GPGPU Programming

An Introduction to the Technology and Programming Tutorial

Dave Stampf
CSC/BNL
drs@bnl.gov

August 23, 2013

Slides & Software at github.com/davestampf/GPUTalk

It's all about Performance (slashdot.org)

"I am an intermediate-level programmer who works mostly in C# NET. I have a couple of image/video processing algorithms that are highly parallelizable — running them on a GPU instead of a CPU should result in a considerable speedup (anywhere from 10x times to perhaps 30x or 40x times speedup, depending on the quality of the implementation). Now here is my question: What, currently, is the most painless way to start playing with GPU programming? Do I have to learn CUDA/OpenCL — which seems a daunting task to me — or is there a simpler way? Perhaps a Visual Programming Language or 'VPL' that lets you connect boxes/nodes and access the GPU very simply? I should mention that I am on Windows, and that the GPU computing prototypes I want to build should be able to run on Windows. Surely there must a be a 'relatively painless' way out there, with which one can begin to learn how to harness the GPU?"

and some responses:

- “GPU programming is painful. *A painless introduction doesn't capture the flavor of it*.”
- “Since the whole point of GPU programming is efficiency, don't even think about VBing it. Or Pythoning it. Or whatever layer of a shiny crap might seem superficially appealing to you. Learn OpenCL and do the job properly.”

and a another (talking about a class)...

"We were building little throw-away matrix multiply programs - for which we were given horribly inefficient and barely functional source to start with. The challenge was to make it run as fast as possible, with extra credit going to the fastest implementation. It turns out to accomplish this you basically need to understand every tier of the memory architecture of CUDA, the process by which it reads in cache lines to avoid collisions, how to optimize the read/write patterns, how the job would be split up among the GPU's (and the parameters used for the splitting), and basically every nit-picking detail of how the hardware actually runs. *This runs counter to the level of abstraction that most CS majors are used to dealing with - if we wanted to do hardware we would've gone the EE or CE route - but if you want to truly want to grok CUDA, you have to become a hardware wiz.* Otherwise you'll always be stuck wondering why you can never seem to get the level of speedup that the benchmarks suggest should be possible."

But my favorite ;-)

“Yeah, it would be like S&M without the pain . . . cute, but something essential is missing from the experience.

Heidi Klum has a TV show call "Germany's Next Top Model". She basically gets all "Ilsa, She-Wolf of the SS" on a bunch of neurotic, anorexic, pubescent girls, teaching them how a top model needs to suffer.

Heidi Klum would make a good GPU programming instructor.”

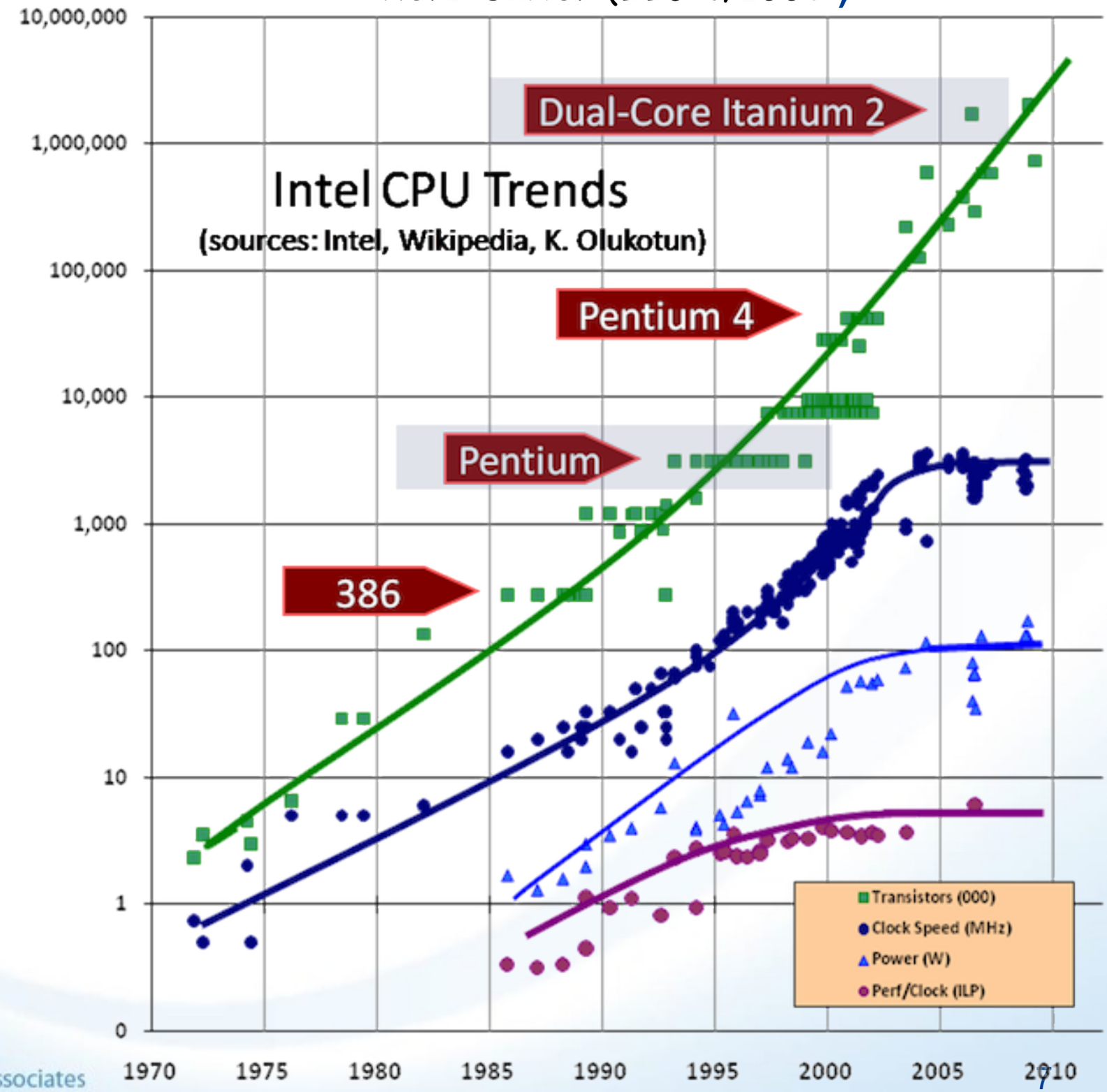
So, why the pain?

- A CPU's (single core) performance has stagnated for the past 10(!!!) years.
- GPU/Coprocessor technologies offer a disruptive opportunity to get back on the exponential track.
- While the imaging industry (Adobe, Apple, etc.) has been using these technologies for years, the scientific applications are late to the game - techniques are not fully utilized by scientific programmers.

Parallel programming *IS* mainstream programming!

Classic (by now) Performance Graph

(Source: "The Free Lunch is Over: A Fundamental Turn Toward Concurrence in Software" by Herb Sutter (DDJ 3/2005)



So, if you are a computer engineer, where do you spend your xistor budget?

CPU

- CISC
- reordering
- data & instruction cache
- hyper-threading
- several cores
- *Strives for minimizing latency*

GPU

- basically solve c = Ax + b
- many simple processors each with 1 thread
- One instruction (program) queue
- many, many cores
- *Strives for maximizing bandwidth*

Latency vs Bandwidth

- Classic example
  - Low Latency/low bandwidth
    - race car with 1 rider travels 2400 miles @ 100 mph. Net delivery of passengers = 1 every 24 hours. Pretty low latency. You call up your friend and she arrives 24 hours later.
  - high latency/high bandwidth
    - bus with 80 people traveling the same distance @ 50mph. The first person arrives 48 hours later, but she has a whole orchestra with her!
- Classical CPUs deliver low latency - they want to get the first answer to you as quickly as possible
- For the GPU - getting the top left pixel of a screen is not useful if the bottom right pixel of the screen arrives too late. You want all of the pixels as quickly as possible
- To program GPUs, YOU MUST UNDERSTAND THIS!

Your options (in increasing pain and performance):

- Ignore the hype (see graph - this is not an option)
- Buy COTS software (think Adobe) and relax
- Use libraries (cu-fftw, cublas, thrust, etc.)
  - I'll discuss one project using cufftw today
- Use directives (openACC, openMP, etc.)
  - Perhaps study more at the next tutorial...
- Use (naively) openCL, CUDA
  - You'll have this skill (and more) by the end of the morning
- Buckle down and study the hardware
  - A little today, but much, much more study is required
- Buckle down and learn parallel patterns
  - A little today, but much, much more study is required

Today's Plan

- Overview of CUDA
  - Setup
  - Hardware model
  - Programming model
- Some real code illustrating:
  - map
  - reduce (e.g., numerical integration)
  - scan (e.g., prefix sum/max/min/...)
  - Matrix Transpose (if time permits)
- Porting techniques (ongoing study)
  - fftw
- Resources

Reality Check

- Many schools (and MOOCs!) have semester courses on these topics while we have 3 hours.
  - I'm going to resist taking the conversation down a path that loses 50% of the attendees or that really requires it own 3 hours to cover properly.
  - I will be taking some simplified shortcuts aimed to maximize the topics we can cover.
  - I won't tackle the questions of hardware system design - that deserves way more than 3 hours!
- My goals:
  - You will have a better idea of what CUDA is
  - You will have a realistic idea of the work involved in developing parallel programs and porting non-parallel code to perform well
  - You will know where to get more information about hardware and software
  - You will know what areas we can explore in future tutorials
  - You can listen intelligently to the afternoon talks

Overview - setup

Overview - setup

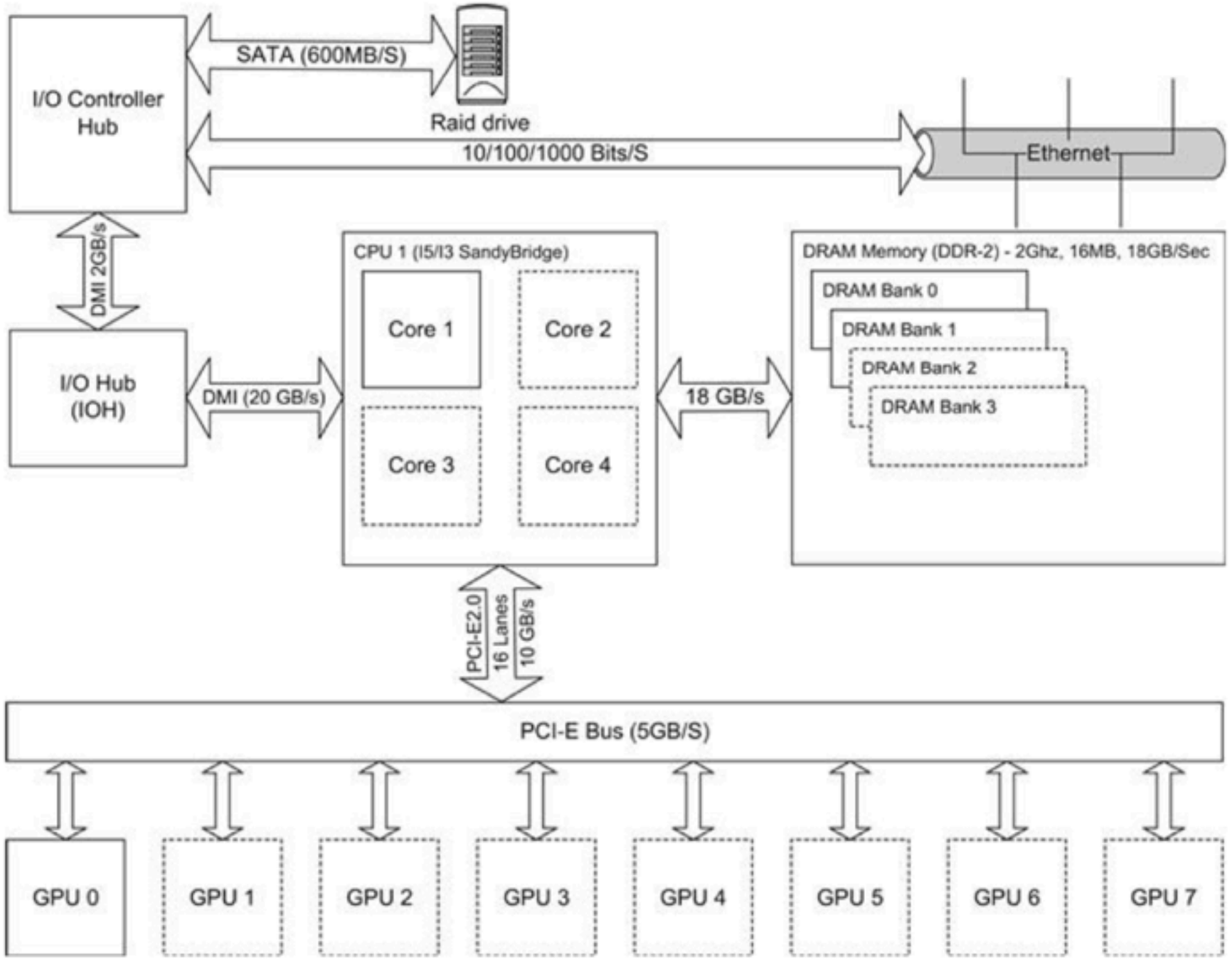
- Make sure your system supports the CUDA SDK (see web site)
- Download the SDK from nvidia.com
- Install by reading their directions (it changes so it is not worth recording here)
- Set up your PATH to include the CUDA compilers and other binaries.
- Be sure to browse the contents of
  - .../cuda/doc
  - .../cuda/samples - seriously - you can learn a lot

Overview Hardware

Overview - Hardware

- Intel systems seem to be evolving at about a 2 year cycle
- nVidia systems are evolving at the same (non-synchronized) pace.
- Since this is a tutorial, we are going to keep this at a high level - when you get access to a system, you can learn all of the relevant details.
- If you are designing a system - you need more than this tutorial!

A Typical System (Sandy Bridge)

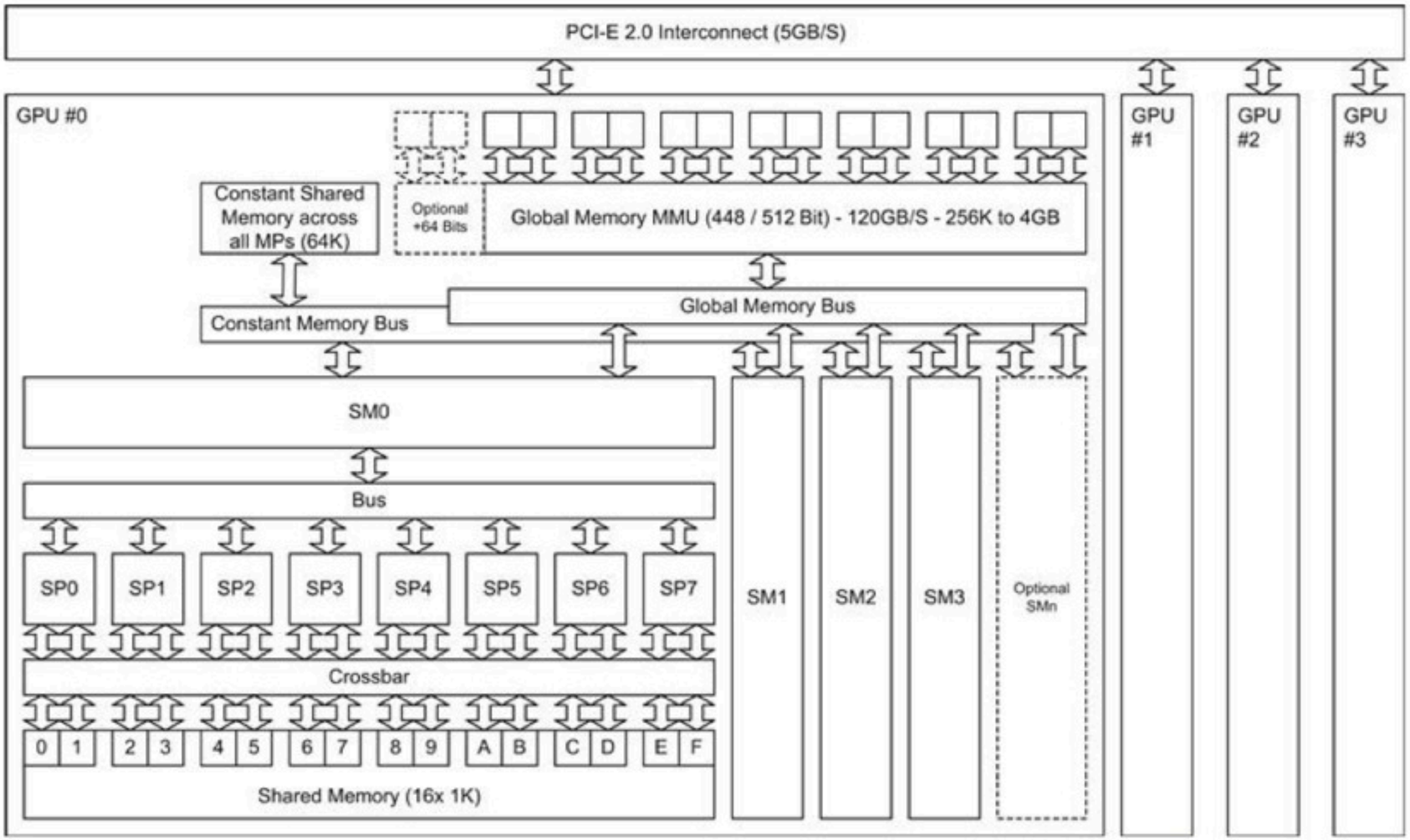


Source: CUDA Programming A Developer's Guide to Parallel Computing with GPUs by Shane Cook, Morgan Kaufmann

Two instances

- GeForce CTX480
  - 448 CUDA cores
  - Memory
    - 1G global memory
    - 320 bit interface width
    - 134GB/s
  - Cost - ~$500 from amazon.com
- Tesla C2050
  - 448 CUDA cores
  - Memory
    - 3G global
    - 384 bit interface width
    - 144 GB/s
  - Cost - $1350 from amazon.com

Block Diagram of GPU (G80/GT200)



Source: CUDA Programming A Developer's Guide to Parallel Computing with GPUs by Shane Cook, Morgan Kaufmann

Programming Interlude - Discovery-1

```
/*
 * Just how many cuda enabled devices on this machine?
 * Also, what are their properties?
 *
 * Note - EVERY cuda call returns an error value. While
 * this is vital in real code, it gets in the way of
 * tutorial code.  I'm showing it here for cudaGetDeviceCount
 * but will omit it for the rest of the tutorial.
 */

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {

        int numberOfDevices;
        cudaError_t err;

        err = cudaGetDeviceCount(&numberOfDevices);
        if (err != cudaSuccess) {
                fprintf(stderr,"fail - cudaGetDeviceCount %d\n",err);
                exit(1);
        }
        printf("Number of cuda devices = %d\n",numberOfDevices);
```

Programming Interlude - Discovery-2

```
/* the cudaDeviceProp struct is fairly large - read about it in the
   docs. */
for (int dev = 0; dev < numberOfDevices; dev++) {
        cudaDeviceProp props;
        cudaGetDeviceProperties(&props,dev);
        printf("Device # %d\n",dev);
        printf(" name = %s\n",props.name);
        printf(" version = %d.%d\n",props.major,props.minor);
        printf(" total global memory = %ld\n",props.totalGlobalMem);
        printf(" shared Memory/Block = %ld\n",props.sharedMemPerBlock);
        printf(" registers/block = %d\n",props.regsPerBlock);
        printf(" warp size = %d\n",props.warpSize);
        printf(" Max threads/block = %d\n",props.maxThreadsPerBlock);
        printf(" Max Threads Dim = %d x %d x %d
\n",props.maxThreadsDim[0],
                  props.maxThreadsDim[1],props.maxThreadsDim[2]);
        printf(" Max Grid Size = %d x %d x %d\n",props.maxGridSize[0],
                  props.maxGridSize[1],props.maxGridSize[2]);
        printf(" Multi-processor count = %d
\n",props.multiProcessorCount);
        printf(" Max Threads/multiprocessor = %d
\n",props.maxThreadsPerMultiProcessor);
    }
    return 0;
}
```

Programming Interlude - output

```
drs@gpu2:~/Talk$ ./cuda-devices
Number of cuda devices = 4
Device # 0
 name = Tesla C2050
 version = 2.0
 total global memory = 2817982464
 shared Memory/Block = 49152
 registers/block = 32768
 warp size = 32
 Max threads/block = 1024
 Max Threads Dim = 1024 x 1024 x 64
 Max Grid Size = 65535 x 65535 x 65535
 Multi-processor count = 14
 Max Threads/multiprocessor = 1536
```

Overview - Software

Programming CUDA

- In the bad old days, programming your GPU meant that you had to cast your problem as a graphics manipulation. CUDA (and openCL, etc.) permit you to treat the device as a more-or-less general purpose computer.
- Your programming of CUDA requires that you write code for both the host (e.g., the Intel CPU) and the device - the GPU.
- Functions that run on the device are called “kernels”
- Both host and device code is written in CUDA-C, a (syntactically) minor extension of C (basically a handful of additional keywords and a strange calling syntax)
- The host code does all of the setup and breakdown and “launches” kernels.
- The kernels, once launched run asynchronously
- Data xfers are synchronous by default

The Basic Cuda Dance (host's view)

1. Allocate space on the device
2. Copy data from the host to the device
3. Launch one or more kernels
4. Copy data from the device back to the host
5. Free space on the device

The Basic Cuda Dance (Kernel view)

- A kernel's code describes what one thread does (think the "run" method of the Thread class in Java)
- Each thread that is created when a kernel is launched has a (unique) number (zero based index) and each thread magically knows what its number is.
- Frequently, when a computation produces an array of data as its result, each thread will be used to compute just 1 element of the result.
- So, basically, you replace an external for loop with a ton of threads

The code structure resembles simply unrolling loops

Non-CUDA

```
for (int i = 0; i < n; i++)
{
  /compute output element i
  result[i] = ...
}
```

CUDA

```
get threadId
result[threadId] = ...
```

```
get threadId
result[threadId] = ...
```

```
get threadId
result[threadId] = ...
```

...

```
get threadId
result[threadId] = ...
```

But you only have to write 1 of these!

Overview - Programming

- The “native” way to program CUDA devices is by using a variant of C.
  - These files typically have the extension “.cu”
- These files are compiled by the “nvcc” program that
  - picks out the CUDA kernels and compiles to “PTX” code (the machine code of the GPU)
  - passes the host code onto the standard C compiler for your system.

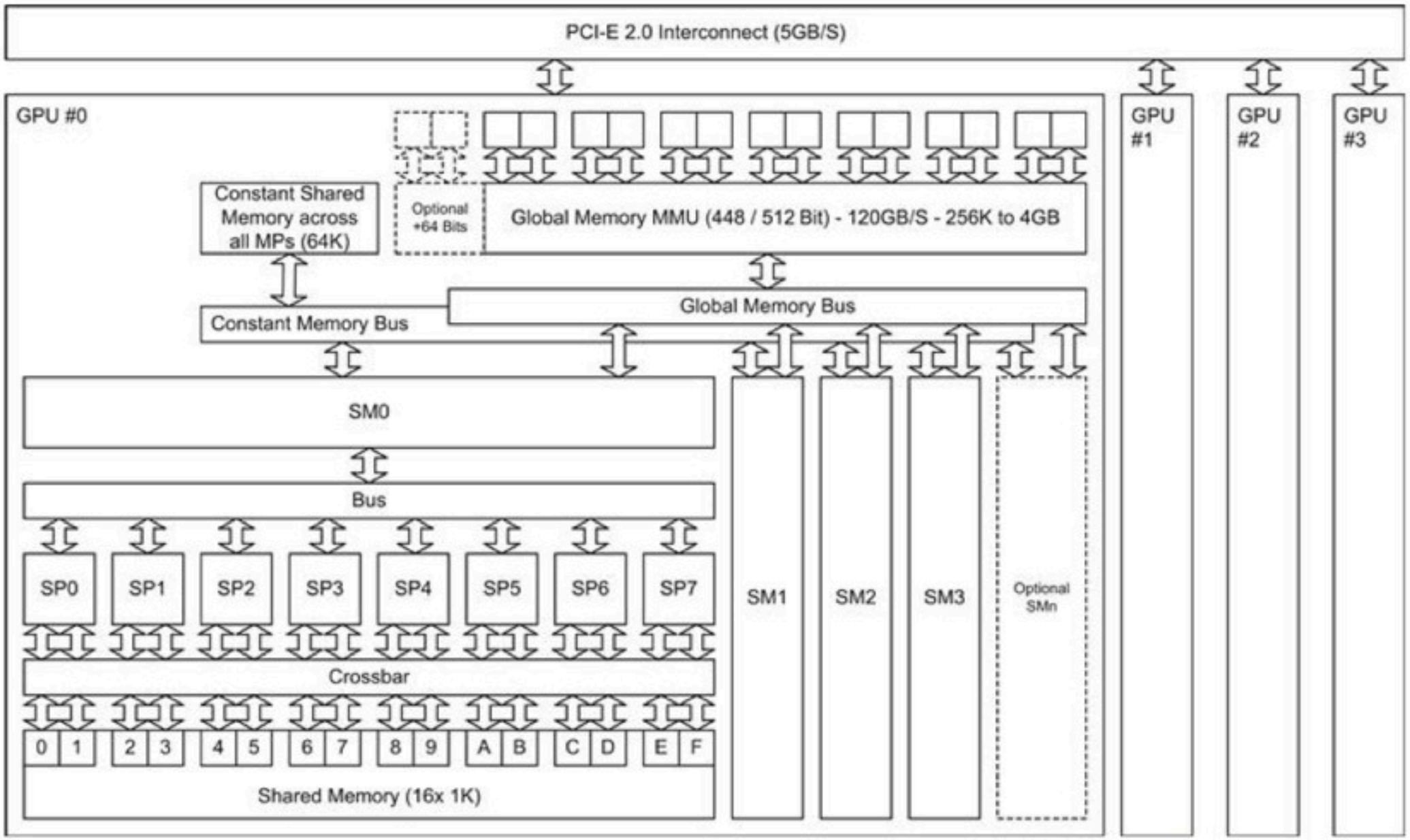
Overview - Programming Model First Pass - Threads

- A thread executes the instructions in your program.
- In CUDA, threads are cheap and are allocated by the 1000's if not 1,000,000's.
- Threads have a small amount of local (private) memory (c not m)
- If your program computes an array as output, you typically have 1 thread compute 1 element of the array.
- You need to think - "I have to write a program that only computes 1 element of the answer.

Overview - Programming Model First Pass - Blocks

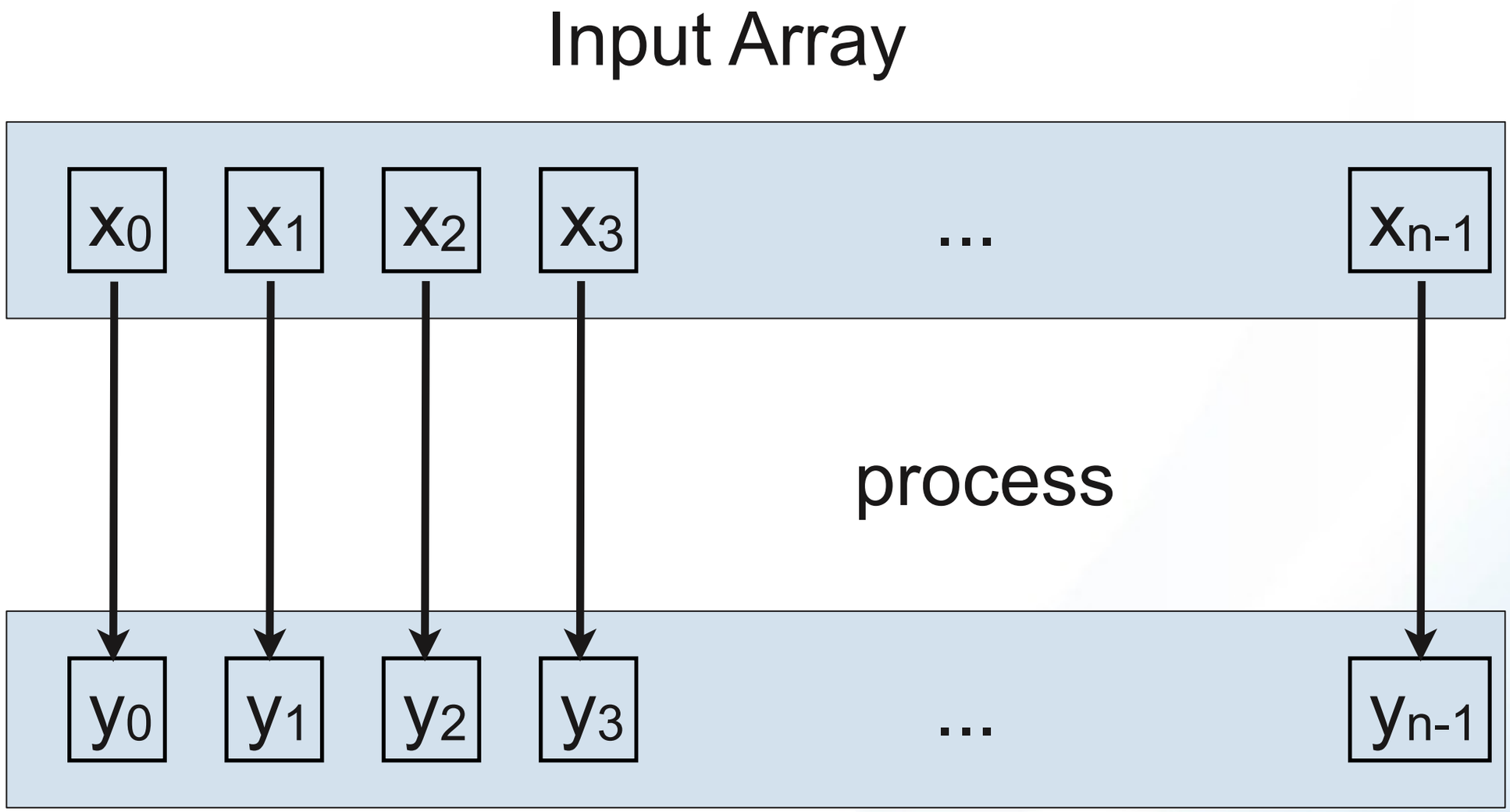
- A Block is a bunch of threads (up to 1024 on modern devices)
- When you launch a kernel, you create 1 or more blocks.
  - It is your choice for the number of blocks and the number of threads/block - you choose to fit the problem & for performance
- The block is the unit of scheduling for the GPU
  - It is one reason why the GPU is so scalable
  - They can run in any order in parallel or sequentially
  - So, on a small GPU, you might run 1 block after another while on a larger GPU you might run a dozen or more in parallel.
- Blocks cannot communicate with each other directly (only indirectly through global memory)

Block Diagram of GPU (G80/GT200)



Source: CUDA Programming A Developer's Guide to Parallel Computing with GPUs by Shane Cook, Morgan Kaufmann

Finally, A Data Access Pattern - the Map



The Map in code

Sequential

```
for (int i = 0; i < n; i++)
{
  y[i] = f(x[i]);
}
```

Parallel

```
y[threadId] = f(x[threadId])
```

Quick quiz: If you have enough threads to cover the array, what is the “O” speed of these two algorithms?

Code Interlude - sequential map

```
float square(float x) {
        return x*x;
}

int main(int argc, char** argv) {

        if (argc < 2) {
                printf("Usage: %s #-of-floats\n",argv[0]);
                exit(1);
        }
        int size = atoi(argv[1]);
        printf("size = %d\n",size);

        float *h_in;
        float *h_out;

        h_in = (float*) malloc(size*sizeof(float));
        h_out =(float*) malloc(size*sizeof(float));

        for (int i = 0; i < size; i++) {
                h_in[i] = i;
        }

        startClock("compute");
        nonCudaMap(h_out,h_in,size);
        stopClock("compute");

        for (int i = 0; i < size; i++) {
                printf("%f -> %f\n",h_in[i],h_out[i]);
        }

        free(h_in);
        free(h_out);

        printClock("compute");
}

void nonCudaMap(float* out, float* in, int size) {
        for (int i = 0; i < size; i++) {
                out[i] = square(in[i]);
        }
}
```

Code Interlude - CUDA map (host)

```
int main(int argc, char** argv) {

        if (argc < 2) {
                printf("Usage: %s #-of-floats\n",argv[0]);
                exit(1);
        }
        int size = atoi(argv[1]);
        printf("size = %d\n",size);

                  cudaDeviceProp props;
                  cudaGetDeviceProperties(&props,0);
                  if (size > props.maxThreadsPerBlock) {
                                    fprintf(stderr,"Max size for the small model is %d\n",
                                                      props.maxThreadsPerBlock);
                                    exit(1);
                  }

                  void *d_in;       // device data
                  void *d_out;
                  float *h_in;      // host data
                  float *h_out;

                  cudaMalloc(&d_in,size*sizeof(float));
                  cudaMalloc(&d_out,size*sizeof(float));
                  h_in = (float*) malloc(size*sizeof(float));
                  h_out =(float*) malloc(size*sizeof(float));

                  for (int i = 0; i < size; i++) {
                                    h_in[i] = i;
                  }

                  startClock("copy data to device");
                  cudaMemcpy(d_in,h_in,size*sizeof(float),cudaMemcpyHostToDevice);
                  stopClock("copy data to device");

                  startClock("compute");

                  // use one block and size threads

                  map<<<1,size>>>((float*) d_out,(float*) d_in,size);
                  cudaThreadSynchronize();             // forces wait for map to complete

                  stopClock("compute");

                  startClock("copy data to host");
                  cudaMemcpy(h_out,d_out,size*sizeof(float),cudaMemcpyDeviceToHost);
                  stopClock("copy data to host");

                  for (int i = 0; i < size; i++) {
                                    printf("%f -> %f\n",h_in[i],h_out[i]);
                  }

                  free(h_in);
                  free(h_out);
                  cudaFree(d_in);
                  cudaFree(d_out);

                  printClock("copy data to device");
                  printClock("compute");
                  printClock("copy data to host");
}
```

Code Interlude - CUDA map (device)

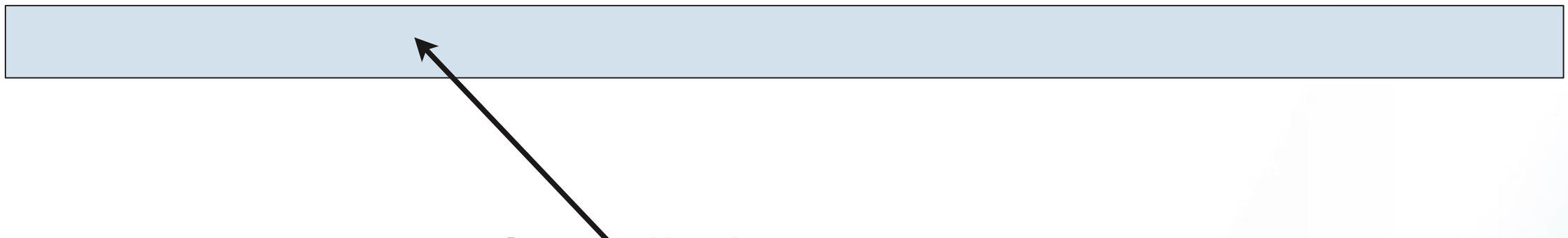
```
/*
 * squaring map kernel that runs in 1 block
 */

/*
 * runs on and callable from the device
 */

__device__ float square(float x) {
        return x*x;
}

/*
 * runs on device, callable from anywhere
 */

__global__ void map(float* out, float* in, int size) {
        int index = threadIdx.x;
        if (index >= size) return;
        out[index] = square(in[index]);
}
```

Code Interlude - Timings for Map

Code Interlude - Timings for Map

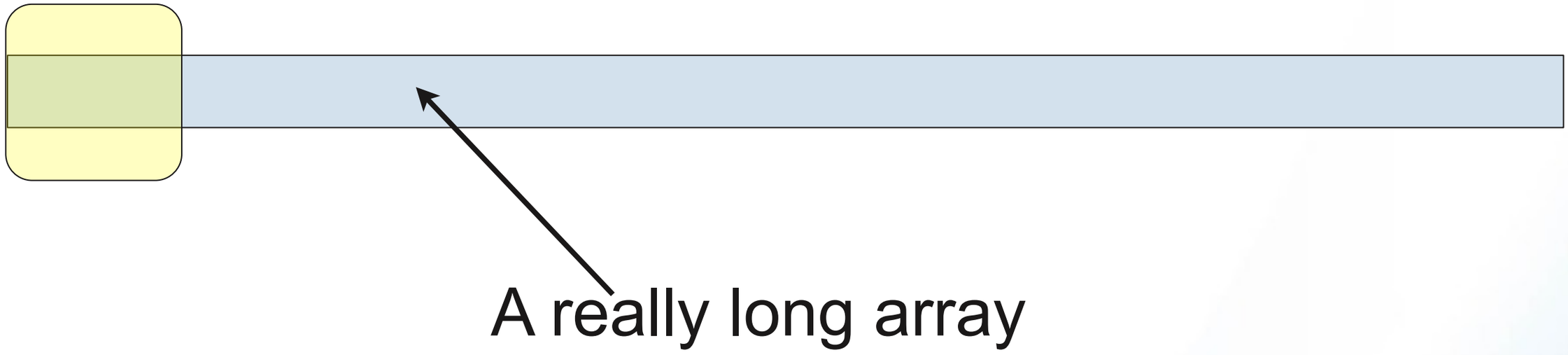
Size of array	Sequential Time in micro seconds	CUDA Data Copy Time in micro seconds	Cuda Compute Time in micro seconds
256	4	42	37
512	7	42	37
1024	11	44	38

So, How to deal with arrays > 1024?



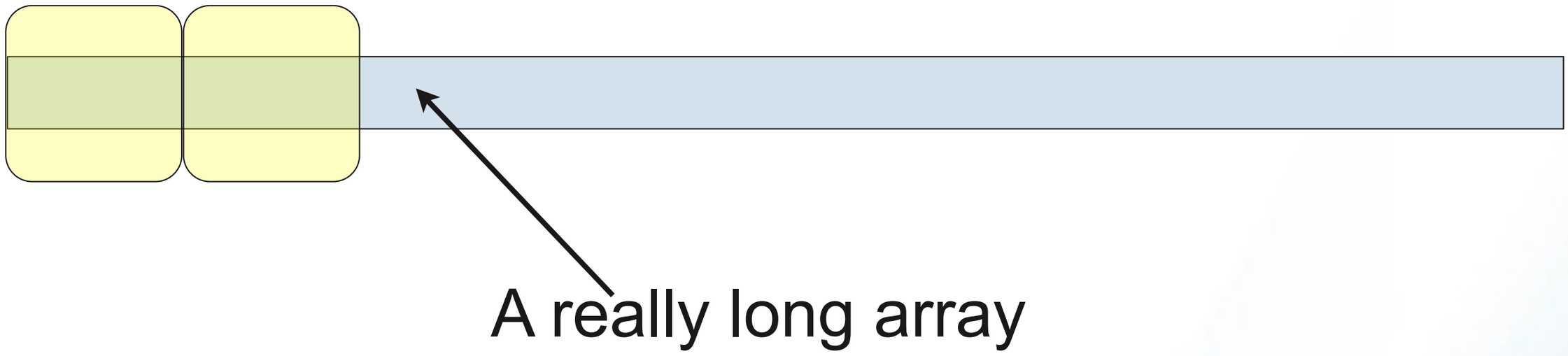
Then, (somehow) each kernel knows which block it belongs to and which thread within each block, so it can easily compute it's index into the array.

So, How to deal with arrays > 1024?



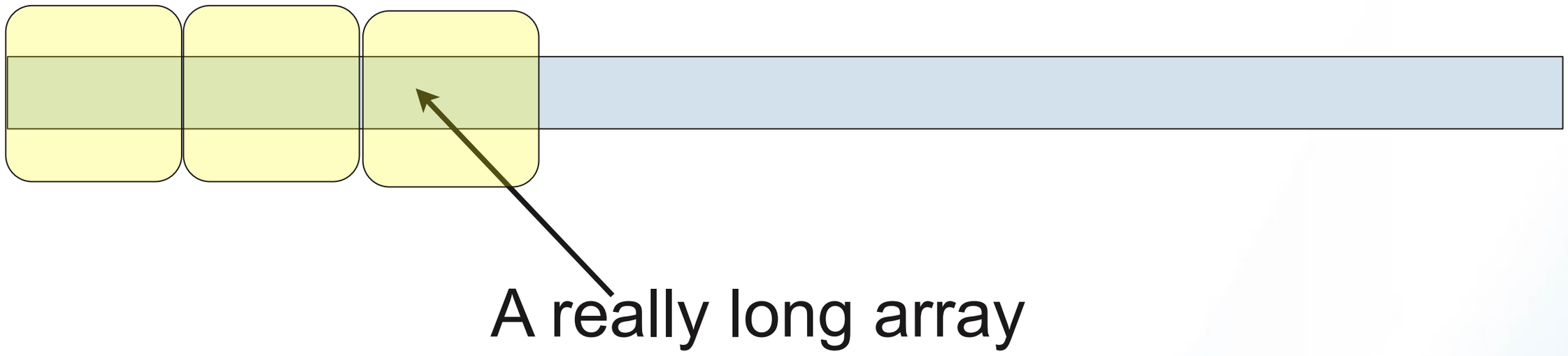
Then, (somehow) each kernel knows which block it belongs to and which thread within each block, so it can easily compute it’s index into the array.

So, How to deal with arrays > 1024?



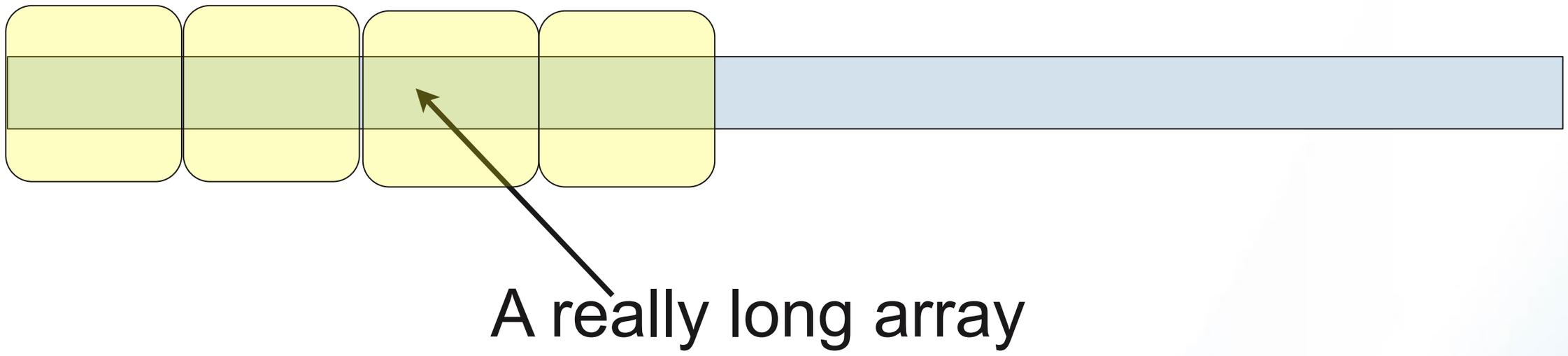
Then, (somehow) each kernel knows which block it belongs to and which thread within each block, so it can easily compute it’s index into the array.

So, How to deal with arrays > 1024?



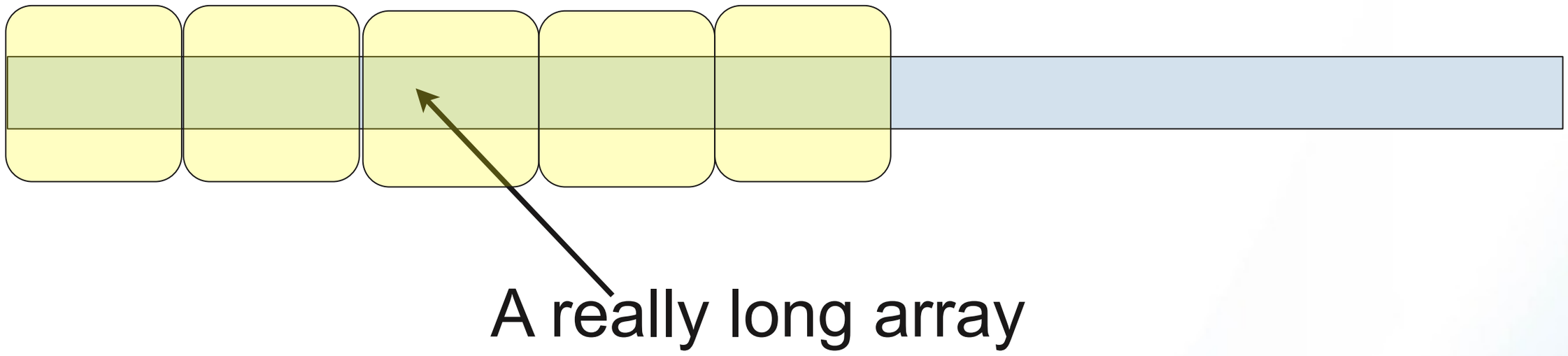
Then, (somehow) each kernel knows which block it belongs to and which thread within each block, so it can easily compute it’s index into the array.

So, How to deal with arrays > 1024?



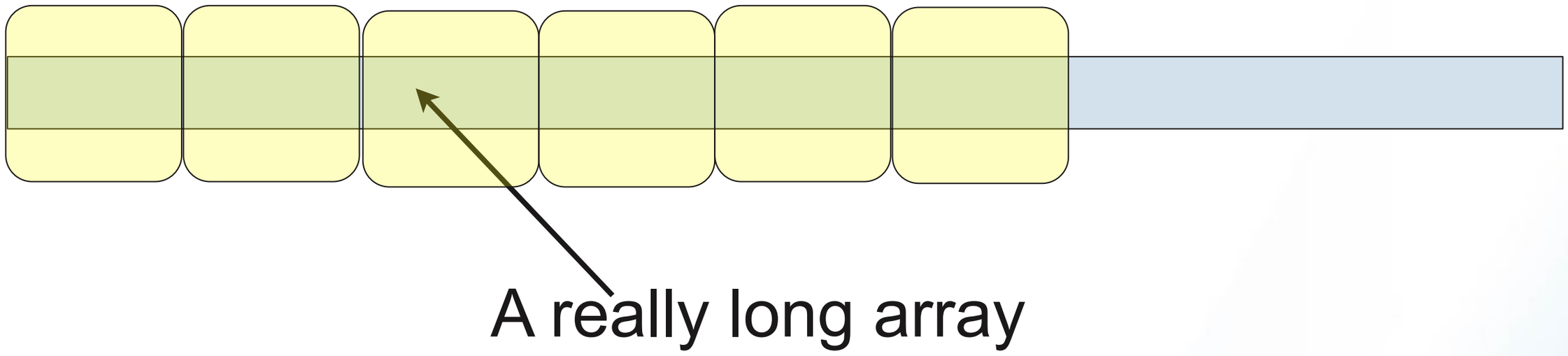
Then, (somehow) each kernel knows which block it belongs to and which thread within each block, so it can easily compute it’s index into the array.

So, How to deal with arrays > 1024?



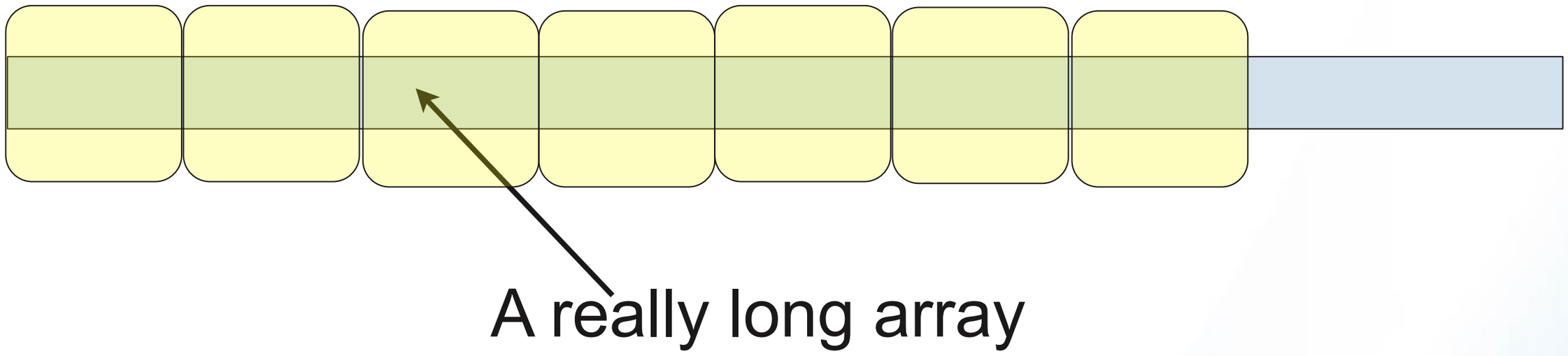
Then, (somehow) each kernel knows which block it belongs to and which thread within each block, so it can easily compute it’s index into the array.

So, How to deal with arrays > 1024?



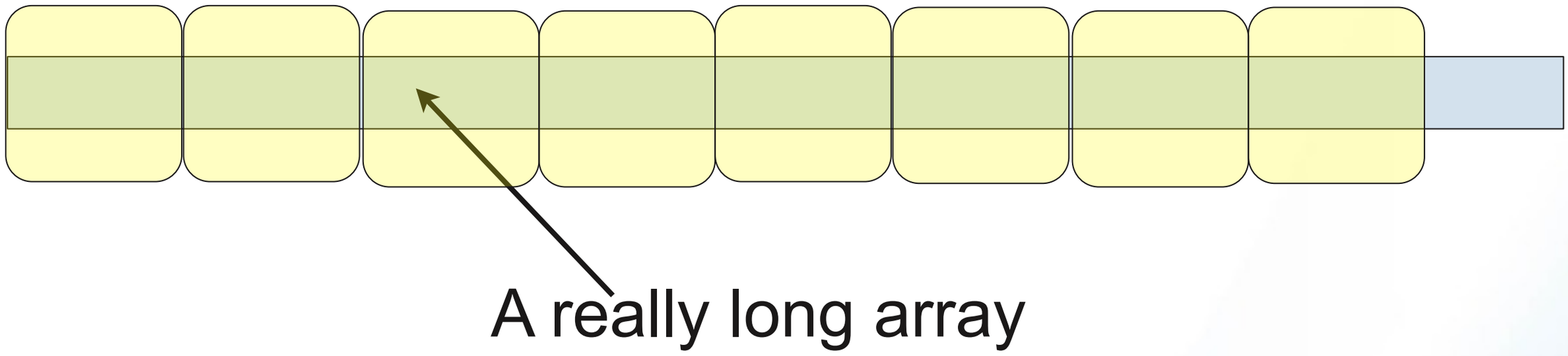
Then, (somehow) each kernel knows which block it belongs to and which thread within each block, so it can easily compute it’s index into the array.

So, How to deal with arrays > 1024?



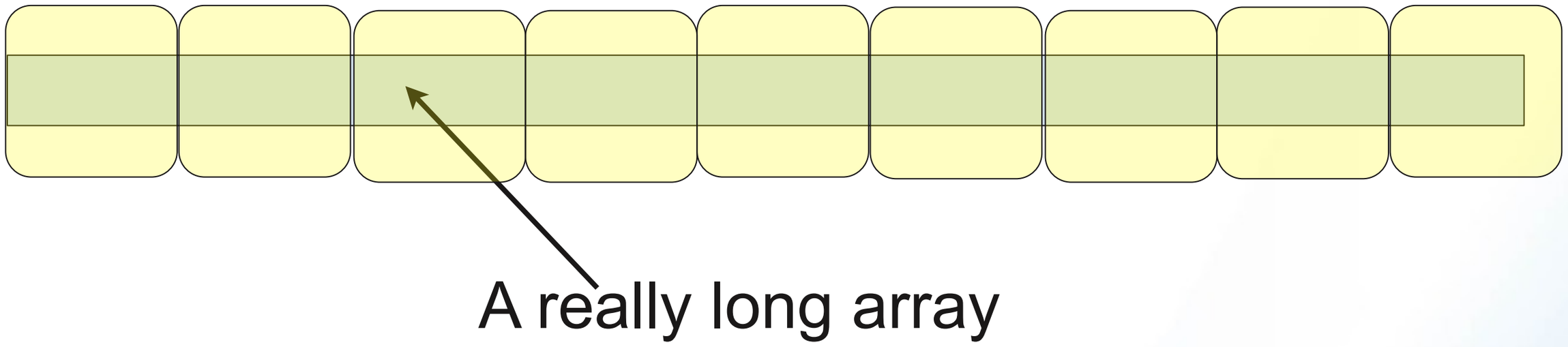
Then, (somehow) each kernel knows which block it belongs to and which thread within each block, so it can easily compute it’s index into the array.

So, How to deal with arrays > 1024?



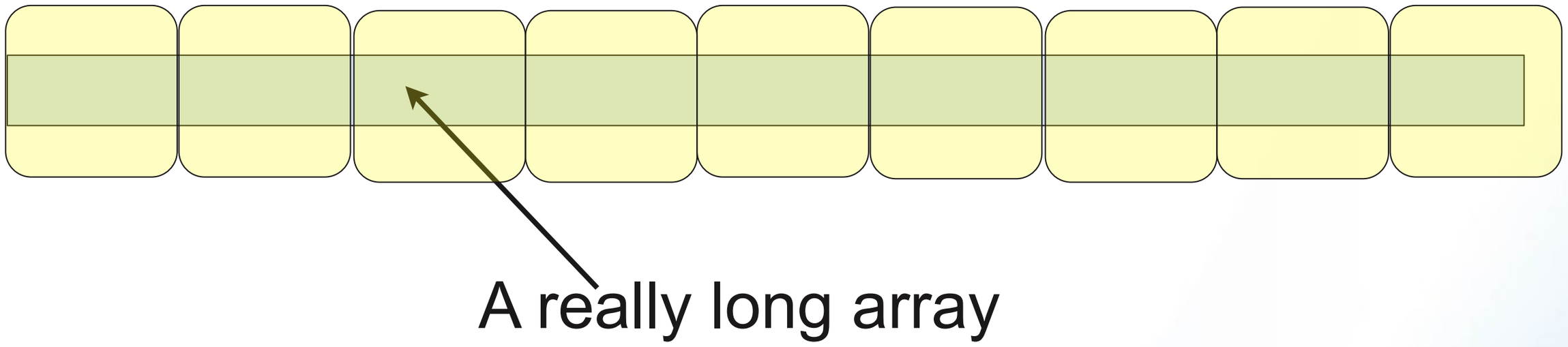
Then, (somehow) each kernel knows which block it belongs to and which thread within each block, so it can easily compute it’s index into the array.

So, How to deal with arrays > 1024?



Then, (somehow) each kernel knows which block it belongs to and which thread within each block, so it can easily compute it’s index into the array.

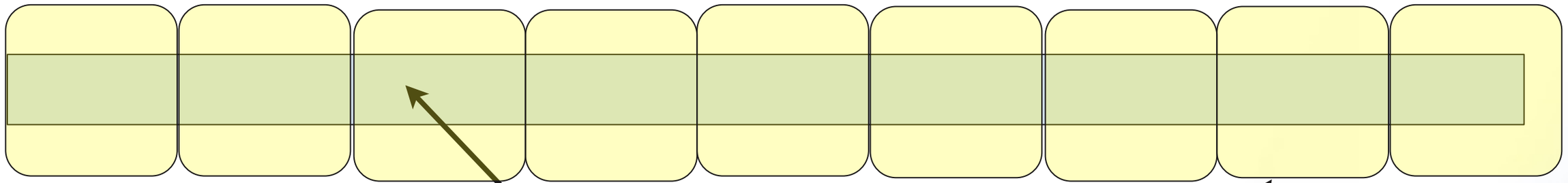
So, How to deal with arrays > 1024?



Blocks, each with 1024 or fewer threads...

Then, (somehow) each kernel knows which block it belongs to and which thread within each block, so it can easily compute it’s index into the array.

So, How to deal with arrays > 1024?

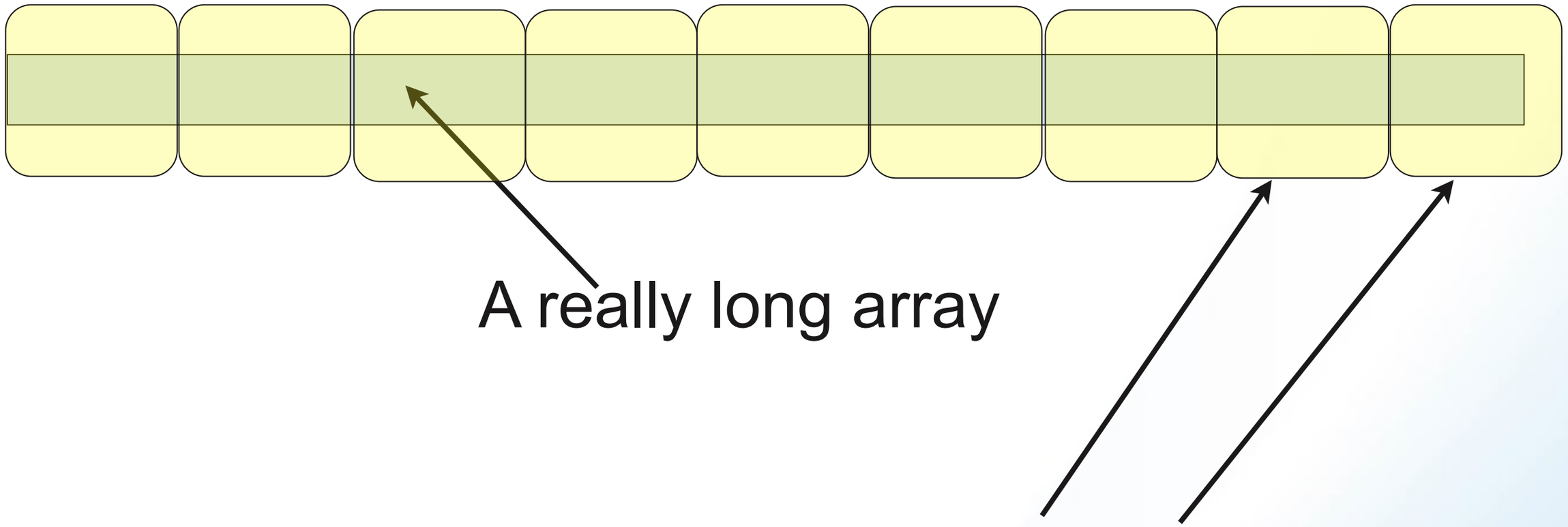


A really long array

Blocks, each with 1024 or fewer threads...

Then, (somehow) each kernel knows which block it belongs to and which thread within each block, so it can easily compute it's index into the array.

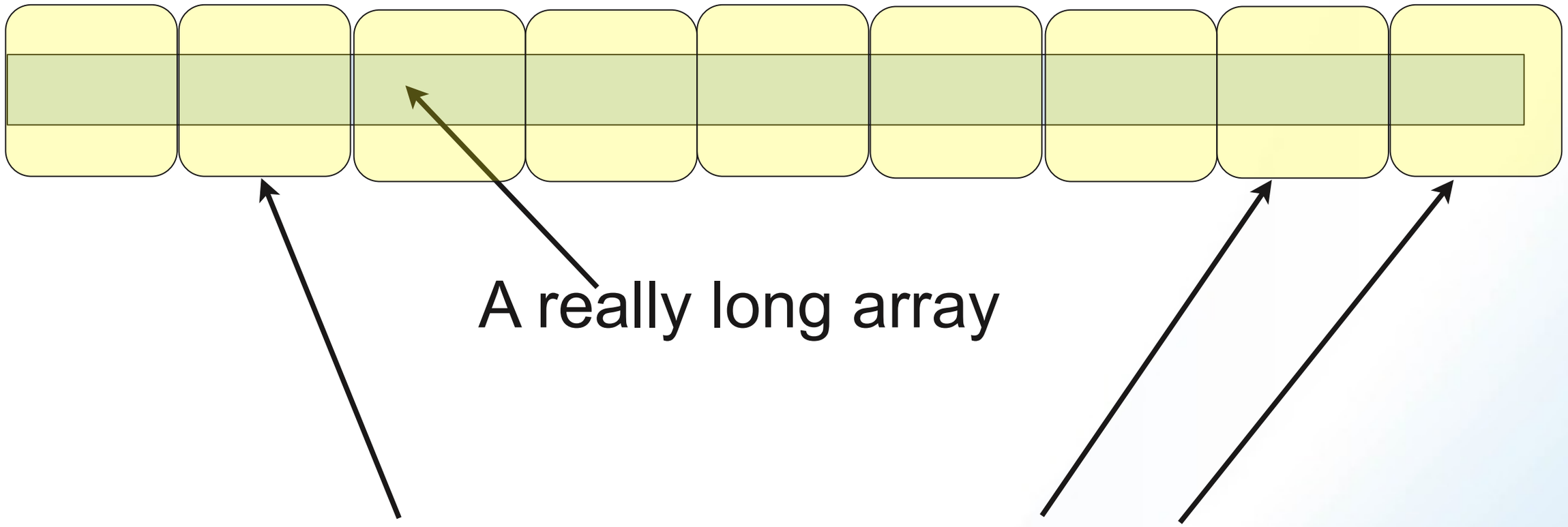
So, How to deal with arrays > 1024?



Blocks, each with 1024 or fewer threads...

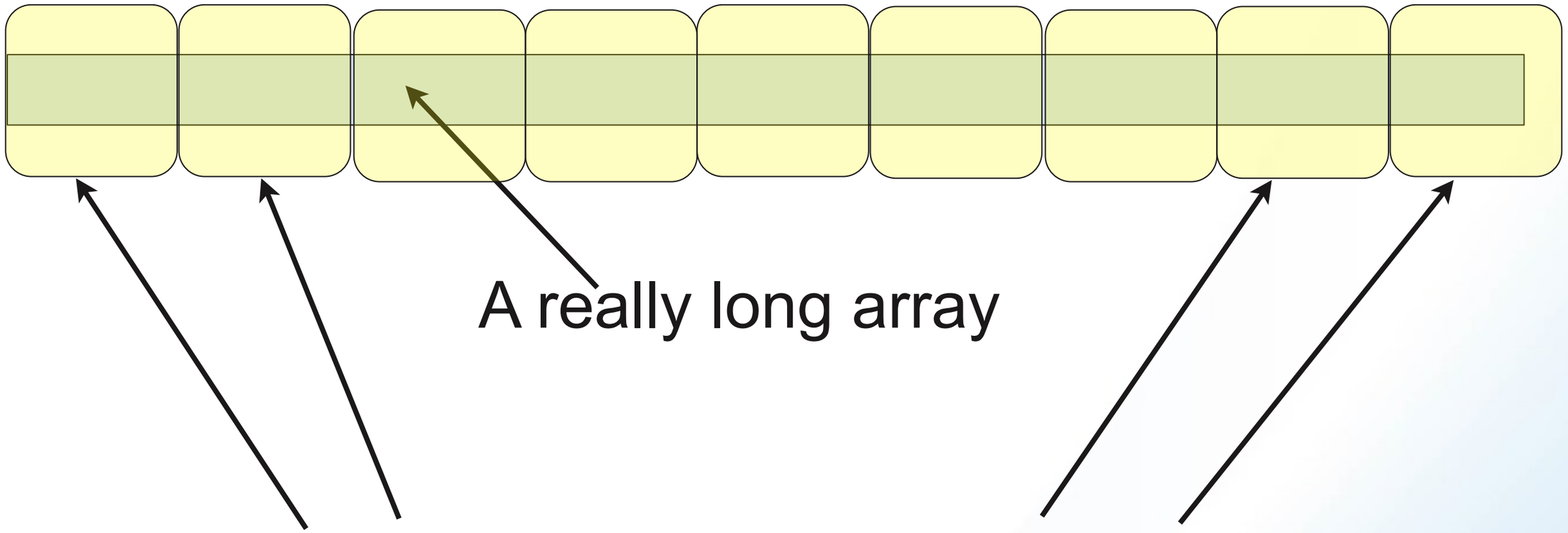
Then, (somehow) each kernel knows which block it belongs to and which thread within each block, so it can easily compute it’s index into the array.

So, How to deal with arrays > 1024?



Then, (somehow) each kernel knows which block it belongs to and which thread within each block, so it can easily compute it's index into the array.

So, How to deal with arrays > 1024?



Then, (somehow) each kernel knows which block it belongs to and which thread within each block, so it can easily compute it's index into the array.

Large Map Details:

Host Code

```
// use max threads/block and the required # of blocks

        int numBlocks = ceil(1.0*size/props.maxThreadsPerBlock);
        map<<<numBlocks,props.maxThreadsPerBlock>>>((float*)
d_out,(float*) d_in,size);
```

Kernel Code

```
__global__ void map(float* out, float* in, int size) {
        int index = blockDim.x*blockIdx.x + threadIdx.x;
        if (index >= size) return;
        out[index] = square(in[index]);
}
```

Code Interlude - Timings for Map

Size of array	Sequential Time in micro seconds	CUDA Data Copy Time in micro seconds	Cuda Compute Time in micro seconds
256	4	42	37
512	7	42	37
1024	11	44	38
2048	20	49	41
4096	41	57	41
8192	77	78	41
16384	157	111	42
65536	612	225	62

Code Interlude - Timings for Map

Size of array	Sequential Time in micro seconds	CUDA Data Copy Time in micro seconds	Cuda Compute Time in micro seconds
256	4	42	37
512	7	42	37
1024	11	44	38
2048	20	49	41
4096	41	57	41
8192	77	78	41
16384	157	111	42
65536	612	225	62

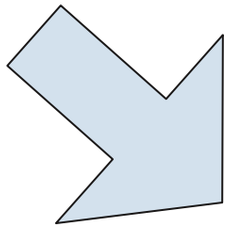
Quick quiz: Why did the compute time jump up for the last trial?

Porting Strategies

- There are 3 main issues when porting code
  - Does it improve the performance?
  - Does it give the same (hopefully correct) answer?
  - You probably do NOT want to irreparably modify the program to run on a GPU so that it cannot run on a CPU any longer
- You should measure performance by profiling
- This leads to 4 types of runs
  - cpu (production)
  - cpu with profiling
  - gpu (production)
  - gpu with profiling
  - cpu and gpu with coherency check
- I'll assume that profiling is external to the program (compiler switch)

Modifying the program

```
{
  cpu only code
}
```



```
if (cpuOnly) {
  run cpu code
} else if (gpuOnly) {
   run gpu code
} else { // both
  capture and copy input state
  run cpu code on original input state
  run gpu code on copied input state producing separate output
  state
  compare two output states for coherency and report discrepancies
}
```

Porting Issues

- Wildly differing answers probably means that you have messed up with synchronization. (Especially if the results and differently different each time your run.) GPUs are NOT flakey!
- Small differences might be explained by the differing ways CPU and GPUs do arithmetic, especially the FMA (fused multiply add) instruction of the GPU where it computes round(a*b+c) rather than round(round(a*b) + c).
  - For modern GPUs, try nvcc --fmad=false to see if it helps with coherency

Memory Allocation on the Device

- C
  - void* d_data;
    cudaMalloc(& d_data, numberOfBytes);
    - This allocates the requested numberOfBytes on the device in global memory returning an error if it fails. The value returned into d_data is opaque.
    - DO NOT dereference d_data on the host! I prefer using void* so as to have any de-referencing fail.
    - It is a good idea to flag device data with d_
- Fortran
  - float, device : d_data[1000]

Memory Deallocation

- C
  - void* d_data;
    cudaFree(d_data);
    - returns error if failure.
- Fortran
  - automatic

Data Movement

- C
  - void* d_data;
    float* h_data;
    cudaMemcpy(d_data,h_data,numberOfBytes,
    cudaMemcpyHostToDevice);
    //...

    cudaMemcpy(h_data,d_data,numberOfBytes,
    cudaMemcpyDeviceToHost);
  - Again, you should always check for errors
  - You can also copy host to host and device to device
- Fortran
  - d_data = h_data
    ...
    h_data = d_data

Device Selection

- cudaSetDevice(deviceNum);
  - All further calls to cudaMalloc, cudaFree, cudaMemcpy, etc. will be directed toward that device.
  - This could be a major win for you if you have multiple independent computations.

Terminology

- There are a number of terms that are used throughout CUDA that have to be assimilated into your way of thinking.
  - Kernel
  - Thread
  - Block
  - Grid
  - Launch
- Some like to think top down, some bottom up - we will do both

Kernel

- A Kernel is a bit of code (function) that describes what 1 thread does.
- A Kernel has a few “magic” variables that we will discuss in a few slides.
- The parameters passed as arguments to a kernel are either simple data types (int, float, etc.) or pointers to global memory areas that had been allocated by the host program.
- A Kernel has a limited amount of local (register) memory.

Thread

- A thread runs the code in a Kernel
- Local data within a Kernel is private to each thread.
- All threads share the global memory (and so have to be careful to cooperate)
- All threads in a block share the “shared” memory (and so have to be careful to cooperate).
- Threads do not have access to other blocks shared memory.

Blocks

- A Block “has-a” bunch of threads
  - for current GPUs, the maximum number of threads varies from 512 to 2048.
  - Sometimes, it is useful to the *programmer* to view the threads linearly, as a 2-d array of threads (think about working on an array or image) or as a 3-d array of threads.
- A Block is the unit of scheduling for the GPU
  - That is, a Block is given to a SM for processing. Its code runs until complete. (An SM can run more than 1 block at a time.)
- A Block has some memory (typically in the “k” range) that it shares among the threads.

Grid

- When you launch a Kernel, you specify how many blocks to create and how many threads are in each block.
- Sometimes, it suits the *programmer* to view these blocks as a linear collection, or as a 2-d array (think images) or even as a 3-d array.
- Every block knows its position within the grid

And the Launch

- kernelName<<<grid,block,sizeOfSharedData>>>(params);
  - where grid is either an int (1-D grid of blocks) or a dim3 type
    - e.g. dim3 grid(10,10,100)
  - block is either an int (1-D array of threads) or a dim3 type
    - e.g. dim3 block(16,16,2)
  - and the 3rd (optional) parameter in the <<<>>> is the number of bytes per block that will be shared.

An finally, back to the thread

- Each thread (which runs in a kernel) has some magic data in the form of structs.
- gridDim.x, gridDim.y, gridDim.z
  - Blocks are arranged in a 3-d Grid with the above dimensions
  - Every thread sees the exact same values
- blockIdx.x, blockIdx.y, blockIdx.z
  - For each thread, this is the particular block that the thread belongs to
- blockDim.x, blockDim.y, blockDim.z
  - Threads in every block are arranged in a 3-d array with the above dimensions
  - Every thread sees the exact same values
- threadIdx.x, threadIdx.y, threadIdx.z
  - For each thread, this is the position of the thread within the block

Types of memory

- Global
  - Largest in size (3G)
  - Slowest in access (100's of clock cycles)
  - Widest memory width (384 bits/48 bytes/12 floats)
  - accessible to all threads
- Shared
  - Rather tight (48K)
  - Much faster than Global
  - Segmented and connected to the processors by a crossbar switch (need to be aware of contention)
- Register
  - Very tight (32K)
  - Single cycle access
  - Private to thread

Another Data Pattern - Reduction

- The simplest example of reduction is to compute the sum of the elements in an array. (e.g., numerical integration). The idea is that all of the elements of the array need to be read and that 1 number is created.
- While I'll use addition in the example, it will work with any associative binary operator (+, *, max, min, etc.)
- The serial version is 0(n)

```
total = 0;
for (int i = 0; i<n; i++) {
  total += data[i];
}
```

Parallel version

- The parallel version makes use of the associative law that (((a+b) + c) + d) is the same as (a+b) + (c + d).
  - The first expression is extremely sequential
  - The second is much more parallel. One can compute a+b while some other processor computes (c+d).
- In our example, we sum up 2**10 items. For the first pass, 1024 blocks of 1024 threads sums up each block and places the answer in an intermediate area. Then we do it again to the intermediate area.

Serial Version:

```
int main(int argc, char** argv) {

            if (argc < 2) {
                        printf("Usage: %s #-of-floats\n",argv[0]);
                        exit(1);
            }
            int size = atoi(argv[1]);
            printf("size = %d\n",size);

            float *h_in;
            float h_out;

            h_in = (float*) malloc(size*sizeof(float));

            for (int i = 0; i < size; i++) {
                        h_in[i] = 1;
            }

            startClock("compute");
            nonCudaReduce(&h_out,h_in,size);
            stopClock("compute");

            printf("The sum is %f\n",h_out);

            free(h_in);

            printClock("compute");
}

void nonCudaReduce(float* out, float* in, int size) {
            *out = 0.0;
            for (int i = 0; i < size; i++) {
                        *out += in[i];
            }
}
```

CUDA Version - Host Side

```
int main(int argc, char** argv) {

        int size = 1024*1024;
        printf("size = %d\n",size);

              void *d_in;     // device data
              void *d_mid;    // device data - middle results
              void *d_out;    // device data - the answer

              float *h_in;    // host data
              float h_out;

              int numBlocks = 1024;

              cudaMalloc(&d_in,size*sizeof(float));
              cudaMalloc(&d_mid,numBlocks*sizeof(float));
              cudaMalloc(&d_out,sizeof(float));

              h_in = (float*) malloc(size*sizeof(float));

              for (int i = 0; i < size; i++) {
                            h_in[i] = 1;
              }

              startClock("copy data to device");
              cudaMemcpy(d_in,h_in,size*sizeof(float),cudaMemcpyHostToDevice);
              stopClock("copy data to device");

              startClock("compute");

              // use max threads/block and the required # of blocks AND
              // ask for some shared memory

              reduce<<<1024,1024,1024>>>((float*) d_mid,(float*) d_in,size);
              reduce<<<1,1024,1024>>>((float*)d_out,(float*)d_mid,1024);
              cudaThreadSynchronize();

              stopClock("compute");

              startClock("copy data to host");
              h_out = -17;
              cudaMemcpy(&h_out,d_out,sizeof(float),cudaMemcpyDeviceToHost);
              stopClock("copy data to host");

              printf("The total is %f\n",h_out);
              free(h_in);
              cudaFree(d_in);
              cudaFree(d_out);

              printClock("copy data to device");
              printClock("compute");
              printClock("copy data to host");
}
```

Cuda Version - Kernel

```
__global__ void reduce(float* out, float* in, int size) {
        __shared__ float temp[1024];

        int index = blockDim.x*blockIdx.x + threadIdx.x;
        int myId = threadIdx.x;

        if (index >= size) return;

        // move data to shared memory for speed

        temp[myId] = in[index];
        __syncthreads();

        int stride = blockDim.x/2;
        while (stride >= 1) {
                if (myId < stride) {
                        temp[myId] += temp[myId + stride];
                }
                __syncthreads();
                stride = stride/2;
        }
        out[blockIdx.x] = temp[0];
}
```

Timing

- For 1,000,000 elements
  - Serial Code 4.775 ms
  - Cuda 1.148 (including copy - .7ms is the actual computation)
- This can be greatly improved if the operator is more complex or if the input data can be computed on the device!
- A deeper study of the hardware can also have a significant impact on the timing. (to be continued at a later date!)

Another Data Pattern - Scanning

- A “scan” of an array is when you replace each element of an array by the sum of all of the elements before it plus itself (think turning a probability density into a distribution).
- This works with all binary associative operators as well (+, *, max, min, etc.)
- Example - the array [1, 7, -3, 2, 9, -4] is transformed to [1, 8, 5, 7, 16, 12].
- Applications range from sorting to checkbook balancing, to management of resources, etc.
- It SEEMS like an intractable serial problem, but it isn’t! (See the white board!)

The point of all of this...

- When I hear colleagues or the typical web curmudgeon make statements like:

  “I don’t know why they ask me about design patterns or O notation at job interviews ... you just don’t need it ... “

  It just drives me nuts. These are the atoms of our business. These form the periodic table for programmers.

  The good people at nVidia will gladly tell you that the key to programming CUDA well is to understand parallel programming, its patterns and its algorithms.

  This will get you 90% to peak performance. If you need the next 10%, you will have to learn the hardware.

One more pattern for the road - transpose

- This isn't so much a pattern as a Kata - a routine that one performs to sharpen your skills - to build muscle memory - to free your mind to be able to work on the big picture. (And it is a lot of fun to make things work faster!)
- The problem: Give a 1024 by 1024 array - transpose it. A(i,j) -> A(j,i).

Serial Code will run in 11.6ms.

```
#define DIM 1024

int main(int argc, char** argv) {

            float *h_in;
            float *h_out;

            h_in = (float*) malloc(DIM*DIM*sizeof(float));
            h_out =(float*) malloc(DIM*DIM*sizeof(float));

            int value = 1;
            for (int i = 0; i < DIM; i++) {
                        for (int j = 0; j < DIM; j++) {
                                    h_in[i + j*DIM] = value++;
                        }
            }

            startClock("compute");
            nonCudaTranspose(h_out,h_in,DIM);
            stopClock("compute");

            free(h_in);
            free(h_out);

            printClock("compute");
}

void nonCudaTranspose(float* out, float* in, int size) {
            for (int i = 0; i < size; i++) {
                        for (int j = 0; j < size; j++) {
                                    out[j + i*size] = in[i + j*size];
                        }
            }
}
```

First Try - no threading - time 367ms

```
#define DIM 1024

int main(int argc, char** argv) {

                float *h_in;
                float *h_out;

                h_in = (float*) malloc(DIM*DIM*sizeof(float));
                h_out =(float*) malloc(DIM*DIM*sizeof(float));

                void *d_in;
                void *d_out;

                cudaMalloc(&d_in,DIM*DIM*sizeof(float));
                cudaMalloc(&d_out,DIM*DIM*sizeof(float));

                int value = 1;
                for (int i = 0; i < DIM; i++) {
                                for (int j = 0; j < DIM; j++) {
                                                h_in[i + j*DIM] = value++;
                                }
                }

                startClock("copy in");
                cudaMemcpy(d_in,h_in,DIM*DIM*sizeof(float),cudaMemcpyHostToDevice);
                stopClock("copy in");

                startClock("compute");
                cudaTransposeSerial<<<1,1>>>((float*)d_out,(float*)d_in,DIM);
                cudaThreadSynchronize();
                stopClock("compute");

                startClock("copy out");
                cudaMemcpy(h_out,d_out,DIM*DIM*sizeof(float),cudaMemcpyDeviceToHost);
                stopClock("copy out");

                free(h_in);
                free(h_out);
                cudaFree(d_in);
                cudaFree(d_out);

                printClock("copy in");
                printClock("compute");
                printClock("copy out");
}

__global__ void cudaTransposeSerial(float* out, float* in, int size) {
                for (int i = 0; i < size; i++) {
                                for (int j = 0; j < size; j++) {
                                                out[j + i*size] = in[i + j*size];
                                }
                }
}
```

Second Try - 1 thread/row - 3.4ms (100x!)

```
int main(int argc, char** argv) {

                float *h_in;
                float *h_out;

                h_in = (float*) malloc(DIM*DIM*sizeof(float));
                h_out =(float*) malloc(DIM*DIM*sizeof(float));

                void *d_in;
                void *d_out;

                cudaMalloc(&d_in,DIM*DIM*sizeof(float));
                cudaMalloc(&d_out,DIM*DIM*sizeof(float));

                int value = 1;
                for (int i = 0; i < DIM; i++) {
                                for (int j = 0; j < DIM; j++) {
                                                h_in[i + j*DIM] = value++;
                                }
                }

                startClock("copy in");
                cudaMemcpy(d_in,h_in,DIM*DIM*sizeof(float),cudaMemcpyHostToDevice);
                stopClock("copy in");

                startClock("compute");
                cudaTransposeRow<<<1,1024>>>((float*)d_out,(float*)d_in,DIM);
                cudaThreadSynchronize();
                stopClock("compute");

                startClock("copy out");
                cudaMemcpy(h_out,d_out,DIM*DIM*sizeof(float),cudaMemcpyDeviceToHost);
                stopClock("copy out");

                free(h_in);
                free(h_out);
                cudaFree(d_in);
                cudaFree(d_out);

                printClock("copy in");
                printClock("compute");
                printClock("copy out");
}

__global__ void cudaTransposeRow(float* out, float* in, int size) {
                int row = threadIdx.x;

                for (int j = 0; j < size; j++) {
                                out[j + row*size] = in[row + j*size];
                }
}
```

Third - fully parallized - .5 ms (10x more!)

```
int main(int argc, char** argv) {

                float *h_in;
                float *h_out;

                h_in = (float*) malloc(DIM*DIM*sizeof(float));
                h_out =(float*) malloc(DIM*DIM*sizeof(float));

                void *d_in;
                void *d_out;

                cudaMalloc(&d_in,DIM*DIM*sizeof(float));
                cudaMalloc(&d_out,DIM*DIM*sizeof(float));

                int value = 1;
                for (int i = 0; i < DIM; i++) {
                                for (int j = 0; j < DIM; j++) {
                                                h_in[i + j*DIM] = value++;
                                }
                }

                startClock("copy in");
                cudaMemcpy(d_in,h_in,DIM*DIM*sizeof(float),cudaMemcpyHostToDevice);
                stopClock("copy in");

                startClock("compute");
                cudaTransposeRow<<<1024,1024>>>((float*)d_out,(float*)d_in,DIM);
                cudaThreadSynchronize();
                stopClock("compute");

                startClock("copy out");
                cudaMemcpy(h_out,d_out,DIM*DIM*sizeof(float),cudaMemcpyDeviceToHost);
                stopClock("copy out");

                free(h_in);
                free(h_out);
                cudaFree(d_in);
                cudaFree(d_out);

                printClock("copy in");
                printClock("compute");
                printClock("copy out");
}

__global__ void cudaTransposeRow(float* out, float* in, int size) {
                int row = threadIdx.x;
                int col = blockIdx.x;

                out[col + row*size] = in[row + col*size];
}
```

One more optimization - global memory coalescing

- Just like your CPU, the GPU is very happy to access consecutive addresses of the global memory
  - We were very good about that for reading, but not very good for writing.
- The big idea - cover the array with tiles 8x8, 16x16, or 32x32. Read a tile into local memory. Then, write out in order to global. Lets try:

Coalesced - .46 - a measly 20%

```
                startClock("compute");

                int tileSize = 8;
                int tempMem = tileSize*tileSize*sizeof(float);
                dim3 blocks(128,128,1);
                dim3 threads(8,8);

                cudaTransposeCoalesce<<<blocks,threads,tempMem>>>((float*)d_out,(float*)d_in,DIM);
                cudaThreadSynchronize();
                stopClock("compute");

                startClock("copy out");
                cudaMemcpy(h_out,d_out,DIM*DIM*sizeof(float),cudaMemcpyDeviceToHost);
                stopClock("copy out");

                free(h_in);
                free(h_out);
                cudaFree(d_in);
                cudaFree(d_out);

                printClock("copy in");
                printClock("compute");
                printClock("copy out");
}

__global__ void cudaTransposeCoalesce(float* out, float* in, int size) {

                __shared__ float shared[1024];


                // starting points inside the input data

                int iStart = blockDim.x*blockIdx.x;
                int jStart = blockDim.y*blockIdx.y;

                // let adjacent threads pick up adjacent items from input
                // transpose on the fly

                float data = in[iStart + threadIdx.x + (jStart + threadIdx.y)*size];
                shared[threadIdx.y + threadIdx.x*blockDim.x] = data;

                __syncthreads();

                // ok now put them back to out, but travel with the grain!

                int temp = jStart;
                jStart = iStart;
                iStart = temp;

                out[iStart + threadIdx.x + (jStart + threadIdx.y)*size] = shared[threadIdx.x + threadIdx.y*blockDim.y];
}
```

Lessons from this example

- There are some real obvious optimizations with enormous payoffs
- There are the more suble (ninja) optimizations with slightly better payoffs
- Believe it or not, there are still a few more optimizations we can get for another 10-20% or so, but there is so much more to learn

A Porting Example

- I was brought a Fortran program - an early fork of a program called QuantumEspresso.
  - It already had some self timing code inside and it identified the “hot spot” of the code to be the use of a Fast Fourier Transform (fft)

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i 2\pi k n / N}.$$

  - This is famously a compute intensive operation but extremely useful for data analysis. There is an excellent library that does this called fftw (fastest fourier transform in the west) that has been ported to CUDA - so that it is somewhat, at least conceptually, compatible.

FFTW vs cuFFT

- FFTW
  - Two steps
    - First you create a “plan” - basically telling it how many data points there are and which direction you are computing (X_n -> x_n or vice versa)
    - Then you execute the plan. The execution of the plan has several more options depending upon how many arrays you have etc.
- cuFFT
  - Two steps (so far so good)
    - First you create a plan which is much more complex, describing both the problem and the setup of data on the device
    - Next you execute the plan with very few parameters

Back to Quantum Espresso

- They have a very sparse 3-D array of points (72x72x72) that they have to transform
  - Since it is sparse, and the FFT is an expensive operation, they spend a considerable amount of time locating lines and planes within the 3D array that are all zeros and for which you do not have to compute the FFT. This was a big win for the CPU version of the code, shown below

```
do k = 1, nz
   do j = 1, ny
      jj = j + ( k - 1 ) * ldy
      ii = 1 + ldx * ( jj - 1 )
      if ( do_fft_x( jj ) == 1 ) THEN
        call FFTW_INPLACE_DRV_1D( bw_plan( 1, ip),m,f( ii ),incx1,incx2 )
        calls = calls + 1
      endif
   enddo
enddo
```

Results:

	Non-Cuda	Naive Cuda	Hey - processors are free Cuda
Full Code	42.90 sec	77.1 sec	21.04 sec
First FFT	18.14 sec	38.03 sec	8.36 sec
Second FFT	17.82 sec	34.15 sec	6.8 sec

Porting Notes

- The golden asset - someone who *knows* the code.
  - If someone can tell you that this routine “maps” or “reduces” or “scans” or ... The battle is over!
- Don’t overlook obvious improvements to start. Almost anyone looking at old code can find some performance improvements.
  - Fortran - take advantage of Matrix operations - they are way faster than doing them by hand
  - C - look out for bad memory use patterns.
  - Use library functions (QE had already been sped up by about 20% using the library fftw rather than the home-grown version)
- MEASURE - DON’T GUESS!!!

Wrap up

- CUDA is different, but it has its charms and learning it will help with
  - learning openGL
  - appreciating with the directive based languages have to do
  - most everything we learned about algorithms applies to openMP, MPI, Bluegene, etc. The details change, the weights change, but the concepts go way back to Connection Machine days (early 1980s)
- It (and its co-frameworks) are disruptive technologies - THEY MUST BE PAID ATTENTION TO!

Resources - 1

- Web sites
  - http://nvidia.com/object/cuda_home_new.html
    - Understand that CUDA is an nvidia product (free, not open source) and works only on nvidia hardware
  - http://www.khronos.org/opencl/
- MOOCs
  - Udacity (www.udacity.com) has an *outstanding* course that is available - Introduction to Parallel Programming. David Luebke and John Owens are master teachers.
  - Coursera (www.coursera.org) has a course called “Hetergeneous Parallel Programming” that is a bit more challenging

Resources - 2

- Books (Note, all of the titles are available via Safari Books Online available at a reasonable cost through BNL & maybe also through ACM Digital Library)
  - **CUDA Programming/A Developer's Guide to Parallel Computing with GPUs,** Shane Cook, 2013, Morgan Kaufmann.
    - Great for non-gear heads
  - **Programming Massively Parallel Processors/A Hands-on Approach,** David B. Kirk & Wen-mei W. Hwu. Second Edition, 2013, Morgan Kaufmann.
    - Considerably drier read then the above. Lots of typos. :-(
  - **GPU Computing Gems/Jade Edition**, Wen-Mei W. Hwu editor. 2012 Morgan Kaufmann.
    - An outstanding collection of 36 papers. You will almost certainly find something of interest.

Questions

- What topics for future tutorials - write to me - drs@bnl.gov
  - Deep dive into nvidia hardware & maximizing performance
  - Parallel patterns
  - IDEs, profilers, debuggers
  - Fortran & cuda
  - Python & cuda
  - OpenCL
  - OpenACC
  - Intel/Phi
  - CUDA libraries
  - Thrust (C++ STL for CUDA)
  - Porting non-parallel code to CUDA
  - Porting parallel code to CUDA
  - Multiple GPUs
  - Streams
  - System design