ARGONNE ATPESC 10 EXTREME-SCALE COMPUTING

GPUs: less scary than you think

How thinking about HPC architectures made me somewhat decent at computational science

Max Katz

ECP Argonne NATIONAL LABORATORY

ATPESC 2013 at Pheasant Run Resort

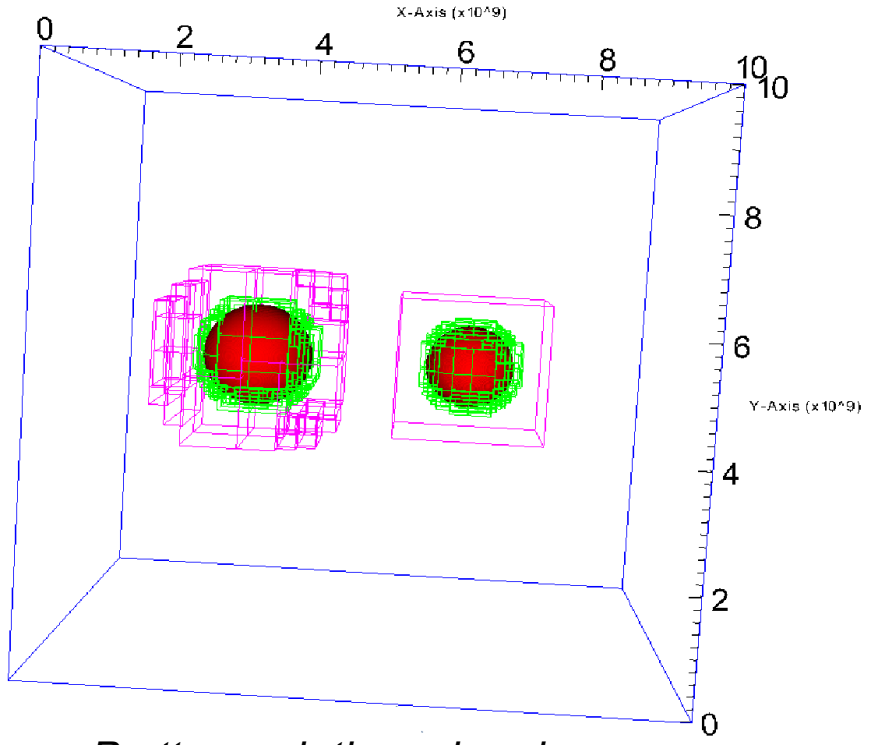
Where I was in summer 2013

- B.S. / M.S. in physics, did some programming in MATLAB and then Fortran

- I had completed the first year of my Ph.D. program at Stony Brook in computational astrophysics
  - My advisor, Mike Zingale, is a dear friend to this day

- I had begun being a developer for Castro, a parallel code built on top of AMReX (then called BoxLib)
  - Most of the physics kernels were Fortran 90, with the driver code in C++
  - I had taken a year of C++ as a freshman, but I saw `std::` one too many times to be interested

- Honestly, I was a mediocre programmer, but I loved it!
  - I maybe was not very prepared for ATPESC though?
  - Impostor syndrome is real



Pretty much the only science I had done in a year

My initial experiences with parallel programming

- BoxLib/AMReX were kind enough to provide MPI wrappers, so my main experience actually writing `MPI_*` was with toy problems like the Jacobi method for solving linear systems
  - I did actually write a few `!$omp parallel` statements, but all I really understood was “threads go brrrr”
- I started just as OLCF’s Jaguar system was being upgraded to a new system called Titan using GPUs from this company NVIDIA
- Since no one* knew how to use GPUs in 2012, NVIDIA and OLCF partnered to start doing “hackathons”
- I went to one of the early GPU hackathons, and I *really* did not like OpenACC 1.0, but at least we found a cool team logo

**No one I knew, anyway*

Then I became a corporate shill, but not for the money

- When I was finishing up at Stony Brook, I assumed I would be a postdoc at a DOE lab

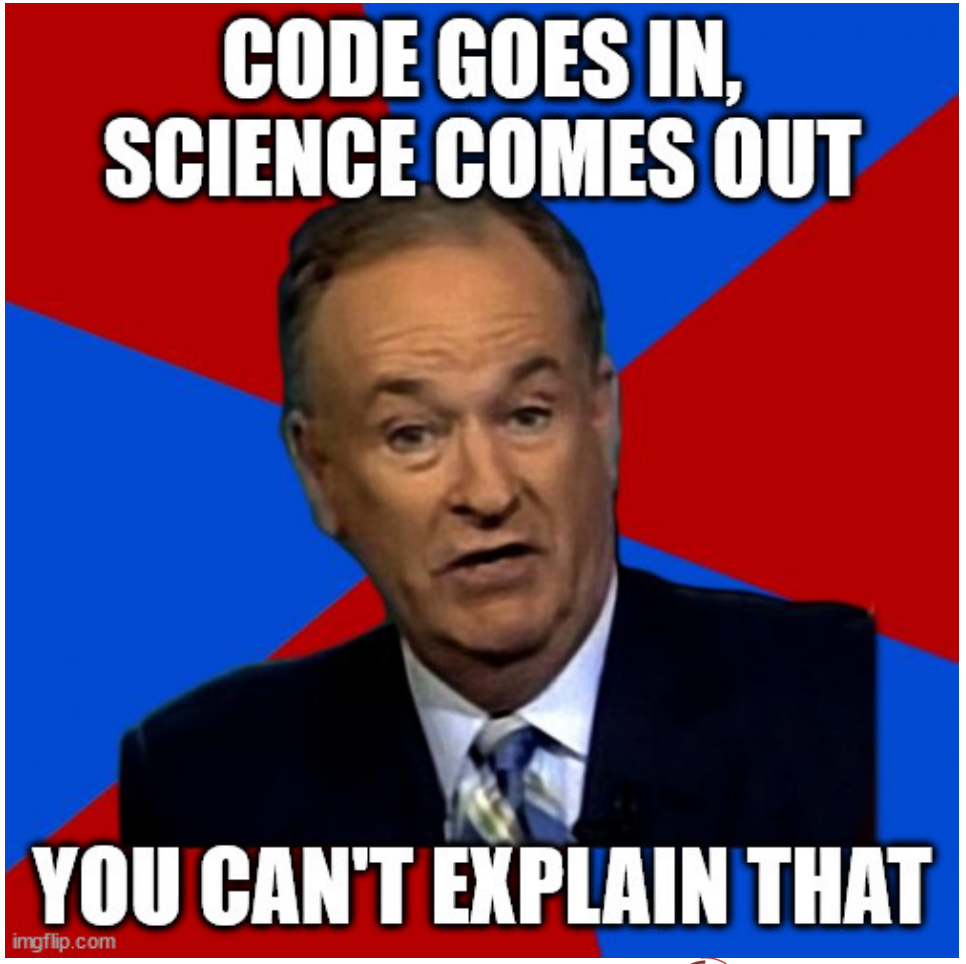
- But I had a random connection to someone from NVIDIA who I learned was hiring for a position working with DOE on their new Summit and Sierra systems, and this sounded challenging, so I did it
  - When I interviewed at NVIDIA, the division lead kept talking about machine learning and this new server platform called “DGX” and I smiled and nodded along and ignored all of it

I knew zero CUDA, and about as much about hardware

- Concepts I probably had encountered but did not really understand in 2016
  - Computer instructions, assembly code, and what compilers do in between `gfortran` and `a.out`
  - Registers
  - Caches
  - Hardware multi-threading
  - How software threads map to CPU cores
  - PCIe and server architectures
  - How MPI messages map to NICs
  - Whatever the heck “Dragonfly” and “3D Torus” meant
  - Parallel file systems

- Consequently, my understanding of HPC at the time:

- But I hadn’t felt a strong incentive to learn more!



Learning GPUs to teach about them

- One of my primary functions at NVIDIA was to help computational scientists use GPUs, which meant I had to be good at GPUs myself, so I started preparing to teach
- This started through mentoring at GPU hackathons and attending presentations given by my boss
- Eventually, I became confident to start leading training sessions on my own
- Let's look at some of the content I learned to teach about, and I will share some insights I gained

GPU Streaming Multiprocessors

- This was a typical slide I would show in a talk

- In order to make sense of this, I had to learn about a lot:
  - What “64-bit computing” means
  - Difference between scalar instructions and vector instructions
  - What instructions like “fused multiply-add” mean
  - What caches do
  - What registers are
  - …

VOLTA GV100 SM

	GV100
FP32 units	64
FP64 units	32
INT32 units	64
Tensor Cores	8
Register File	256 KB
Unified L1/Shared memory	128 KB
Active Threads	2048

So why do we have to think about all this?

- Limitations of GPUs run much deeper than not being able to use `std::cout`

- A register is the source and/or destination of all mathematical instructions

- For example, `x[0] = y[0] * z[0]` could mean:
  - Read `y[0]` from DRAM (main memory) and store it in a register `R0`
  - Read `z[0]` from DRAM and store it in a register `R1`
  - Perform the multiplication of `R0` and `R1` and store it in a register `R2`
  - Write register `R2` to `x[0]` in DRAM

- The maximum number of (32-bit) registers per thread on NVIDIA GPUs is 255
  - In other words, you get ~100 double precision variables in thread-local/thread-private memory
  - 90% of you* will naively want to write code that uses more local memory than this
  - 100% of you* will accidentally write code that uses more local memory than this

**extremely scientific estimates*

GPU restrictions give us an excuse to review our code

- Consider the following code representing some operation on a 2D grid:

```
for (int j = 0; j < N; ++j) {
    double tmp_input[N], tmp_output[N];
    for (int i = 0; i < N; ++i)
        tmp_input[N] = input[i + j * N]; // load from main memory
    do_physics(tmp_output, tmp_input); // calculate over the x-dimension
    for (int i = 0; i < N; ++i)
        output[i + j * N] = tmp_output[i]; // store to main memory
}
```

- Why would we write code like this?
  - Vectorization is explicit, and your advisor learned to make vectorization explicit when they were a grad student
  - `do_physics` might be a legacy routine that happened to operate on an array and it was never changed
    - Perhaps there's a good reason it operates on an array, like it is a stencil operation that reuses data
  - Because we are giving a talk and need to come up with an example to make a point

GPU restrictions give us an excuse to review our code

- If we were to parallelize over the j dimension, the code we just looked at will compile and run on a GPU, but if N >> 100, it may run extremely slow

- What will likely happen is that the local memory "spills" to the caches or back to DRAM
  - i.e. there weren't enough registers to store all of `tmp_input`/`tmp_output` at the same time
  - So by the time we get to the end, we had to store at least some of the data in memory and stream through it
  - This means that we might be increasing by a factor of several the number of DRAM accesses

- Typical GPU memory bandwidths:
  - PCIe/NVLink: O(10 – 100 GB/s)
  - DRAM: O(1 – 5 TB/s)
  - L2 cache: O(5 – 10 TB/s)
  - L1 cache: O(10 – 25 TB/s)

- The more we have to go back to DRAM, the worse our performance gets compared to staying in cache

So why did I bring this up?

- I could have just said “avoid using large amounts of local memory” and left you with a heuristic
  - But you would not be stronger as a computational scientist, because you would not understand why

- Ultimately, it’s hard to understand why without *some* mental model of the chip architecture (and system architecture) and *some* understanding of typical performance numbers

- When I started writing OpenACC code, I got really frustrated by all of the compiler limitations
  - (And I have worked with a lot of researchers who get frustrated)
  - But I had no mental model of how a GPU worked, so I assumed it was just lazy PGI compiler developers
  - In fact, many of the contemporary limitations in OpenACC (and OpenMP target) reflect the underlying architecture; if the compiler team didn’t implement something, it’s often because they’re protecting you

- It was only years later that I was able to both understand the programming model and write good code from the beginning, and read good or bad code by inspection, because I learned how GPUs work

What else can we learn from this example?

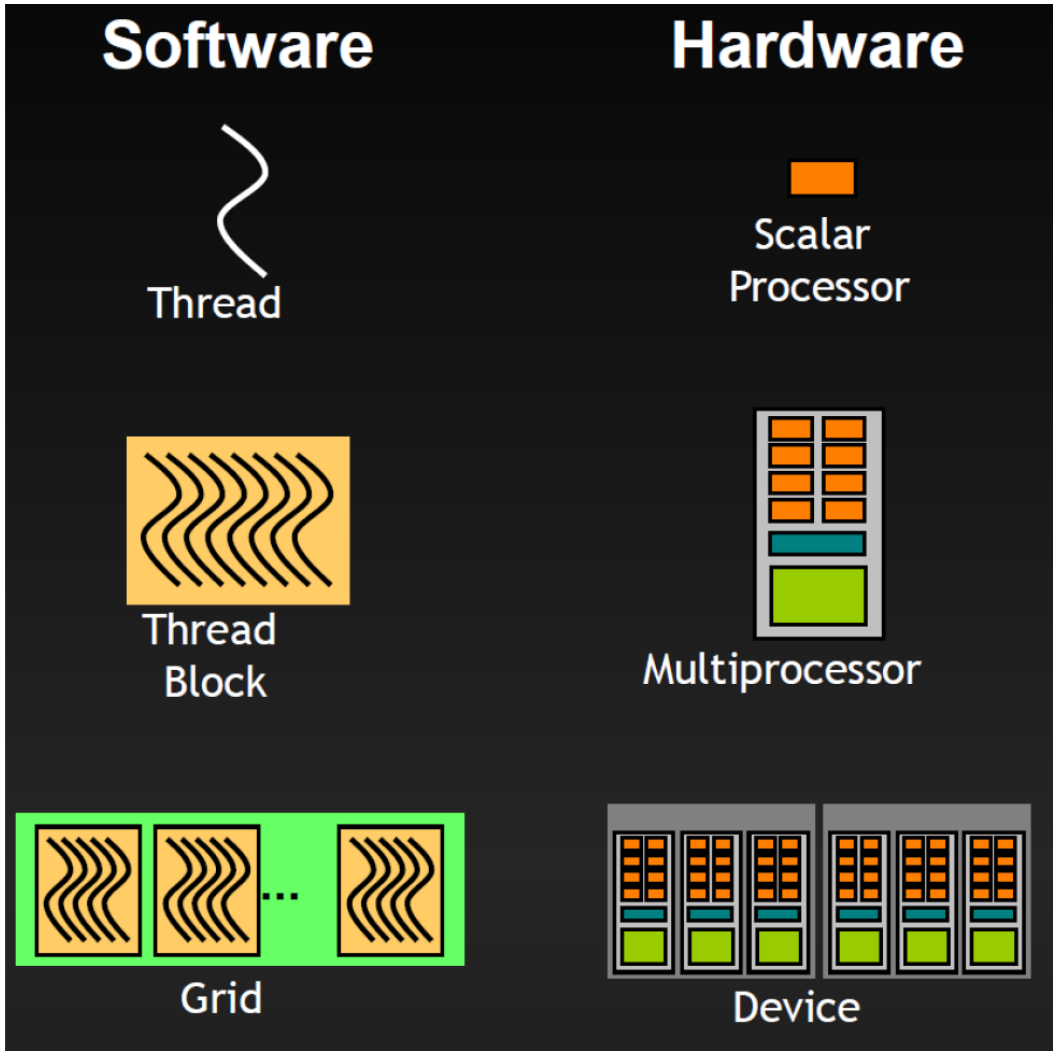
- Memory access patterns matter

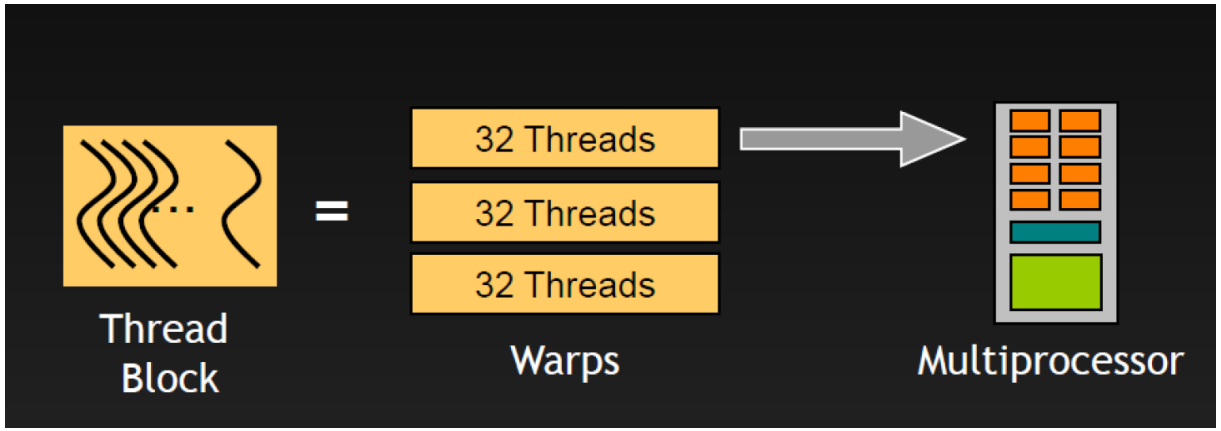
```
#pragma acc parallel loop                    // assume only parallelized over j
 for (int j = 0; j < N; ++j)
     for (int i = 0; i < N; ++i)
          tmp_input[N] = input[i + j * N];
```

- GPUs only achieve peak throughput when they coalesce memory accesses
  - This means combining the memory accesses from multiple threads into a smaller number of transactions

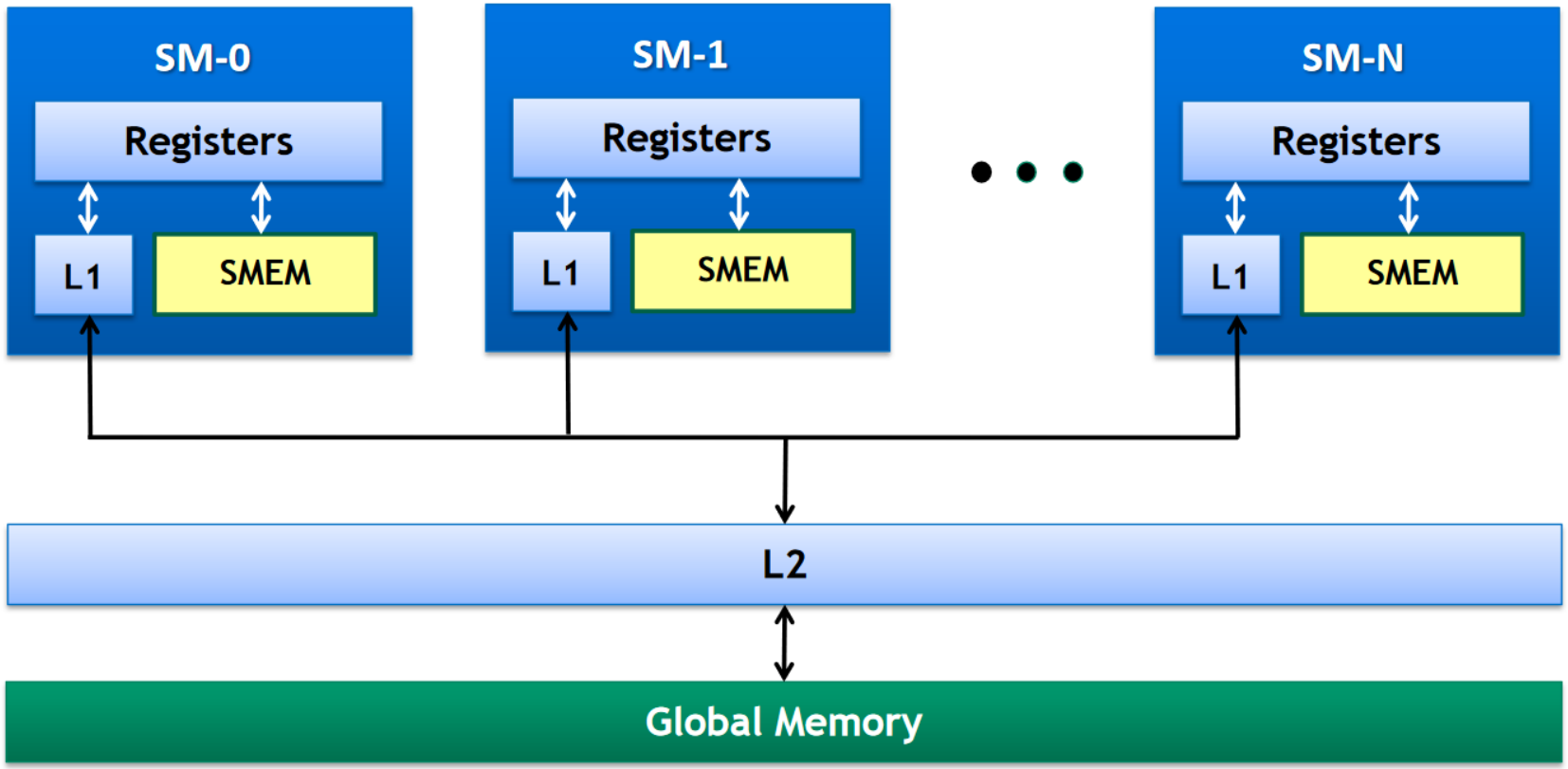
- I can give you a heuristic that coalesced accesses require consecutive threads accessing consecutive array indices, but it's better to understand why that's true

Review: GPU parallel execution model



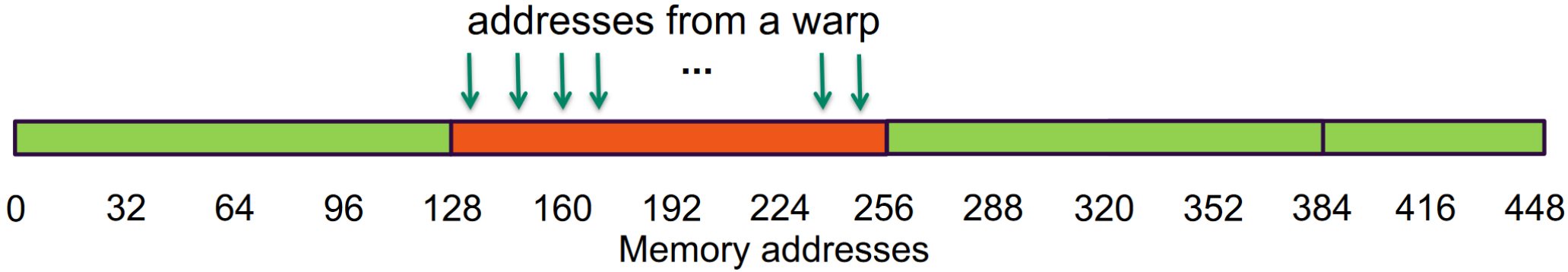


Review: Memory hierarchy

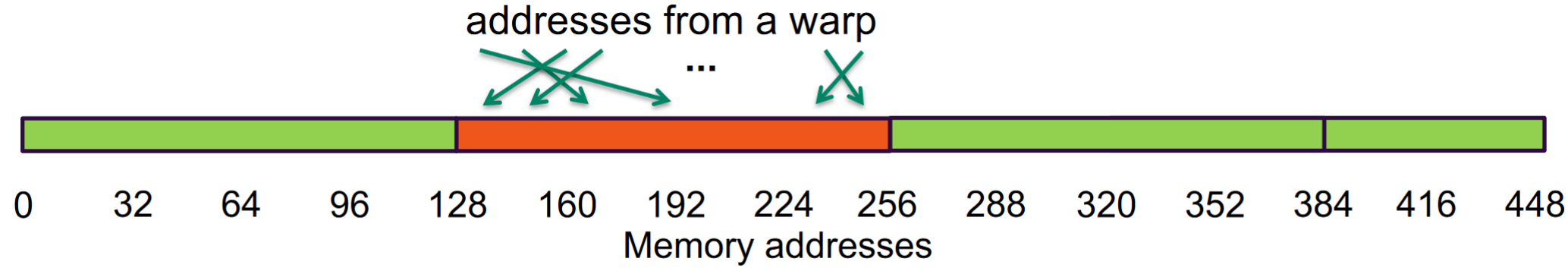


Memory Coalescing

- Imagine addresses (measured in bytes) are laid out linearly, as below. Does it matter whether threads in a warp access consecutive locations in memory? Isn't the point of "random-access memory" that any address should be accessible in approximately the same time?

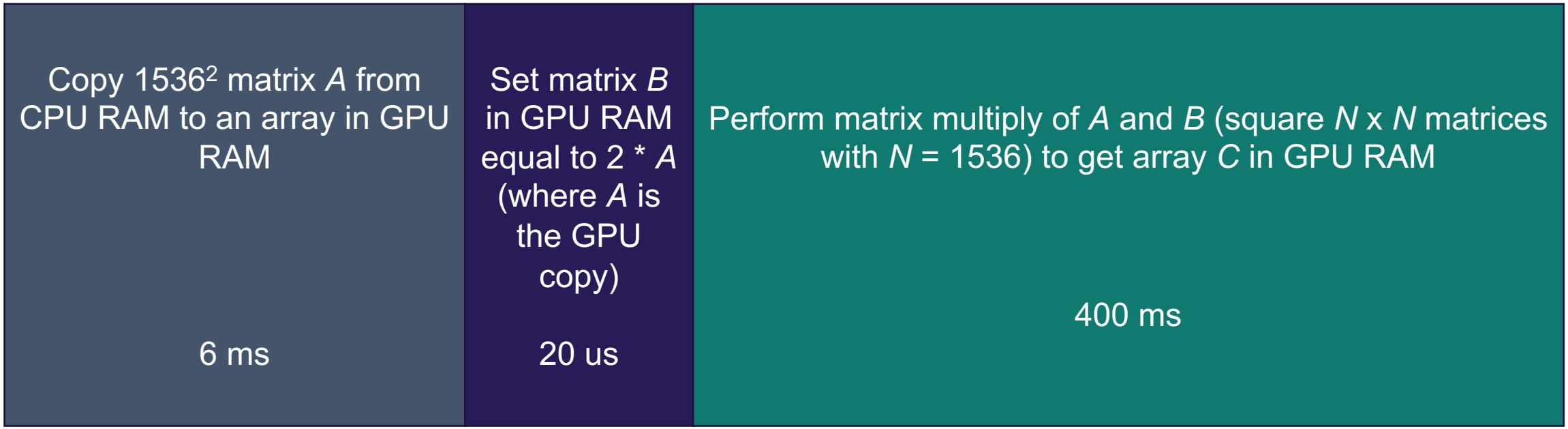


- Enter the concepts of "cache line" and memory requests vs. memory transactions



Understanding Profile Timelines

- Imagine you run your code and collect a profile like below. Do these times make sense? What further information would you need to answer that question?
- Details: PCIe Generation 3 connection between CPU and GPU, 1 TB/s GPU HBM, 10 TF/s math



(obviously, not to scale)

Profiling/analysis driven optimization

Trying to understand the hardware *too* much can be problematic too though. In particular, do not attempt to guess where your code's bottleneck is. Profile the code to find out.

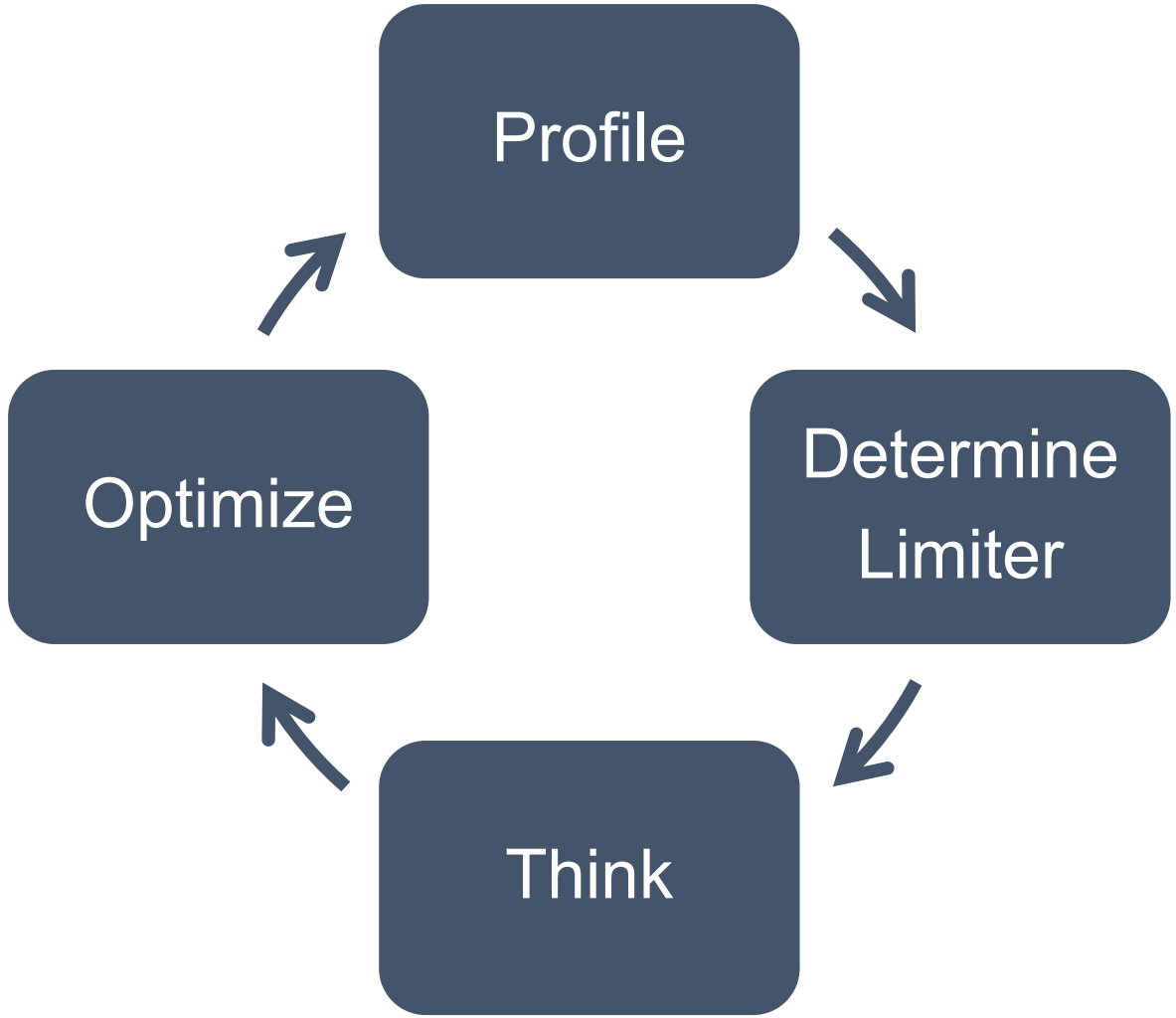
- "I heard that branch divergence is bad for GPUs."

Profile the code anyway.

- "But I read it on Stack Overflow."

Seriously, just profile the code.

- "Look at how many if statements are in my kernel! There's no way this can work well."



Honestly though, I still don't know how compilers work

- But I learned some fun things anyway by working with* users
- For example, consider this code:

```
x += y * z;
```

- What assembly code (i.e. fundamental kind of instruction) should this map to?
- FMA?

Turns out the C++ standard says something about the right-hand side needing to be evaluated first before the left-hand side, and also turns out that the compiler team are nerds who read the standard, so unlike x = x + y * z this cannot be FMA. This code might be half as fast as it could be!

**being confused by their questions and asking the NVIDIA compiler team to help me*

Abstraction layers hide a lot of fun

- Consider this code (from the Kokkos tutorials):

```
auto y = static_cast<double*>(std::malloc(N * sizeof(double)));
Kokkos::parallel_for( "y_init", N, KOKKOS_LAMBDA ( int i ) {
  y[ i ] = 1;
});
```

- And consider this code:

```
setval<<<N / 256, 256>>>(y);
// what should the body of setval look like?
__global__ void setval (double* y) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  y[ i ] = 1;
}
```

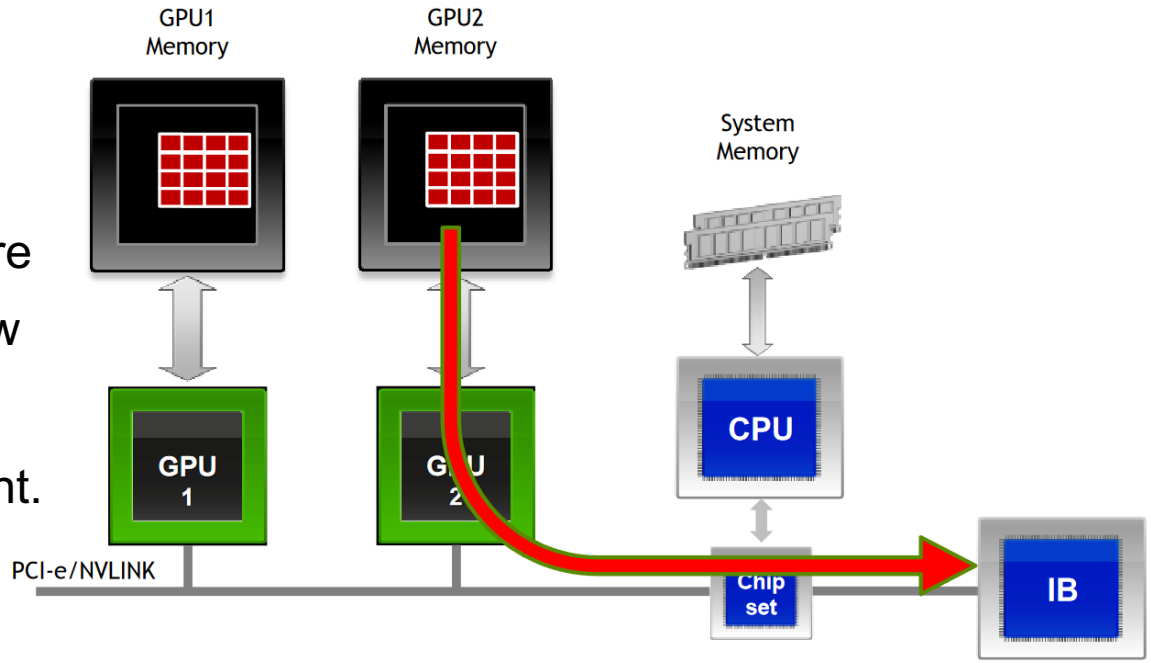
- You should be able to read the former as if it's the latter, and imagine the N threads running on the GPU

Precision is important

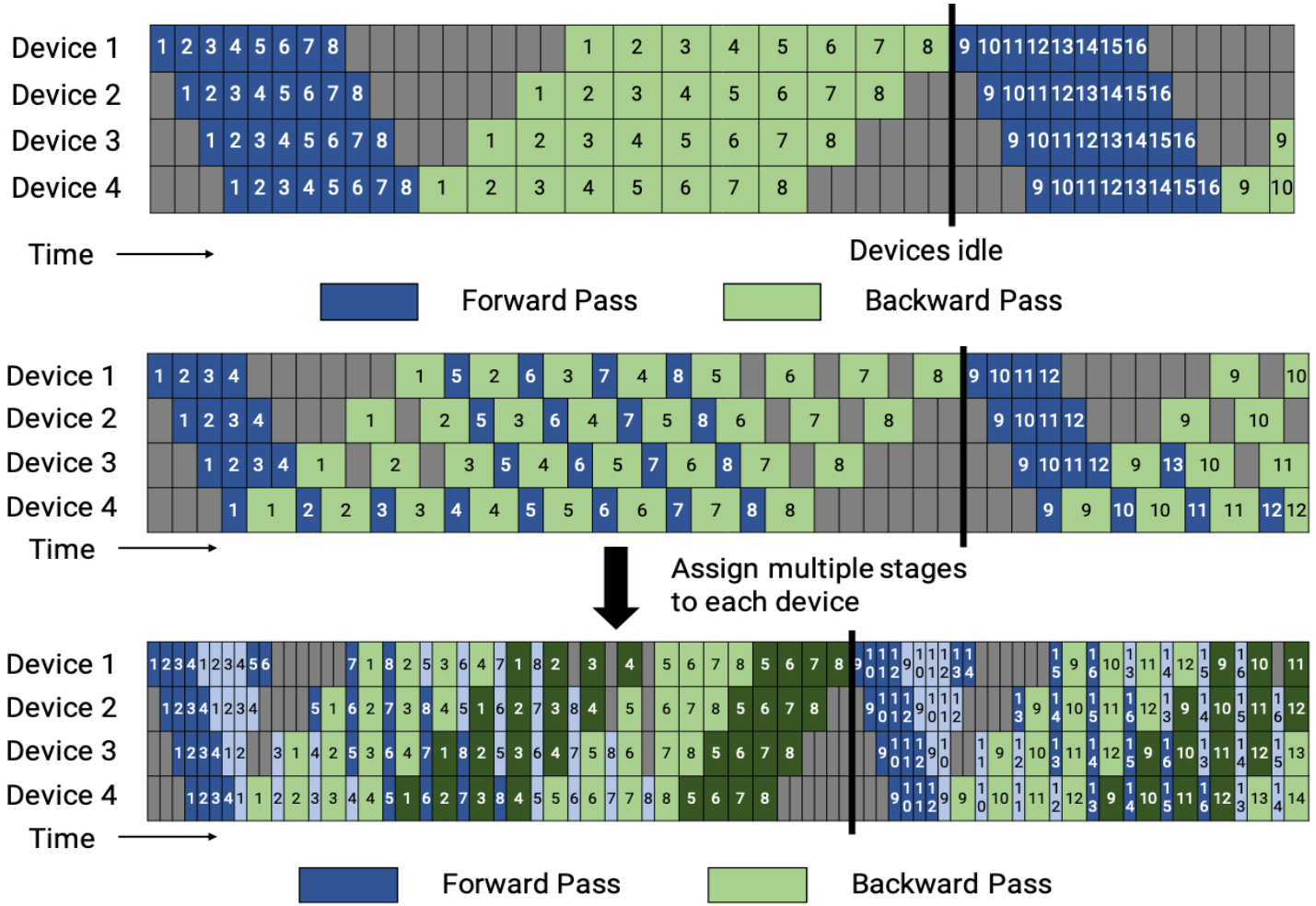
- If you can't describe something precisely, it's likely that you don't fully understand it. My most frequently encountered example of this is CUDA-aware MPI (or GPU-aware MPI).
  - This is defined as using MPI API calls on data that may reside in GPU memory

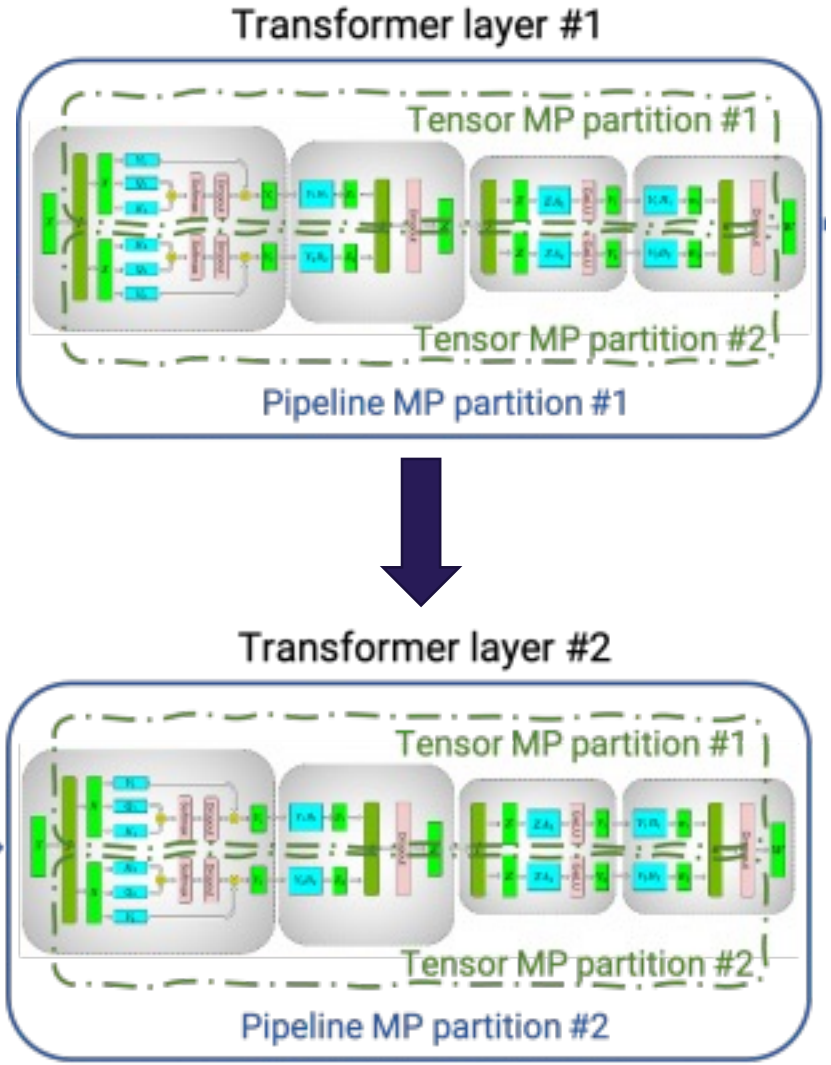
- Despite what you may read, this is NOT the same as GPUDirect (or the made-up term CUDA-direct MPI)

- GPUDirect refers to a family of technologies for GPU to GPU message passing, both within a server and across servers. CUDA-aware MPI is one example of a software technology that sits on top of this hardware technology. If you haven't spent time thinking about how this all works, you aren't likely to think critically about what is going on when things don't work like you thought.
  - CUDA-aware MPI does not require GPUDirect!



Random: large-scale deep learning is basically HPC





Closing thoughts

- Not all of you will work for a hardware vendor, but all of you have the opportunity to spend some time skilling up in how supercomputers work
  - With that said, those of you who are students/postdocs, give some thought to non-academic careers
  - Also, if you have access to vendor support, use it (I thought vendors were icky when I was a student)
  - Vendor tools will only get better if you ask forcefully and persistently – but the vendors do want to help
- Getting science done isn't always compatible with spending time learning what's going on underneath the hood, but you should seriously spend some fraction of your time on the latter
  - And what else are you going to do while your batch job is running?
- Profiling your code regularly is an effective life-hack for getting better at HPC
- You have a choice about whether you're going to stay current on new technologies, or to calcify your knowledge at what you know when you're 28 years old and never learn anything new
  - No pressure on which path you take though

Follow-up resources

- OLCF CUDA Training Series
- GTC 2021: Requests, Wavefronts, Sectors Metrics: Understanding and Optimizing Memory-Bound Kernels with Nsight Compute
- GTC 2019: CUDA Kernel Profiling Using NVIDIA Nsight Compute
- NVIDIA blog posts:
  - NVIDIA AI Platform Delivers Big Gains for Large Language Models
  - Scaling Language Model Training to a Trillion Parameters Using Megatron

DALL·E mini by craiyon.com

AI model generating images from any prompt!

Grad student uses a GPU to do science

Run