Body Example
- Conclusion

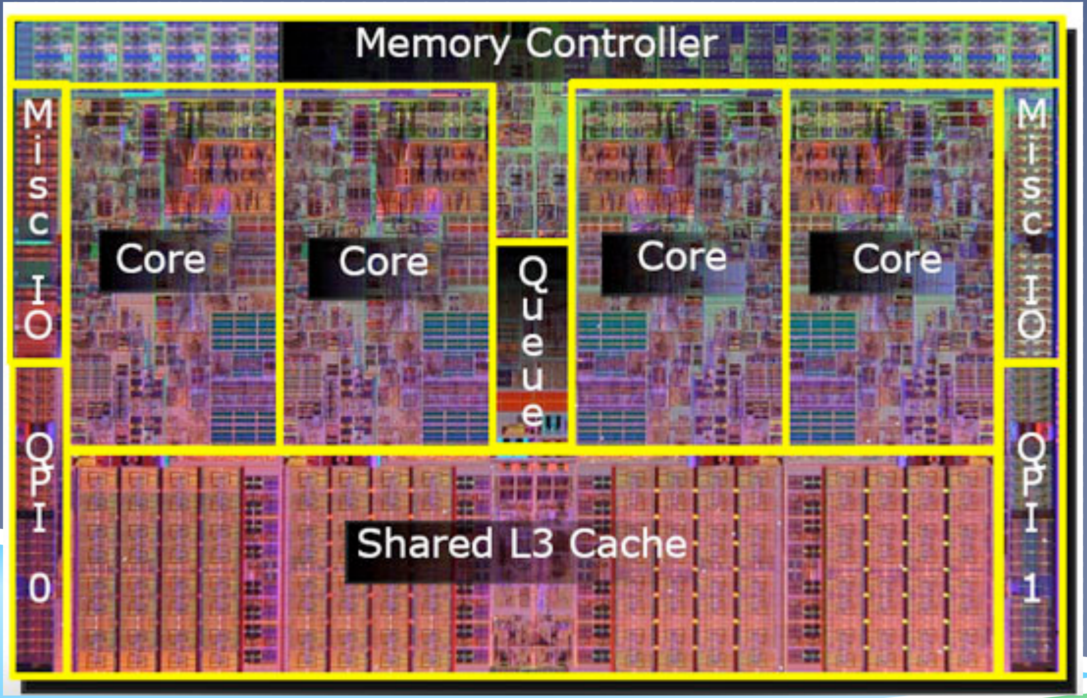
GPU vs. CPU



*D. Kirk & W. Hwu 2010

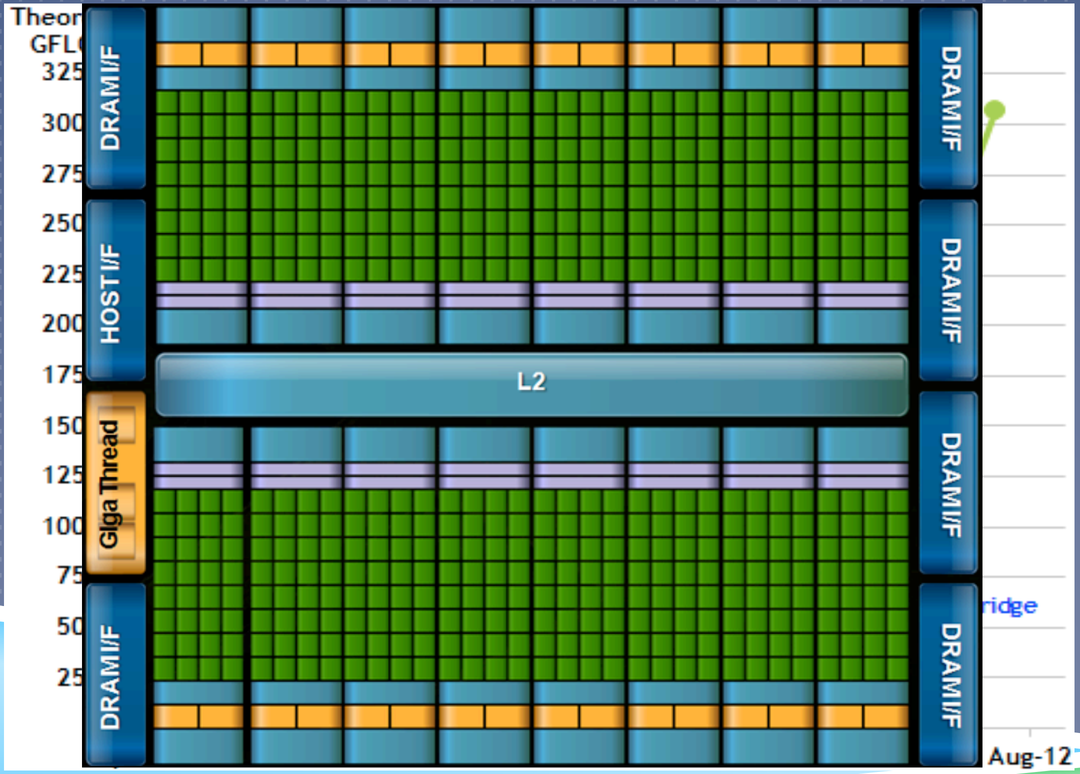
CPU

- Follows the "multicore" design of a microprocessor
  - Attempt to increase the speed of sequential programs.
  - Example Intel i7 processor.
- Optimized to handle out of order execution
- Retains multilevel cache for quick memory access
- Implements sophisticated branch prediction
- Multiple cores allow for increased multi-tasking as well as threading



GPU

- Follows the "many-core" design of a microprocessor
  - Maximize throughput of parallel algorithms.
- Typically the number of cores doubles with each new generation
  - Same is true for CPUs, yet GPU's have many many more cores.
- Throughput of Single Precision has increased dramatically



*http://docs.nvidia.com/

Question

Would you rather outsource to a

Grad Student Capable Contractor

High Latency
Low Throughput

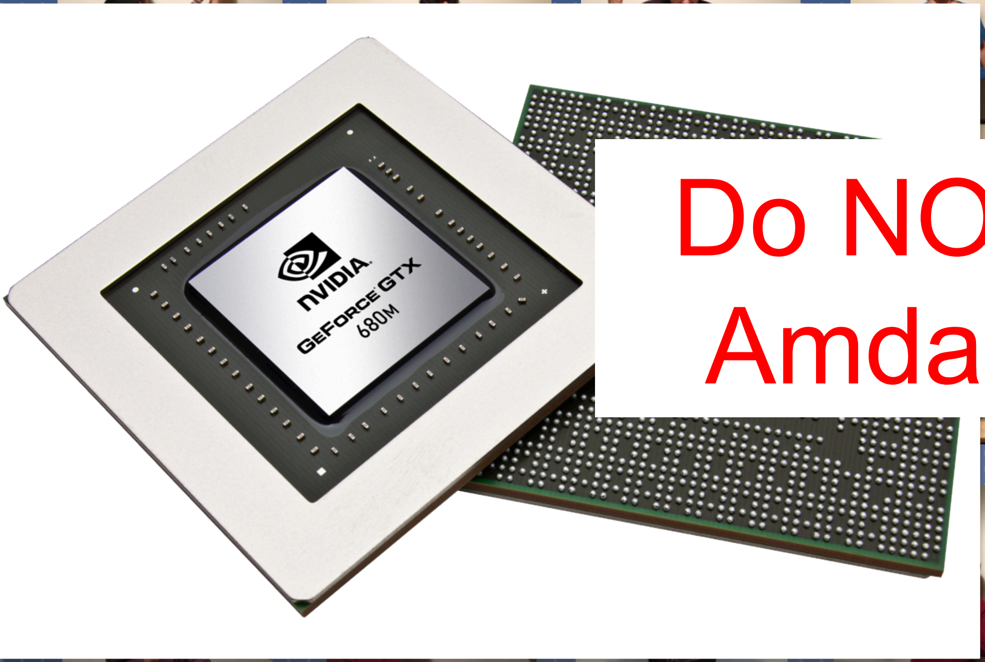
Low Latency
Good Throughput

Question

BUT what about MORE grad students?

BUT what about MORE grad students?

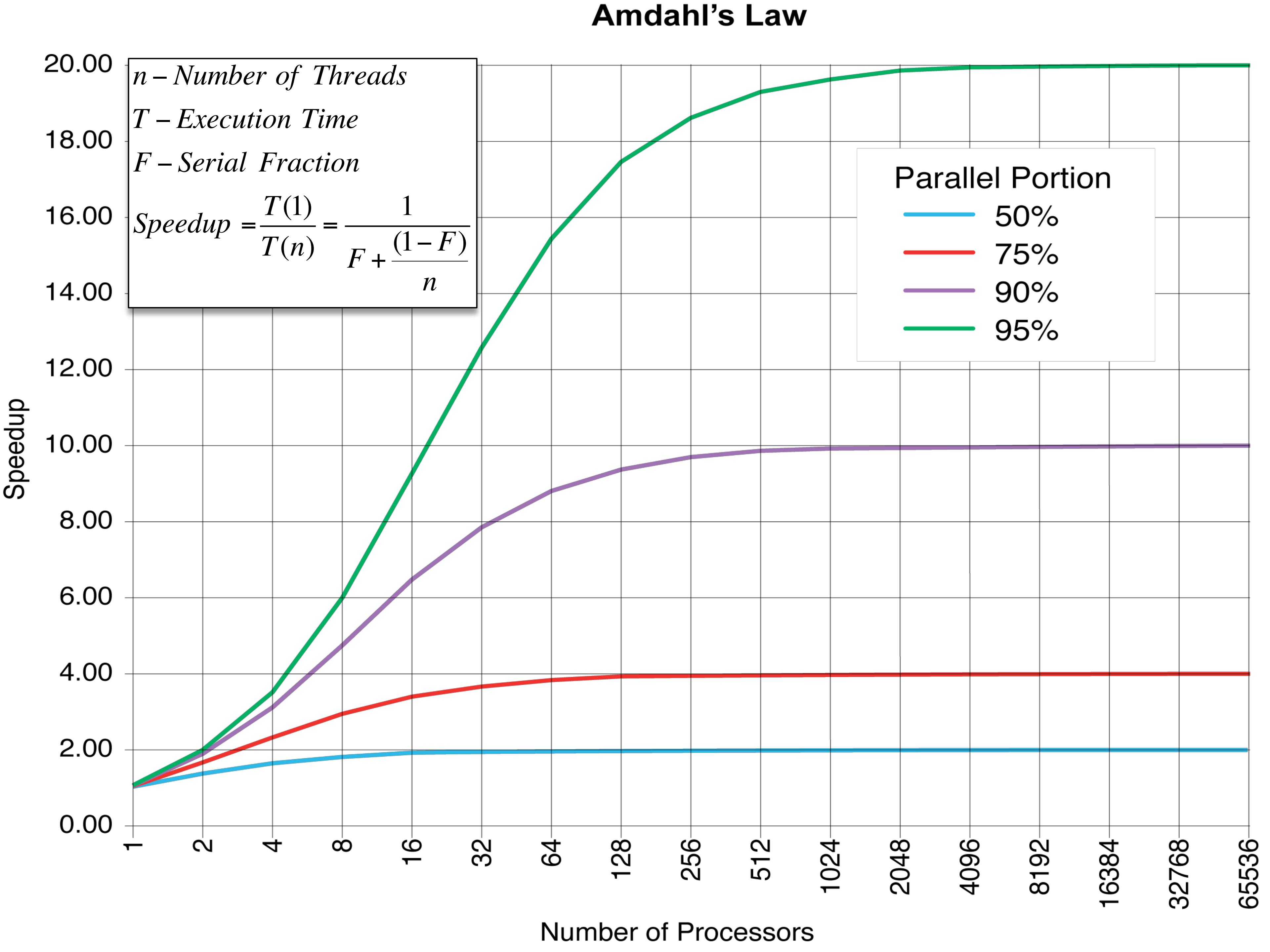
GPU vs. CPU



Do NOT forget Amdahl's law

Ok Latency
High Throughput

Low Latency
Good Throughput



Amdahl's Law
n – Number of Threads
T – Execution Time
F – Serial Fraction
$Speedup = \frac{T(1)}{T(n)} = \frac{1}{F + \frac{(1-F)}{n}}$
Parallel Portion
50%
75%
90%
95%
Speedup
Number of Processors
0.00
2.00
4.00
6.00
8.00
10.00
12.00
14.00
16.00
18.00
20.00
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536

Take Away

Completely limited by the Serial Fraction!

- Examples of GPU accelerated code:
  - Matrix multiplication, Graphics, Tabular applications, Visual Reduction, etc.

OpenCL vs. Cuda

- Both languages capable of executing GPU kernels.
- CUDA is vendor dependent (Nvidia GPUs)
- OpenCL can run on many different heterogeneous platforms (CPU, GPU, DSP, etc)
- CUDA is more mature and as a result has highly optimized libraries
- OpenCL would be considered a "lower level" language and thus harder to code.
- Which to choose?
  - Depends on what you want to do, what platforms you want to use, and the targeted users.

OpenCL Platform Model

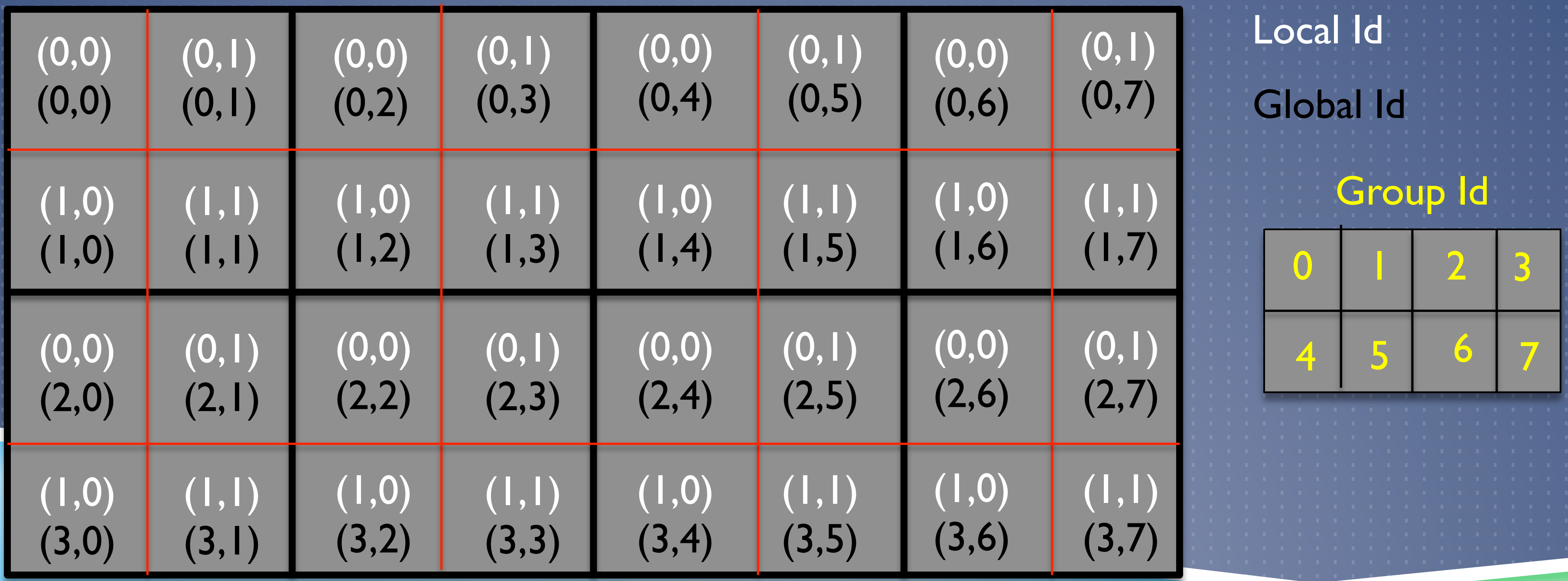
- Host code (CPU)
  - Device Queries and Platform Setups (allows one to use multiple devices)
  - Push/pull memory to/from device (GPU)
  - Compile and Launch Kernel(s)
  - Typically performs the branched logic of the application
- Kernel Code (GPU)

```
__kernel void addTwoArrays(__global float * arr, __global float* arr2,
                           __global float * return)

{
  return[get_global_id(0)]=arr[get_global_id(0)]
                          +arr2[get_global_id(0)];
}
```

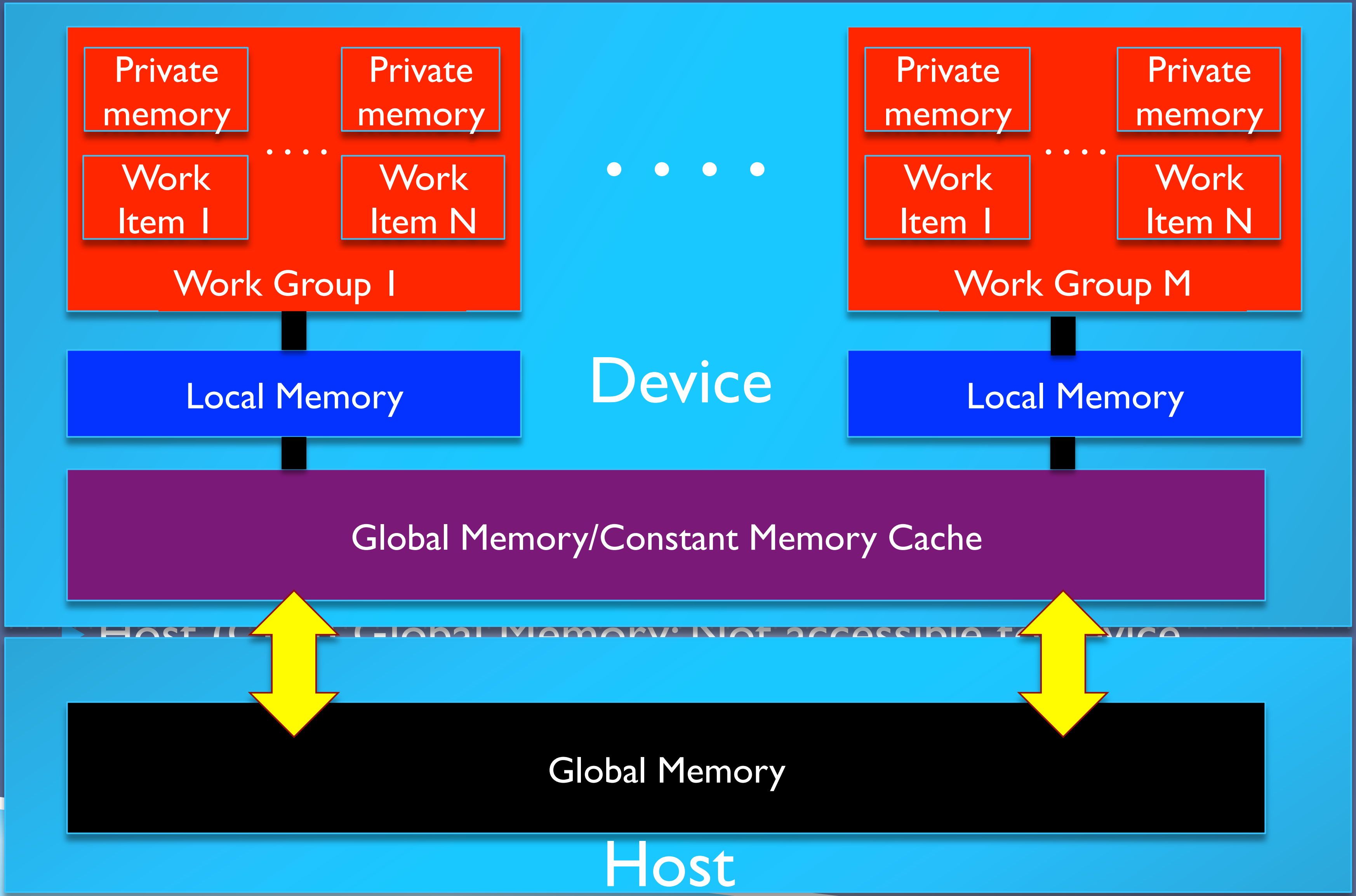
```
// Create an OpenCL context on first available platform
context = CreateContext();

// Create a command-queue on the first device available on the created context
commandQueue = CreateCommandQueue(context, &device);

// Create OpenCL program from HelloWorld.cl kernel source
program = CreateProgram(context, device, "HelloWorld.cl");
// Create OpenCL kernel
kernel = clCreateKernel(program, "hello_kernel", NULL);

  // Set the kernel arguments (result, a, b)
errNum = clSetKernelArg(kernel, 0, sizeof(cl_mem), &memObjects[0]);
errNum |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &memObjects[1]);
errNum |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &memObjects[2]);
size_t globalWorkSize[1] = { ARRAY_SIZE };
size_t localWorkSize[1] = { 1 };

// Queue the kernel up for execution across the array
errNum = clEnqueueNDRangeKernel(commandQueue, kernel, 1, NULL,
                                globalWorkSize, localWorkSize,
                                0, NULL, NULL);
// Read the output buffer back to the Host
errNum = clEnqueueReadBuffer(commandQueue, memObjects[2], CL_TRUE,
                             0, ARRAY_SIZE * sizeof(float), result,
                             0, NULL, NULL);
```

OpenCL Kernel Execution

- Kernels get executed by threads or "work items." Each item is assigned a "global index (id)"
- These work items are collected as "work groups," and assigned a "group id" and "local id"
- These id's allow the kernel code to perform thread, group, or global specific tasks.

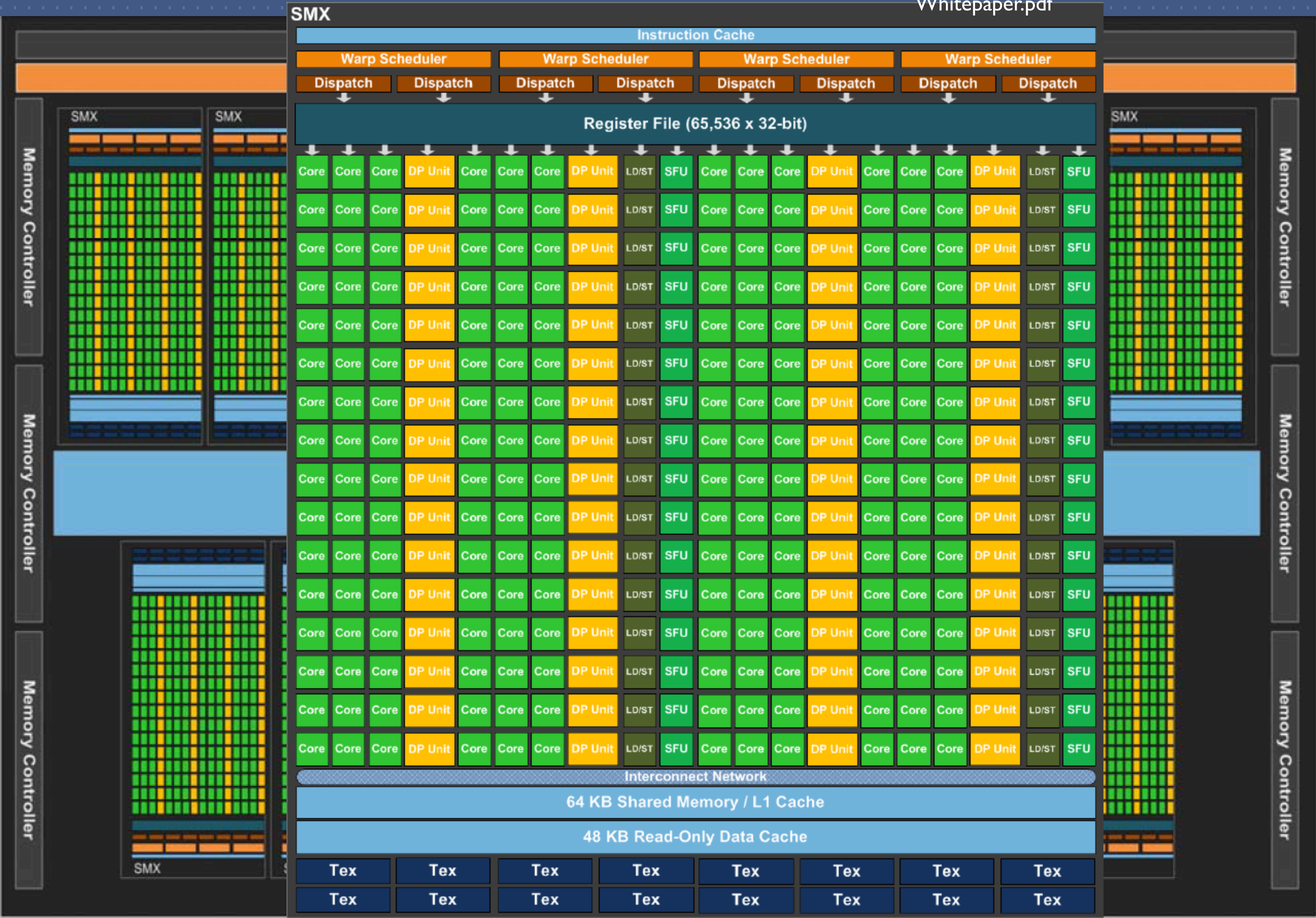


OpenCL Memory Hierarchy



GPU Hardware

http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf



GPU Hardware

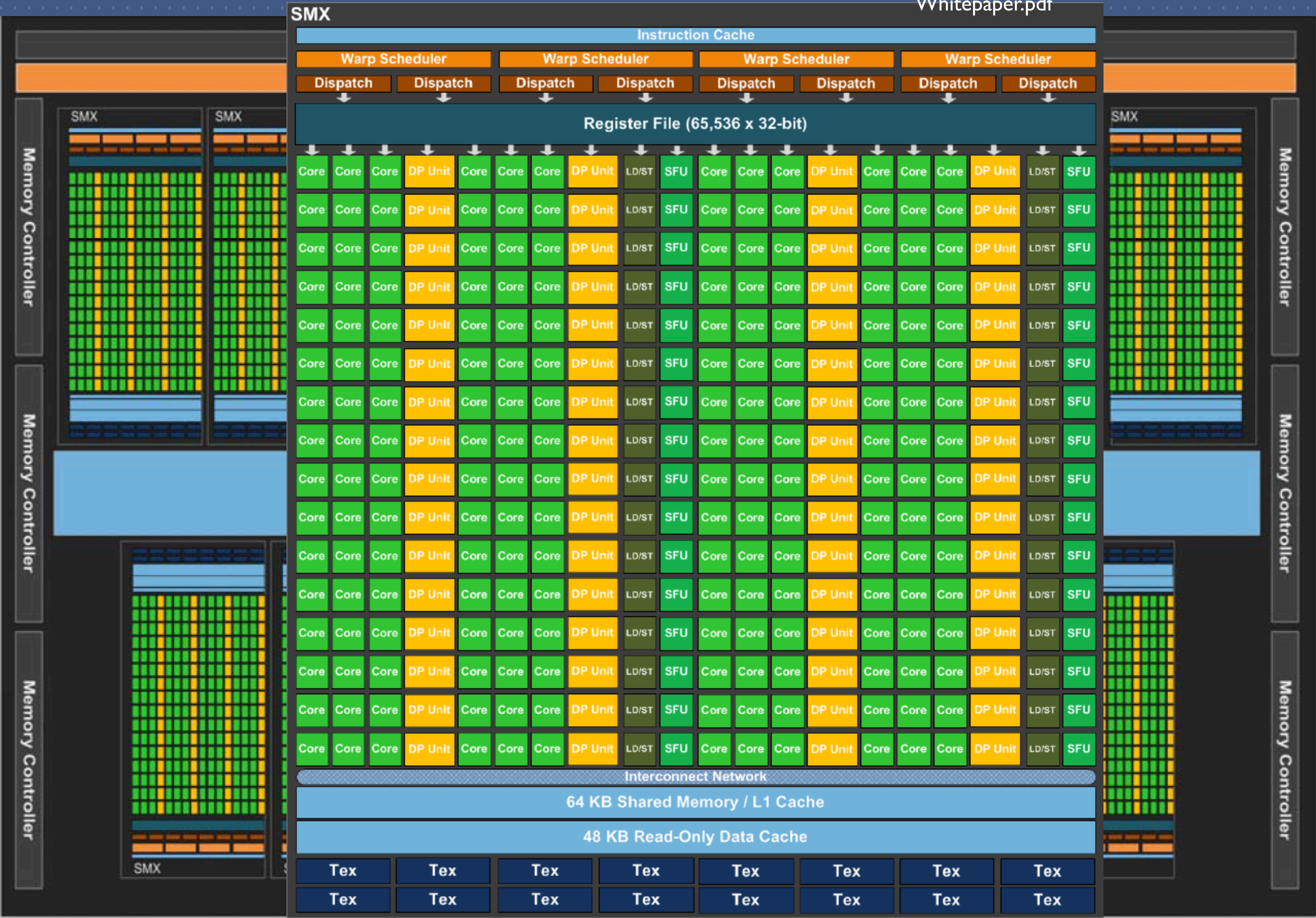
- Each Work-Item (thread) is executed on a Stream Processor (SP)
- SPs are located on one Stream Multiprocessor (SM or SMX)
  - Work-Groups are executed on SM's, where local memory is provided.
- Warp Schedulers execute threads of Work-groups on the SM's; a common optimization goal is to execute as many "warps" on each SM as possible.

GPU Execution Model (SIMT)

- Work Groups are executed via 32 thread launches (aka Warps)
- Warps follow Single Instruction Multiple Threading (i.e. all threads in a warp perform the SAME instruction); Optimization implications, will come back to this.
- Multiple warps can be executed concurrently on the same SM, aka "waves." Keplar GPU's can schedule 4 warps concurrently. However the maximum number of warps will not always launch; Depends on memory , number of threads per warp, number of threads per group, etc.
- Tails: If groups are not divisible by warp size, can affect performance.

GPU Hardware

http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf



GPU Hardware

http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

	FERMI GF100	FERMI GF104	KEPLER GK104	KEPLER GK110
Compute Capability	2.0	2.1	3.0	3.5
Threads / Warp	32	32	32	32
Max Warps / Multiprocessor	48	48	64	64
Max Threads / Multiprocessor	1536	1536	2048	2048
Max Thread Blocks / Multiprocessor	8	8	16	16
32-bit Registers / Multiprocessor	32768	32768	65536	65536
Max Registers / Thread	63	63	63	255
Max Threads / Thread Block	1024	1024	1024	1024
Shared Memory Size Configurations (bytes)	16K 48K	16K 48K	16K 32K 48K	16K 32K 48K
Max X Grid Dimension	2^16-1	2^16-1	2^32-1	2^32-1
Hyper-Q	No	No	No	Yes
Dynamic Parallelism	No	No	No	Yes

GPU obstacle:
Shortage of Memory

- Copying memory from the host CPU to GPU is a necessary step in all GPU kernel applications.
- Unfortunately, GPUs can only store a couple GBs of memory in total; even state-of-the-art Kepler can only hold around 8 GBs. Many applications require more, and as a result employ continuous reading and writing to the GPU. This can typically result in transfer latency performance hits.
- Possible solution: Simultaneously copy memory to the GPU while performing calculations on the previous memory transfer.

Lock-Step Execution

- Simple example is a divergent IF statemet:

```
If(get_local_id (0) < 4) {
        Do something
 } else {
        Do something else
 }
```

- SIMT ensures that when a warp of threads is launched for a work-group and encounters the above statement, both branches are executed (BAD).
- Two Solutions:
  - A) DON'T DO IT!
  - B) Make the branched logic modulo warp size.
  - Regardless should play around with group size

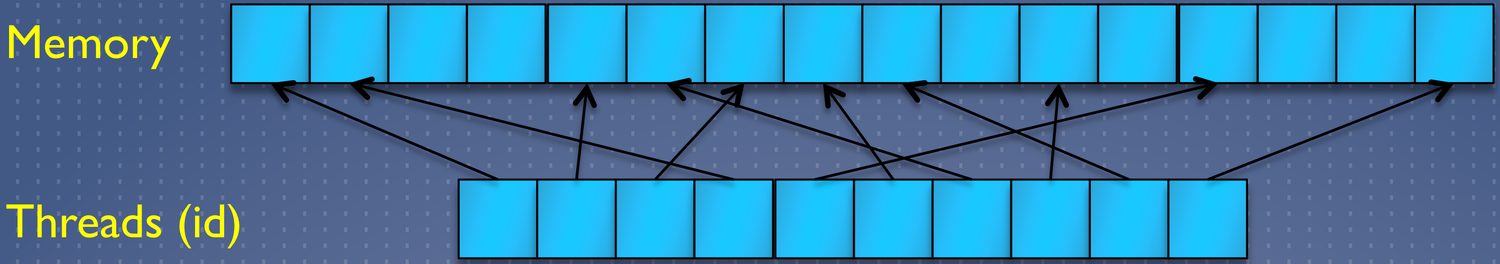
Memory Latency

- Fetching global memory requires many latency cycles (~ hundreds), and a result is one of the biggest performance hits.
- Local Memory on the other hand has much less latency cycles (~tens) but can have bank conflicts (described later)
- Solutions:
  - A) Hide latency with arithmetic calculation; while threads are waiting for a memory transfer other warps can be launched to do calculations. Depends on algorithm.
  - B) Do one copy from global to local memory and use the local memory speed to distribute the data. Can make use of Coalesced memory. NOTE: Global memory has GBs of data, whereas the local memory per SM has KBs. Very important to proceed

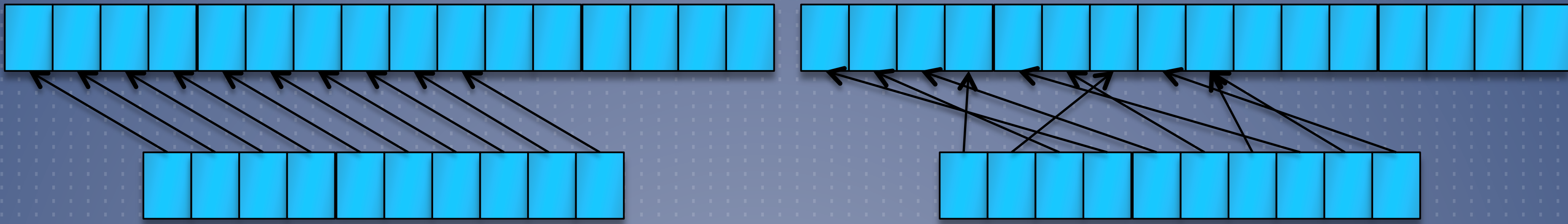
```
__kernel void addTwoArrays(__global float * arr, __global float* arr2,
                           __global float * return)

{
  return[get_global_id(0)]=arr[get_global_id(0)]
                          +arr2[get_global_id(0)];
}
```

Coalesced Memory Transfer

- If memory is accessed non-contiguously, memory fetches will be performed sequentially (BAD if from global memory)



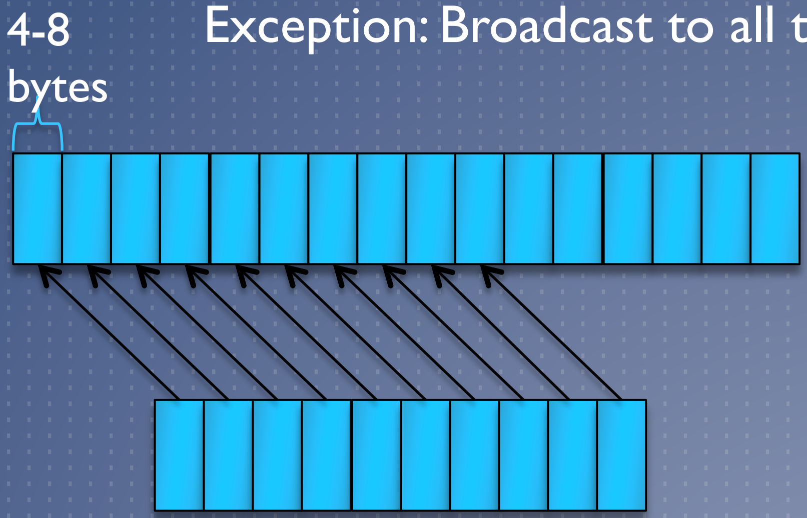
- If desired memory fetches are coalesced, the GPU can perform them all at once (modulo half warp size).

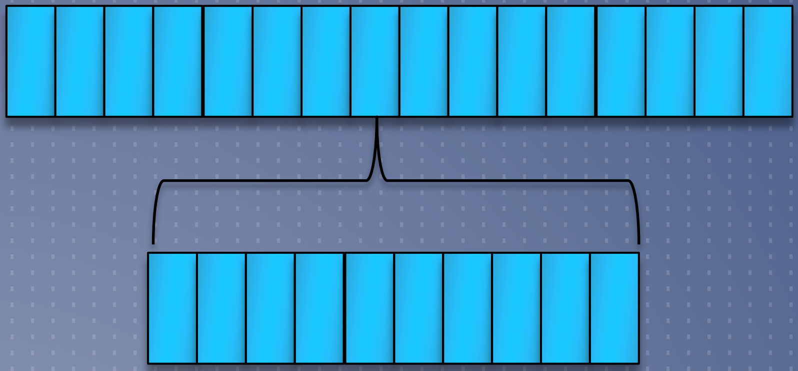


Newly Allowed

Bank Conflicts

- To avoid multiple global memory latencies, one can copy data to Local Memory for quick access. However, Local Memory is fetched with banks.
- Banks contain 4 bytes (Fermi) or 8 bytes (Kepler) of memory.
- GPUs typically contain 32 banks per SM
- If threads access different memory elements, then all fetches occur at maximum speed (GOOD). Otherwise, fetches are sequential (BAD). Exception: Broadcast to all threads is fast, can be very powerful





Broadcast

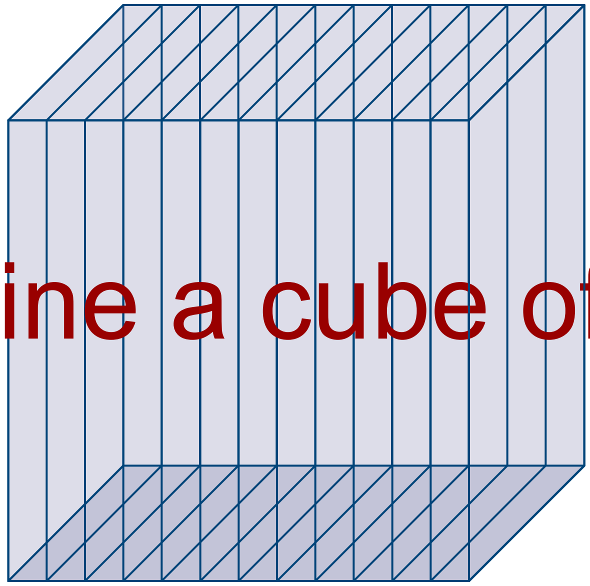
NVDIA OpenCL Visual Profiler

- Can profile kernel execution time, as well as host data transfer time.
- Can analyze memory bandwidth and instruction issue rate.
- Can report number of coalesced loads/stores
- Occupancy
  - Ratio of active warps per SM to maximum allowed.
  - Very informative measure of performance.

Exemplar: Short Range Force Solver

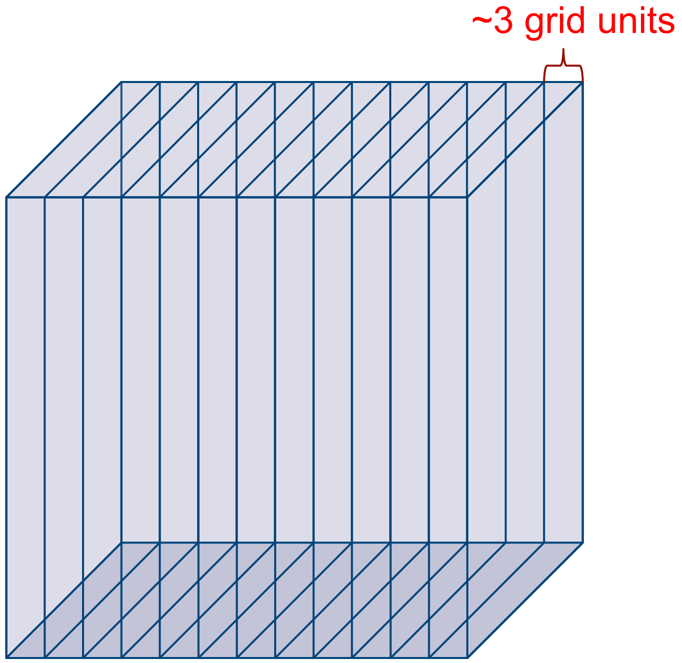
- Our N-body PM solver can resolve forces to ~ 3 grid units. We then require a short range solver to increase the resolution.
- A simple approach is to perform a brute force $O(N^2)$ nearest neighbor calculation (within radius of 3 cells) utilizing an accelerator such as a GPU (The P^3M Method).
- One could also use a tree method to reduce computation. We currently have employed such an algorithm, but is not currently accelerated.

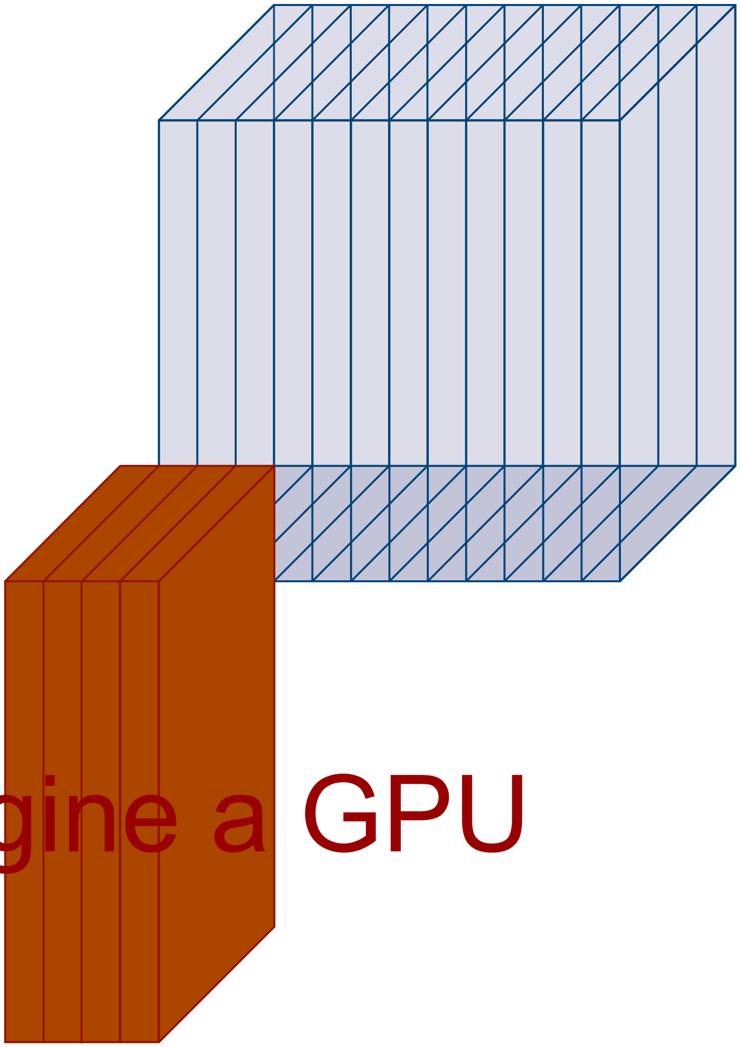
- The Brute Force method is a simple algorithm which combined with the GPU performance enchantment techniques discussed has proven to be a factor of 4-5 faster then the CPU tree code.
- NOTE: GPU code runs at approximately the same speed most redshift.



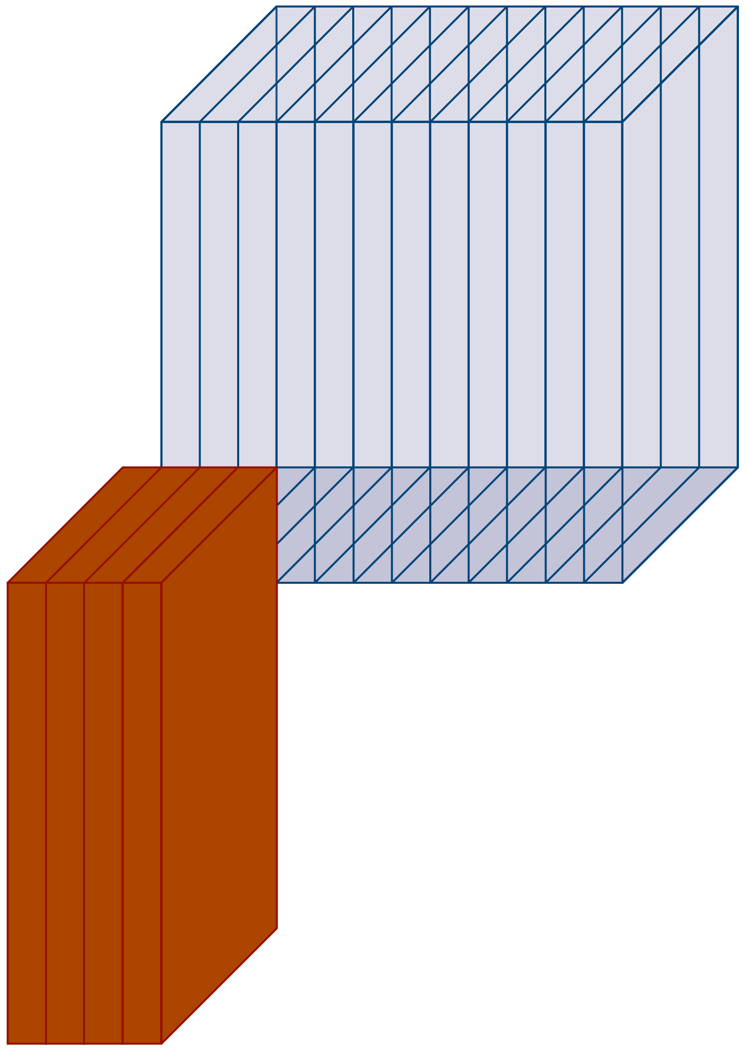
imagine a cube of data

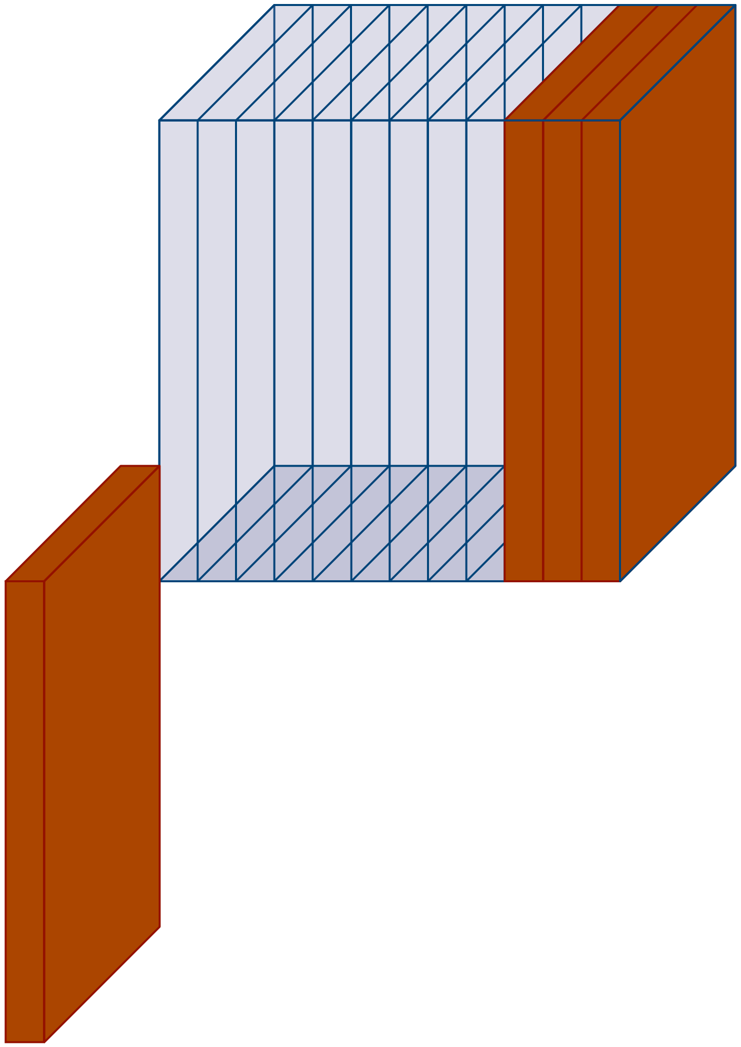
divide it into slabs

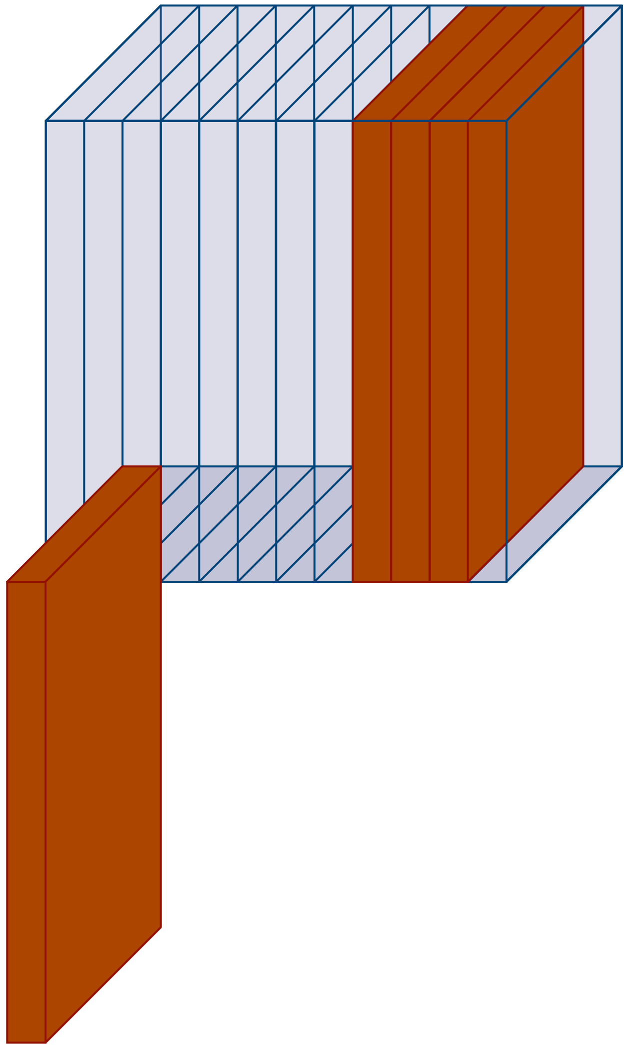

~3 grid units

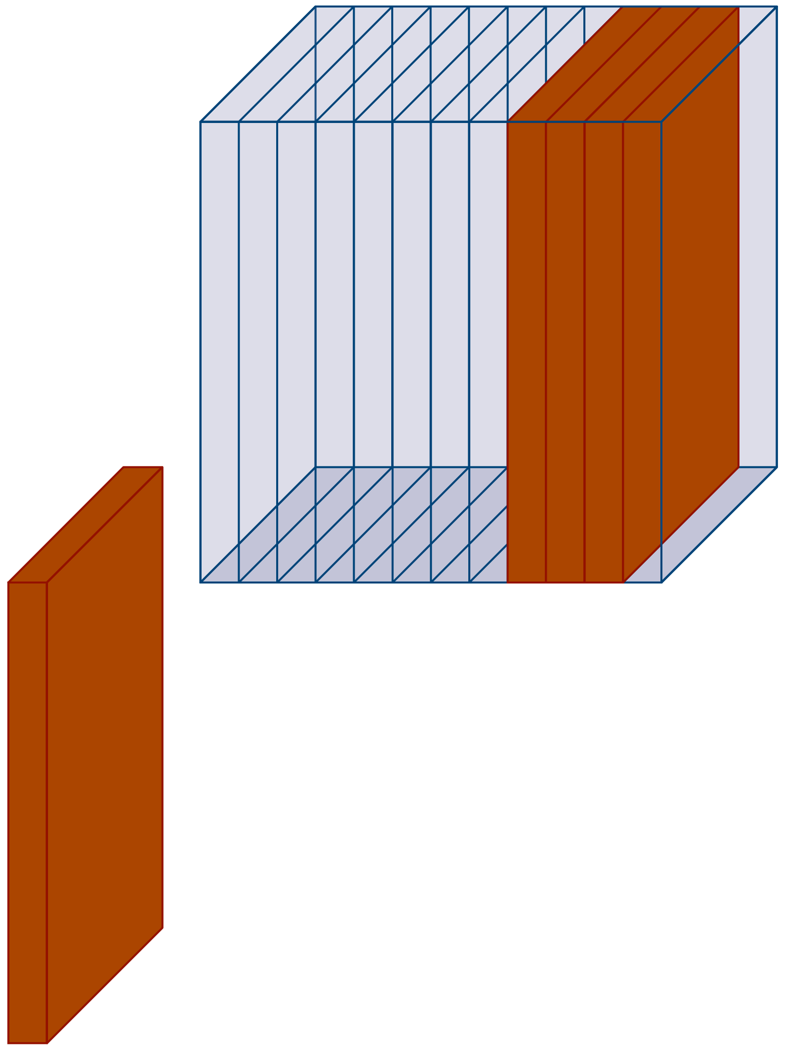


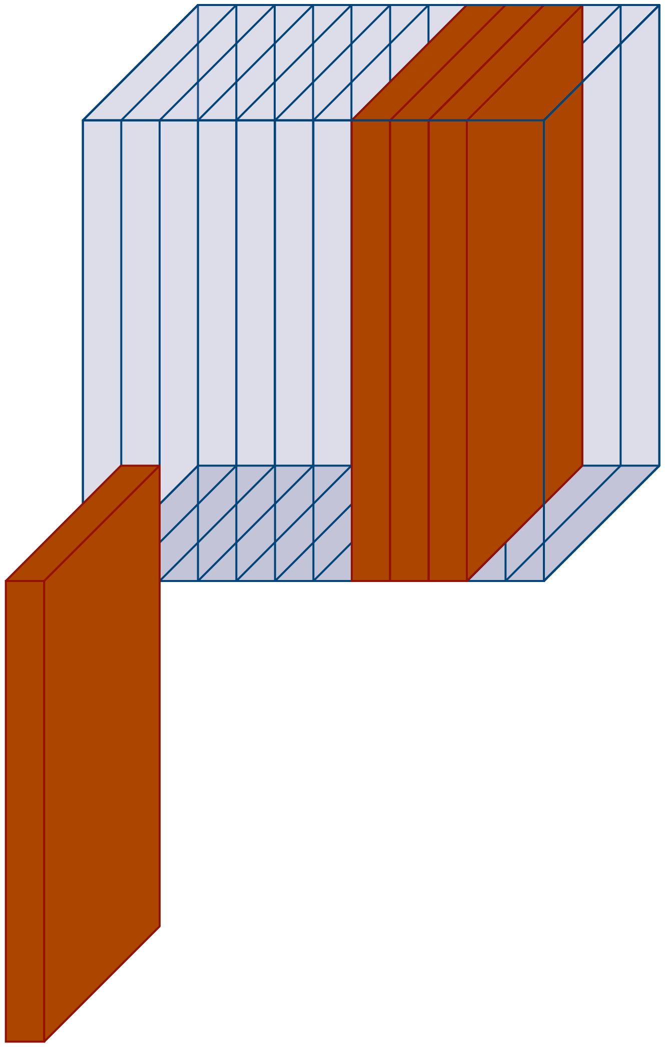
imagine a GPU

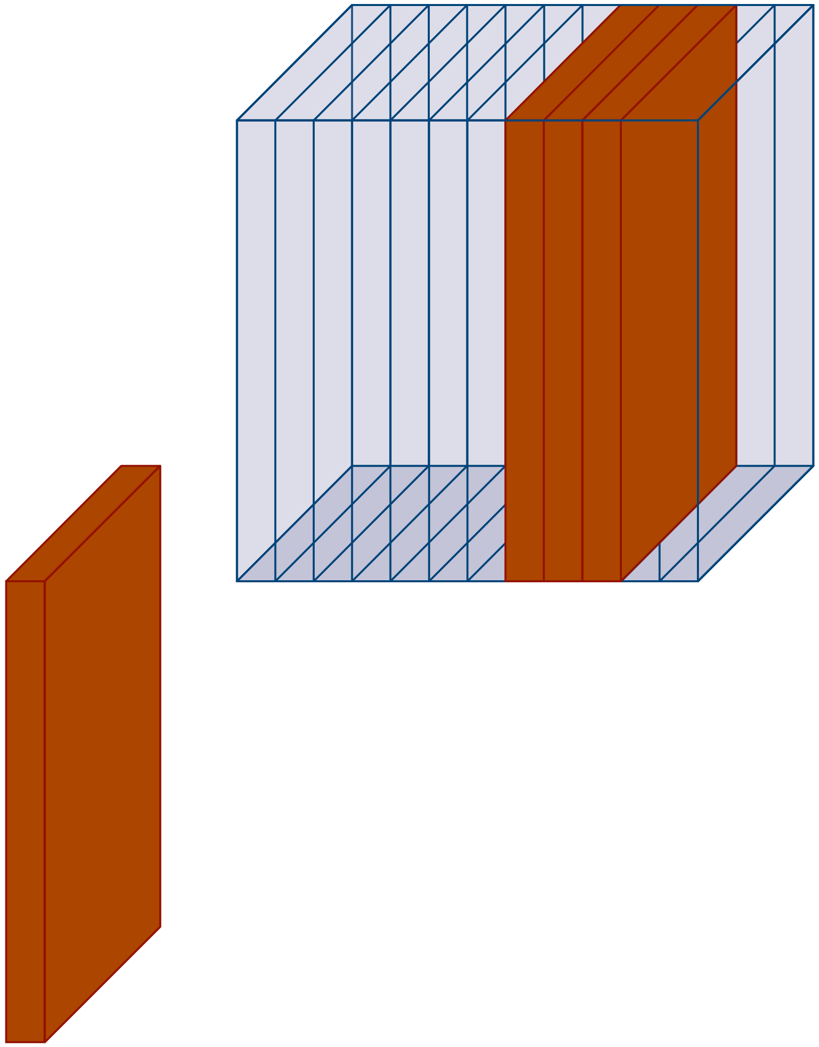


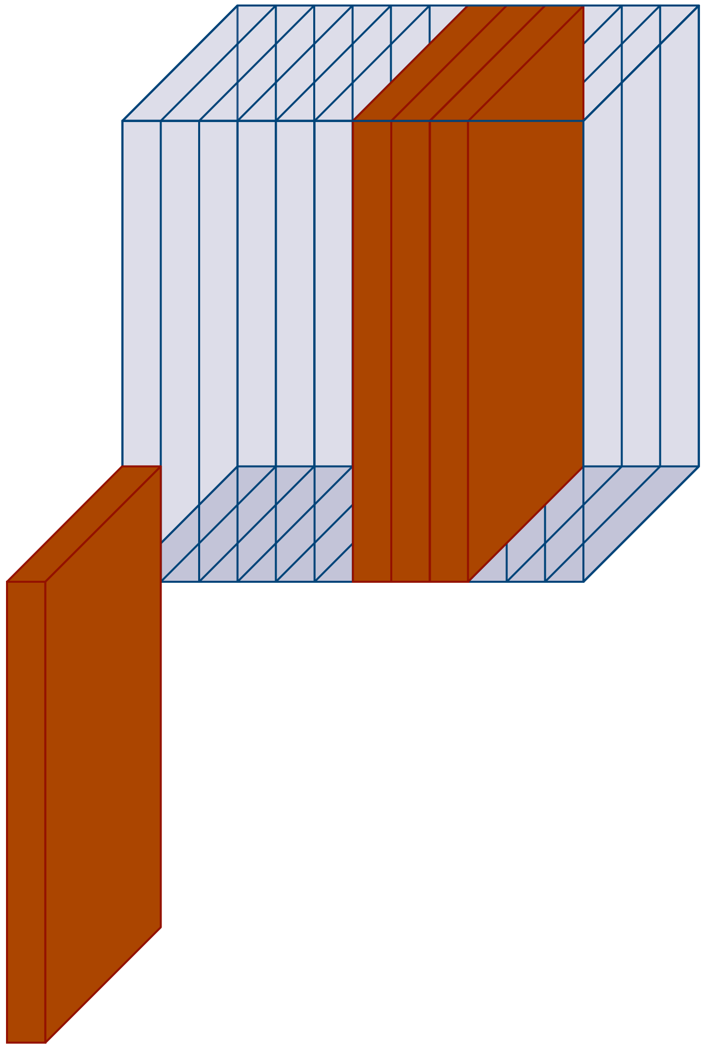


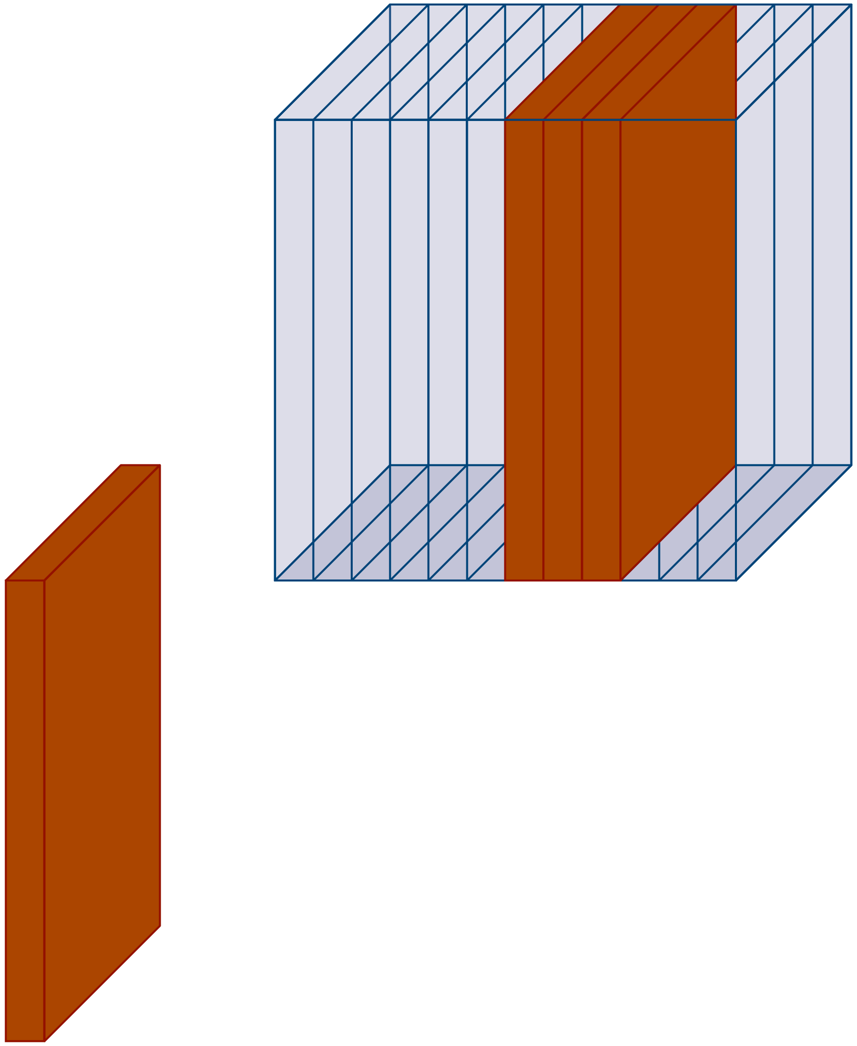


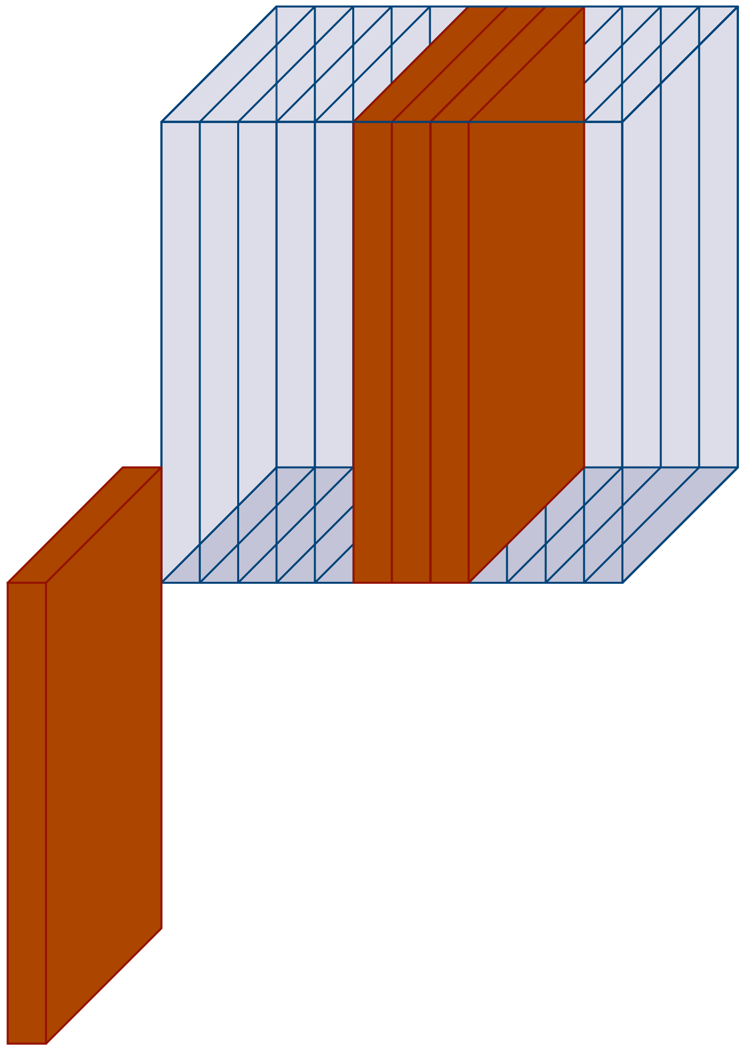




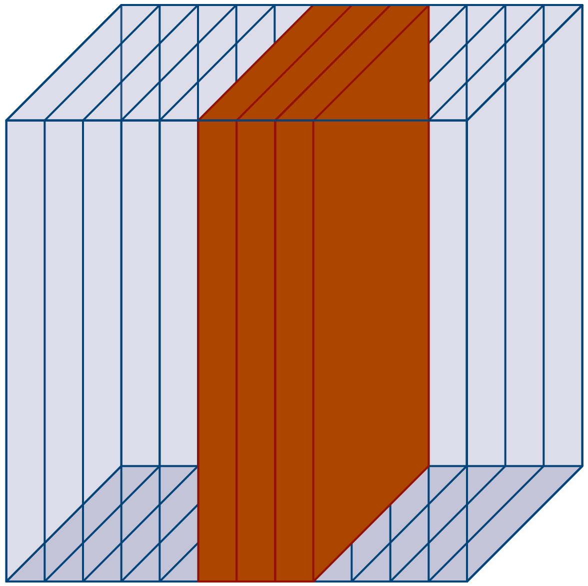


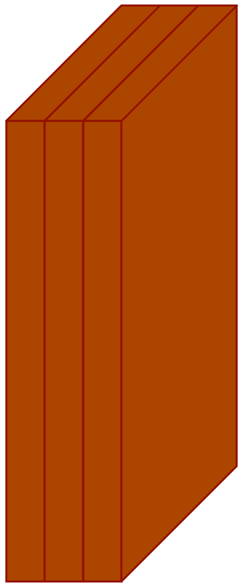


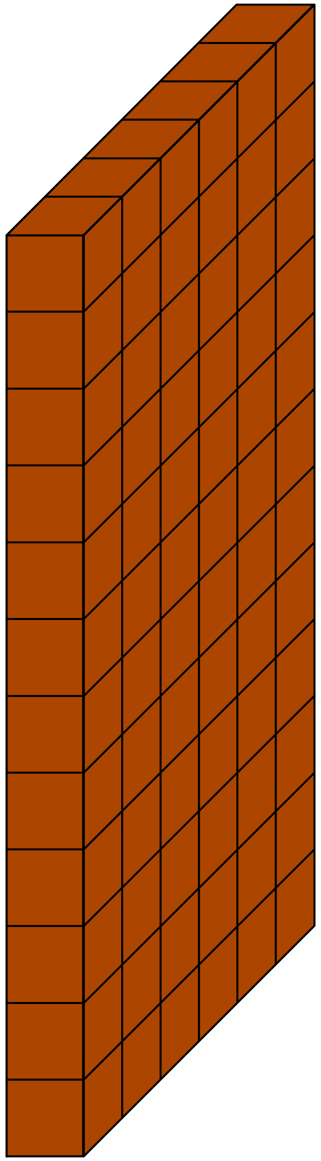


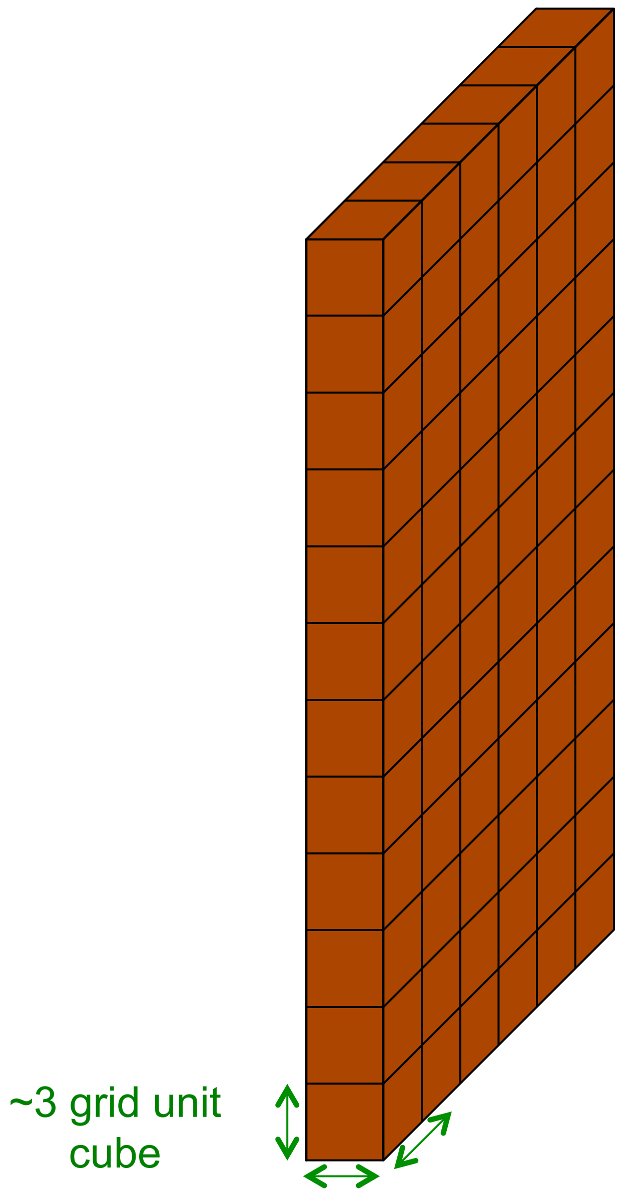


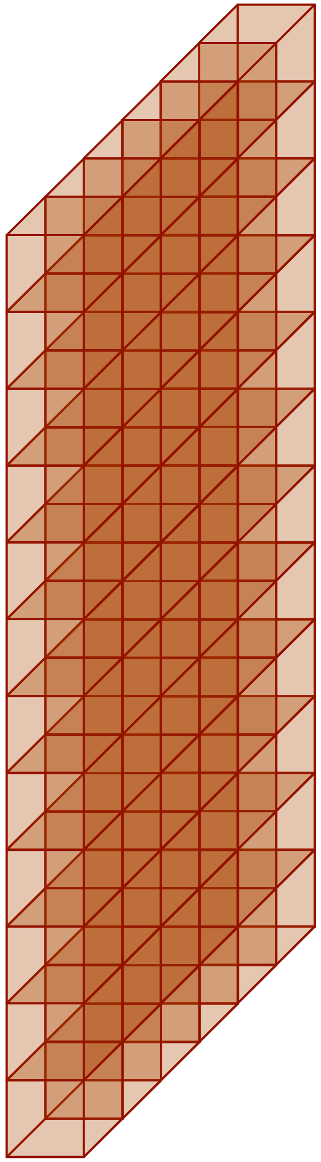
Keep Repeating

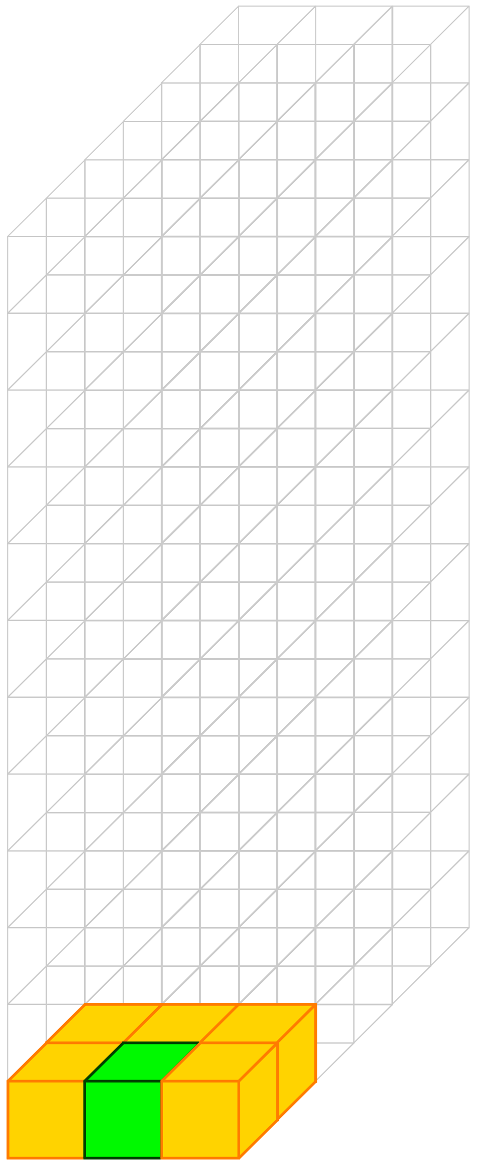


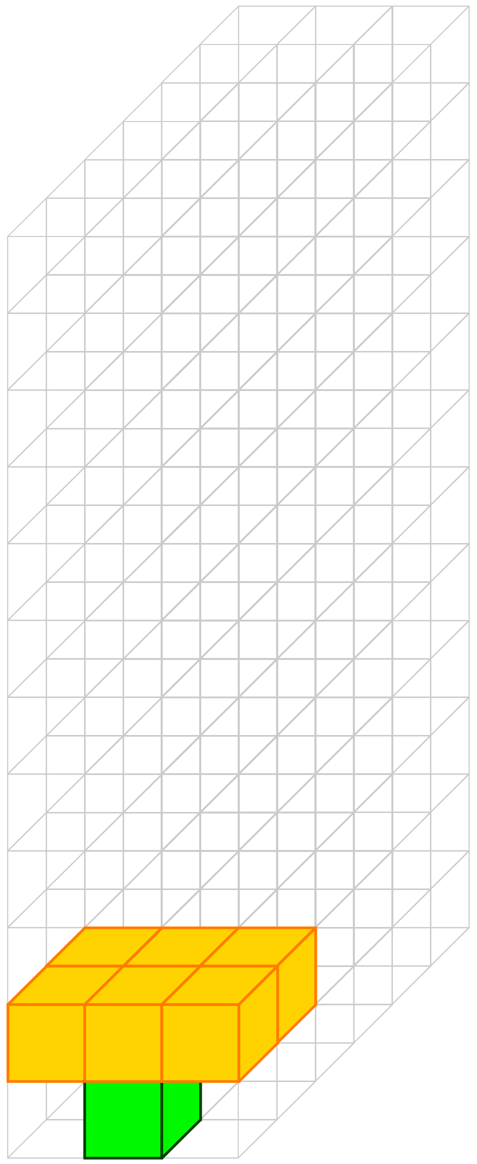


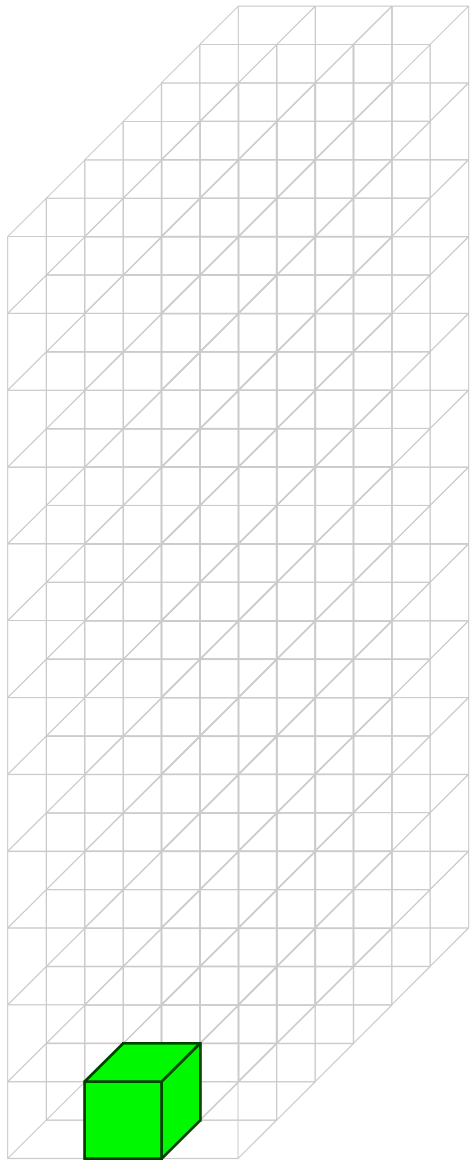


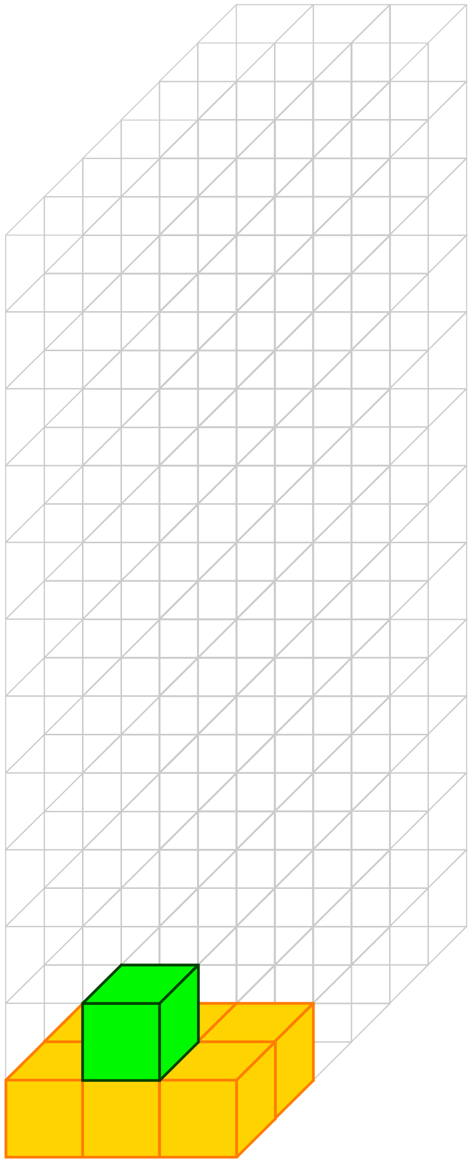

~3 grid unit
cube

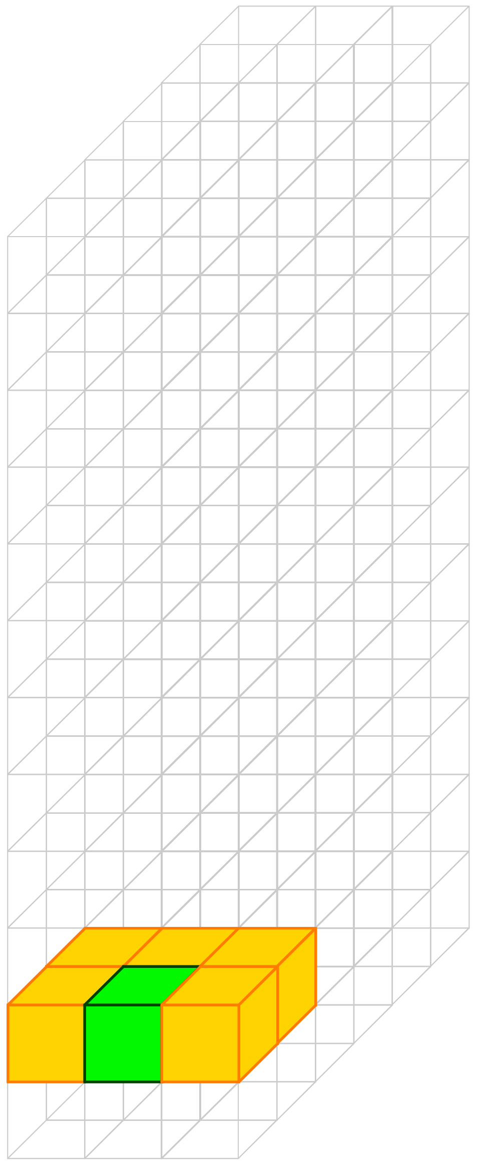


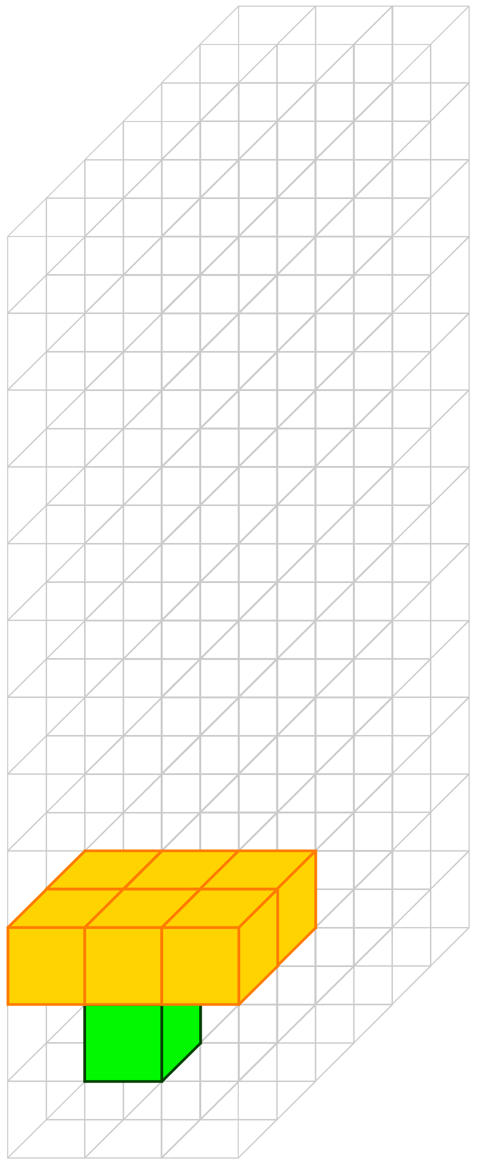


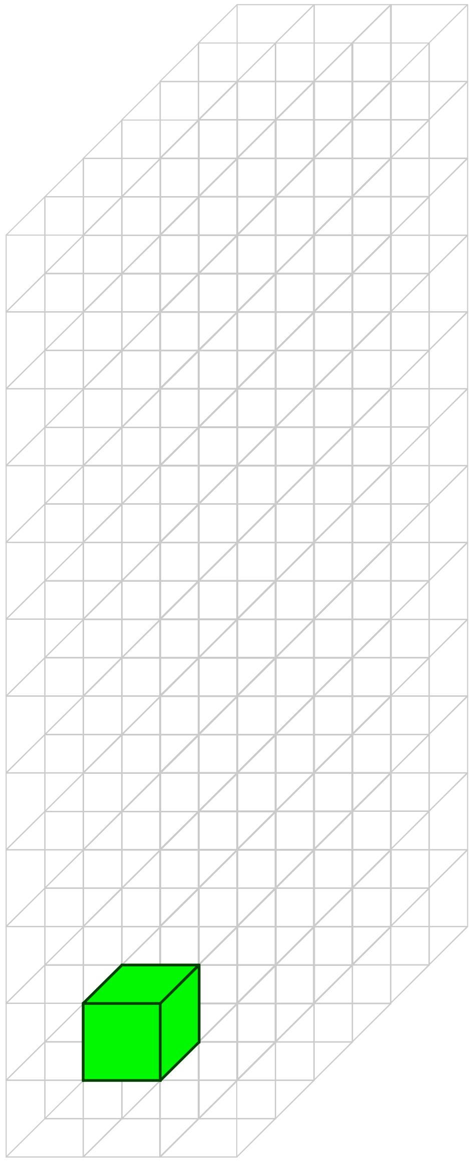


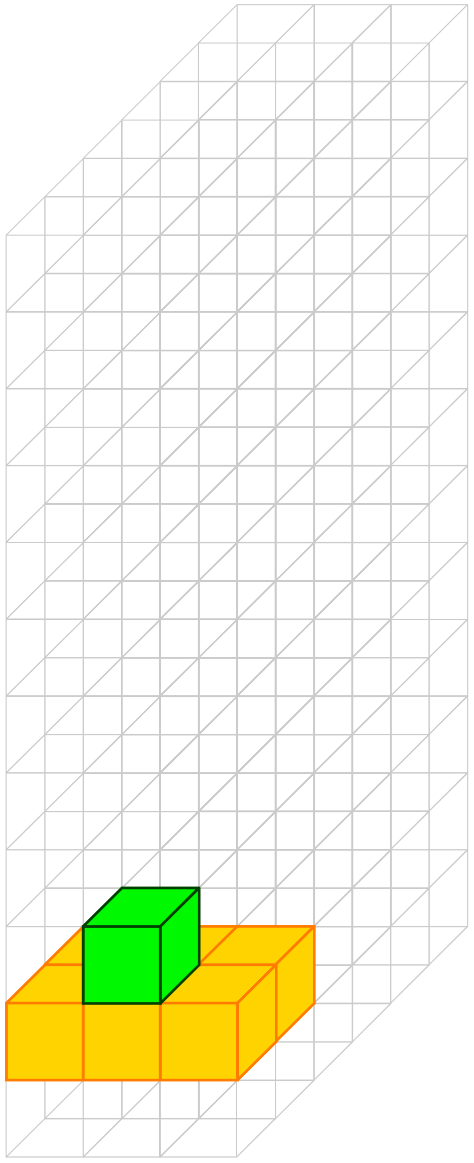


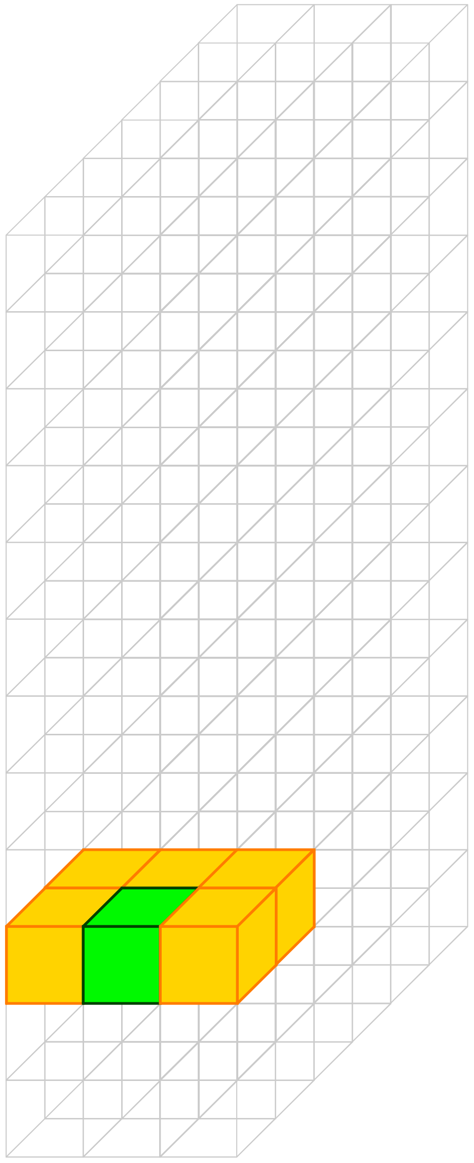


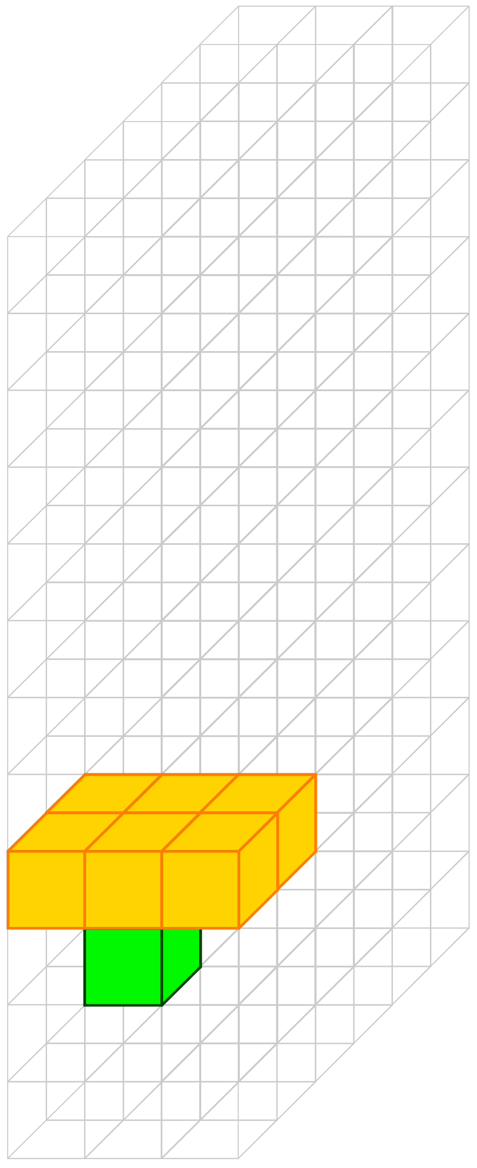


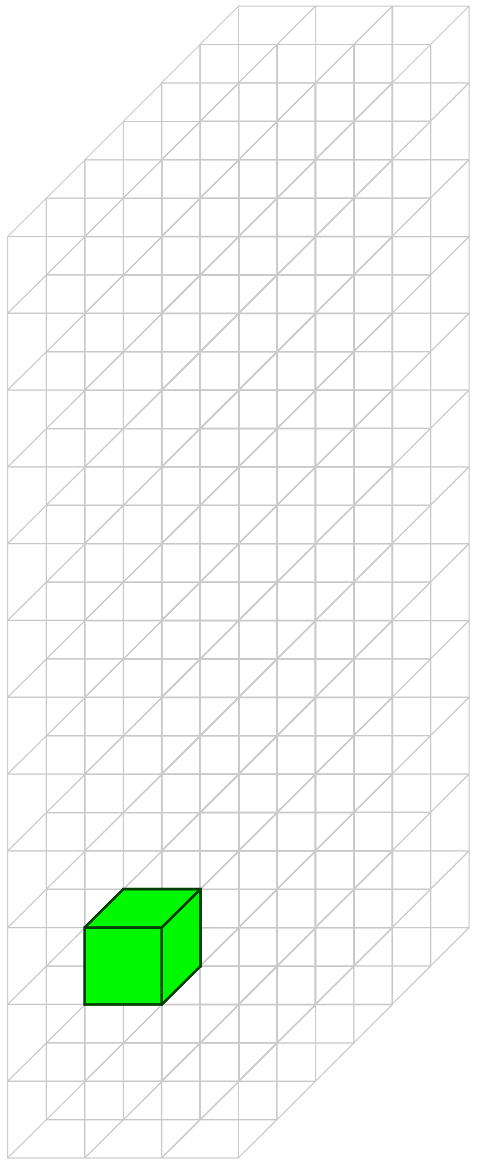


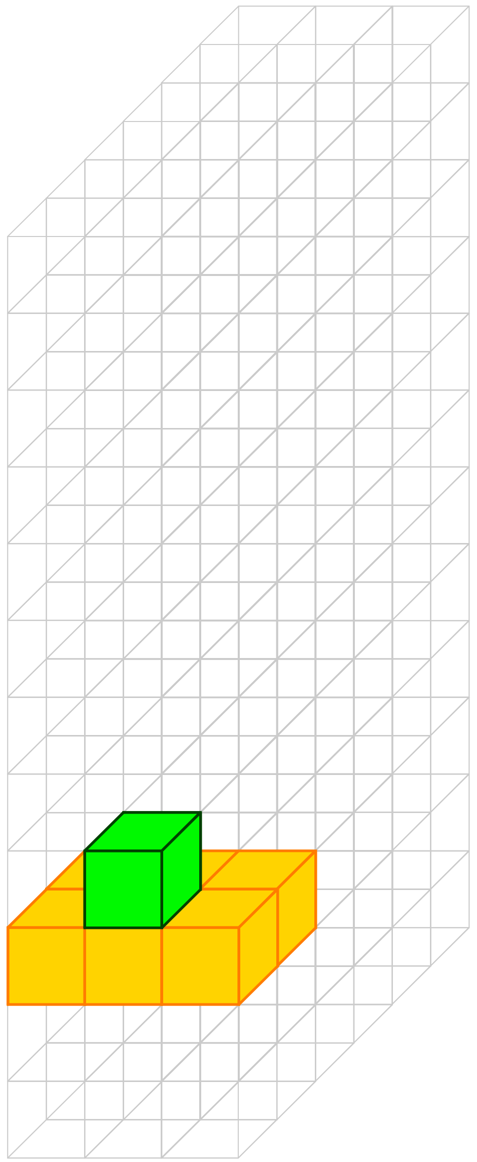


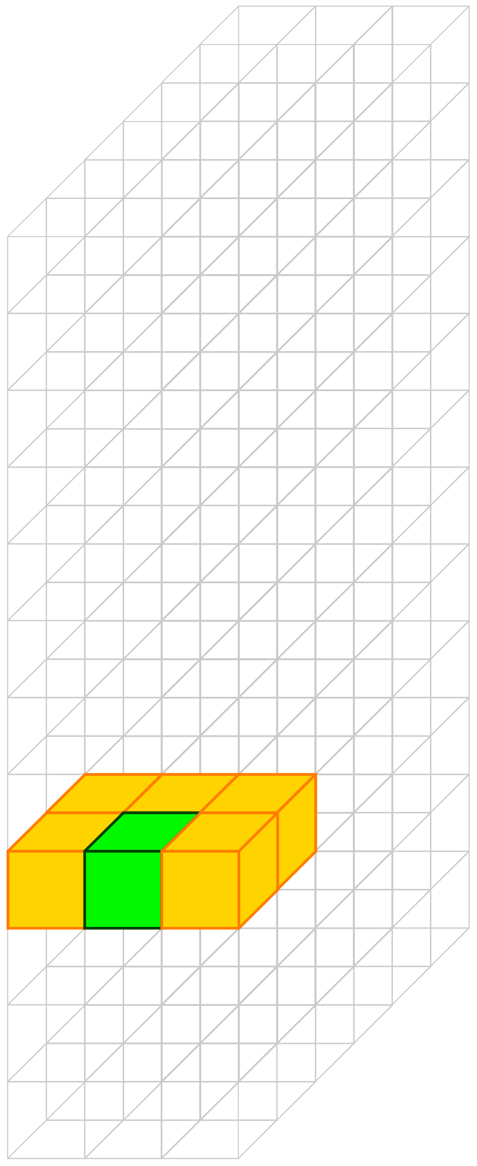


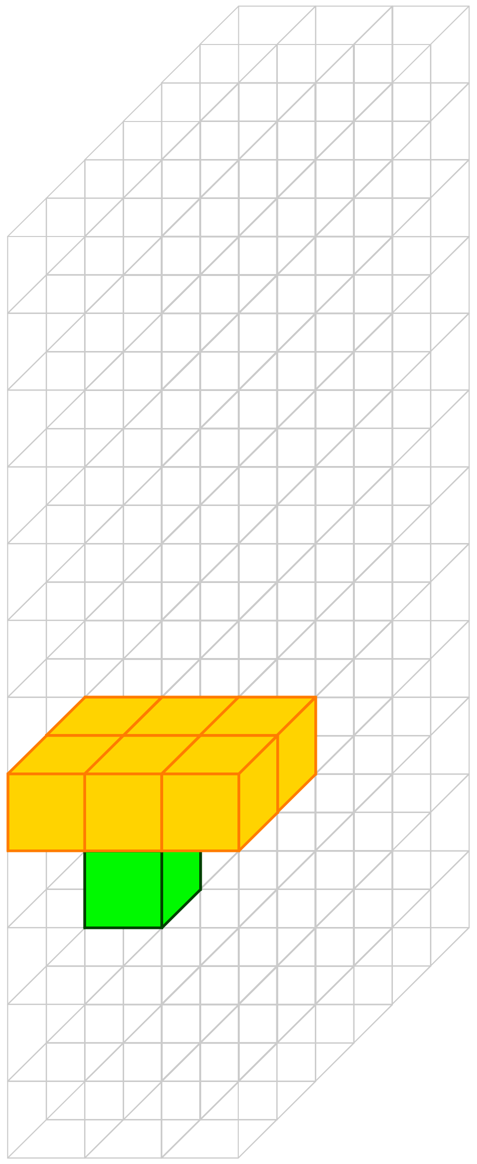


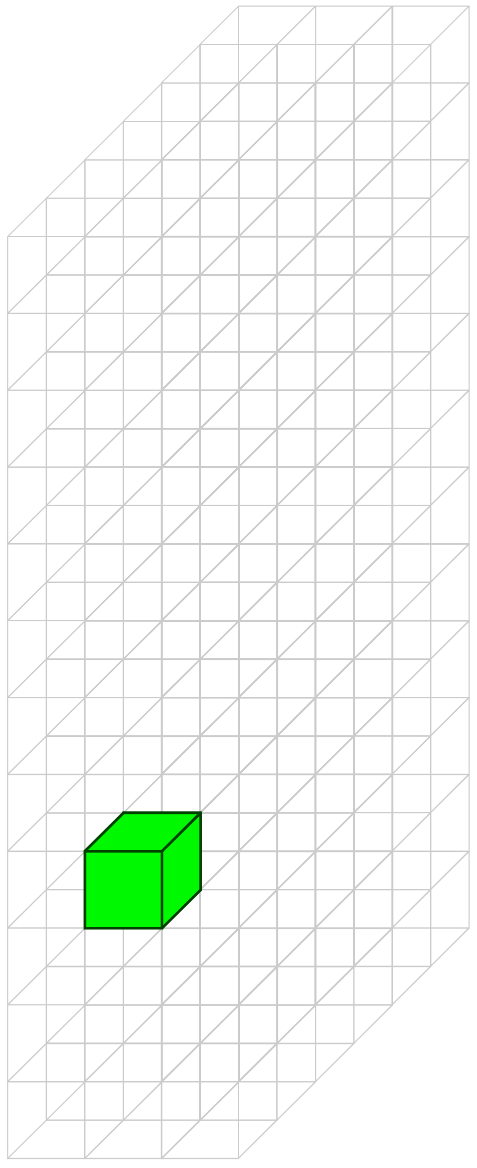


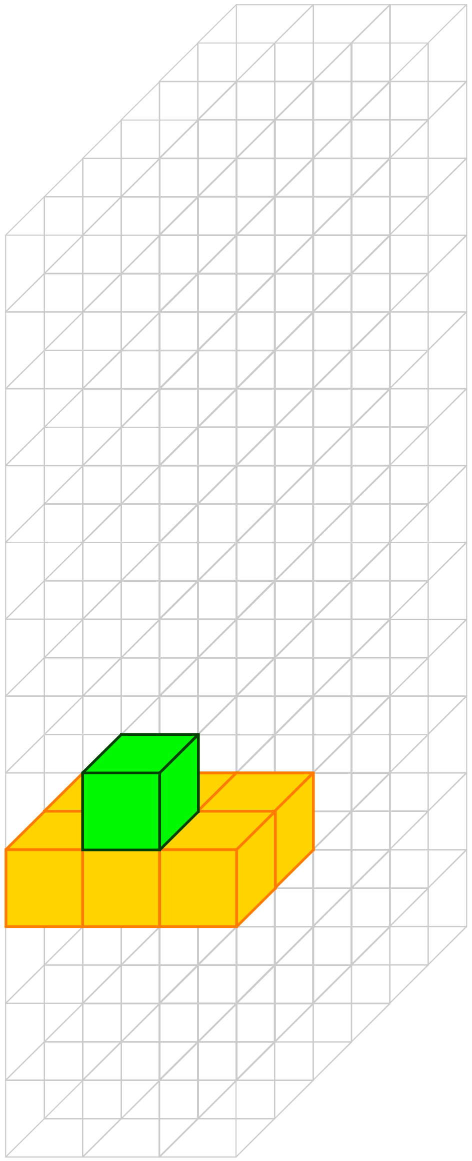


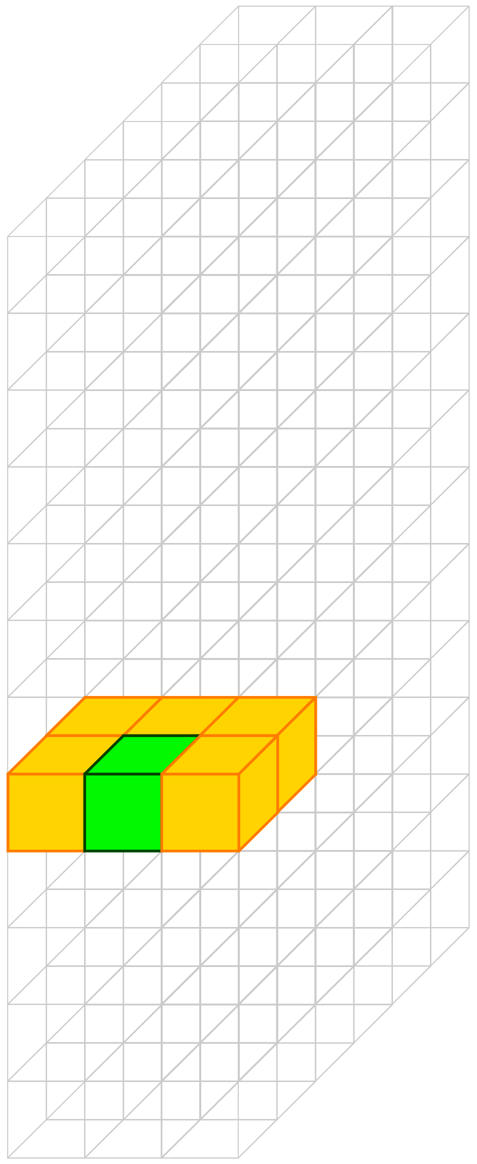


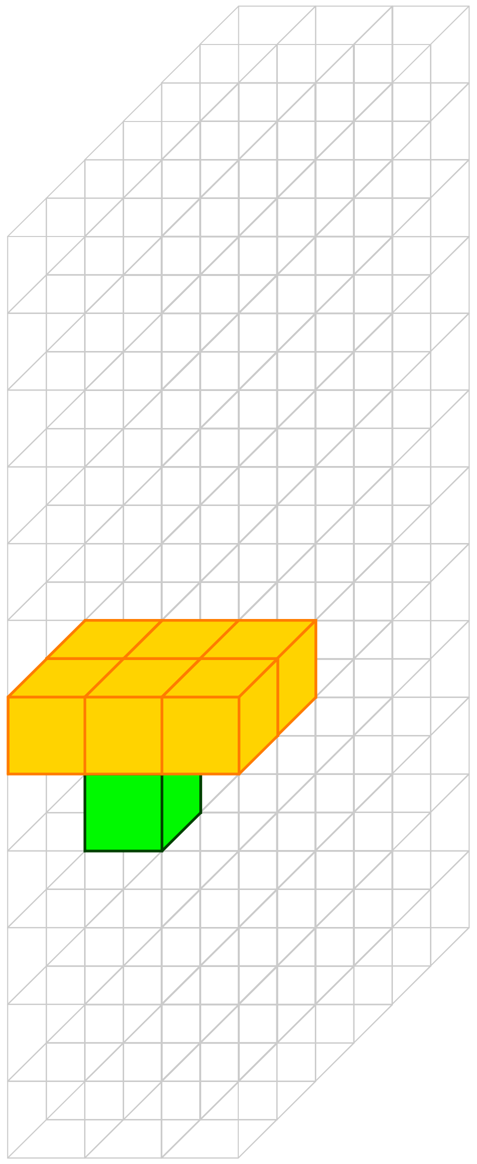




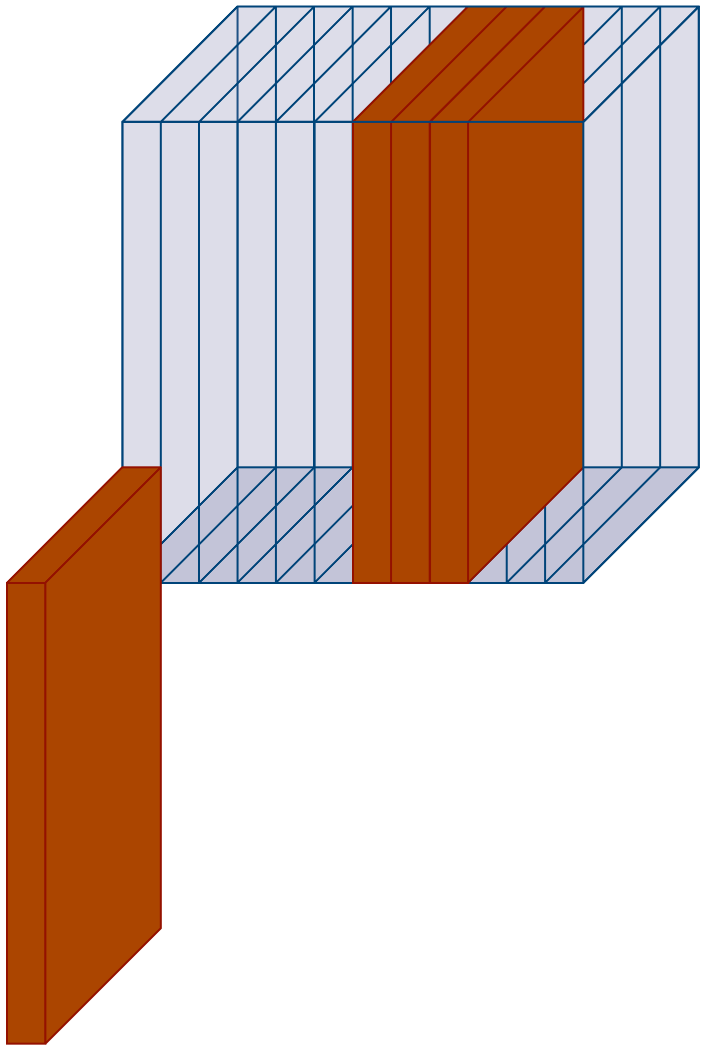


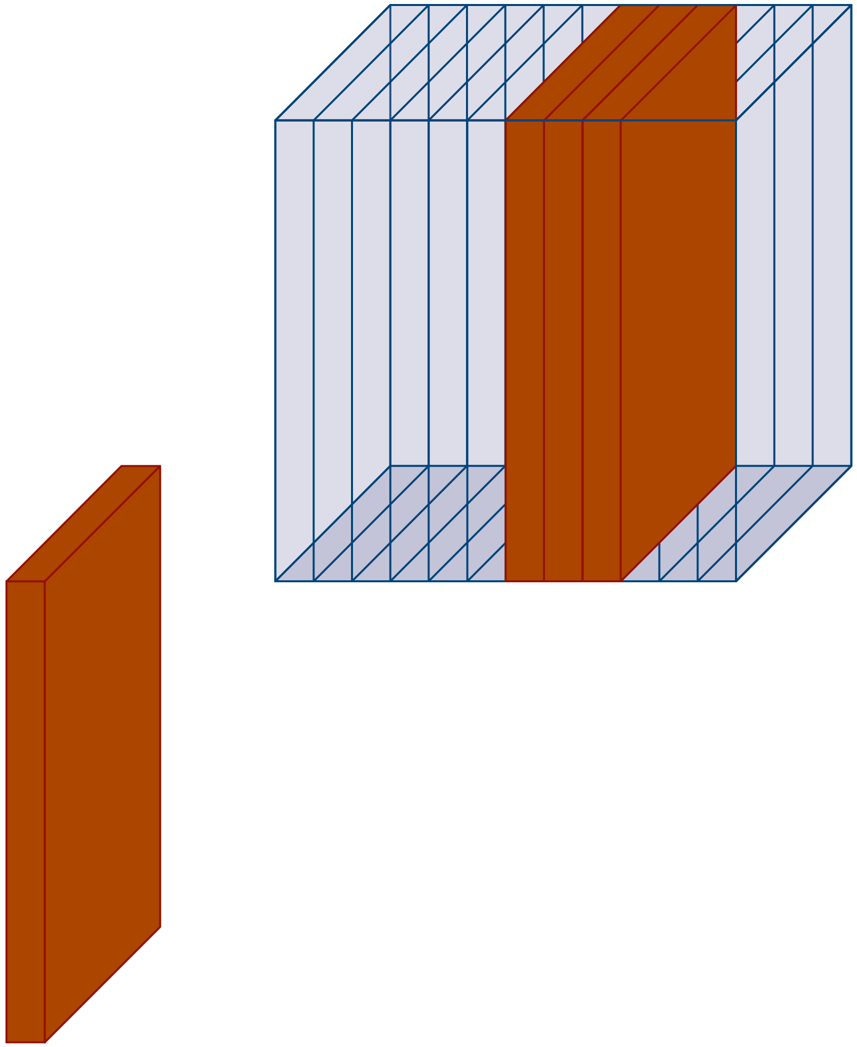


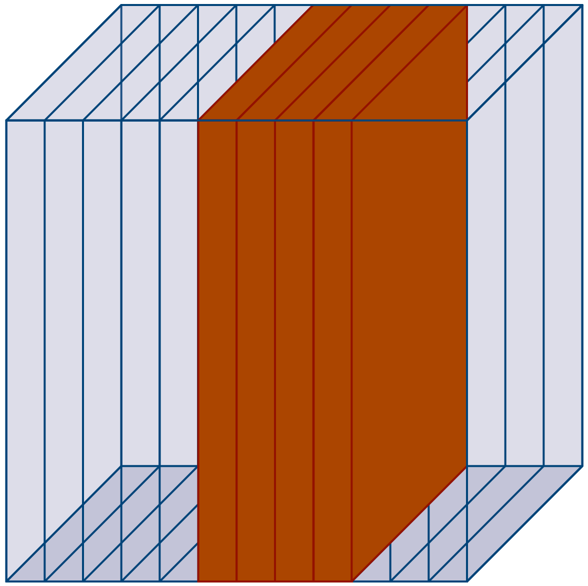


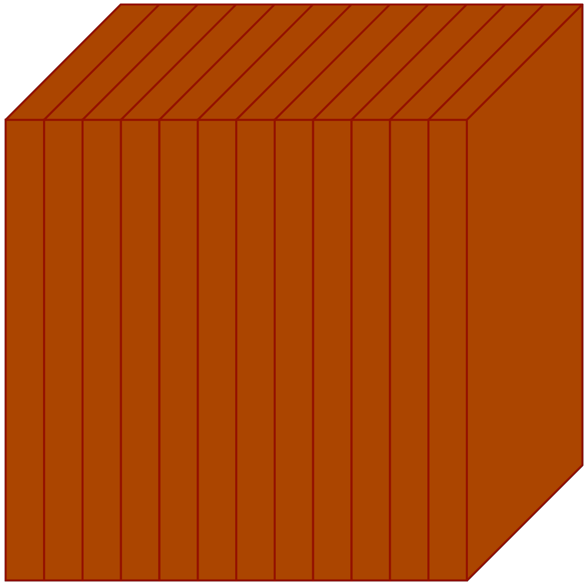


Keep Repeating









Optimization checklist:

- Memory Shortage: ✓
  - Algorithm only requires slabs of data, not the entire cube.
- Lock-step execution: ✓
  - As all particles in a group (chaining mesh) perform the same instruction -- calculate force from neighboring meshes -- there is no divergent logic within a work group.
- Global Memory Latency: ✓
  - Each thread caches one particle from neighboring mesh into local memory; (thus only one fetch per particle per group not per thread!)
- Coalesced Memory Fetching? ✓
  - Yes. As particle order does not matter for each mesh bin calculation, each thread can do a local cache sequentially, further reducing latency.
- Bank Conflicts: ✓
  - Use broadcast from local memory as group of threads need to fetch the same particles.

Conclusion

- GPU acceleration gains are completely determined by serial fraction of the algorithm.
- OpenCL allows one to use any heterogeneous platform vs. CUDA which is a more mature but vendor specific language.
- Multiple GPU specific considerations
  - Memory Storage
  - Lock-step execution
  - Global memory latency
  - Coalesced memory fetching
  - Bank Conflicts
- Profilers exist to aid you in determining performance
- Question? nfrontiere@gmail.com

OpenACC

- Similar to OpenMP, utilized directives
- Perhaps a good first step toward attempting acceleration.
- With every higher level language, one loses sophistication
- Example: Matrix Multiplication

```
!$acc kernels
do k = 1,n1
    do i = 1,n3
       c(i,k) = 0.0
       do j = 1,n2
          c(i,k) = c(i,k) + a(i,j) * b(j,k)
       enddo
    enddo
enddo
!$acc end kernels
```