GPGPU Programming

An Introduction to the Technology and Programming Tutorial

Dave Stampf
CSC/BNL
drs@bnl.gov

July 17, 2014

Slides & Software at github.com/davestampf/GPUTalk2014

a passion for discovery

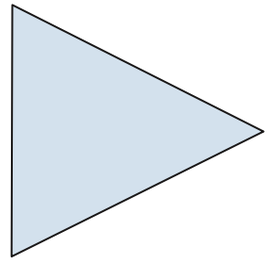
U.S. DEPARTMENT OF ENERGY | Office of Science

First, A Demo - A Summer Project

- Serial Program
  - On a good server, as written, takes 80 seconds/iteration.
  - User wanted at least 100 iterations
  - Also wanted to solve a problem 4x bigger
  - Was looking at 9 hours execution time. (>24 hours on his laptop)
- IMPORTANT - This program was dying to be parallelized!
  - CUDA
  - OpenMP
  - MPI

First, A Demo - A Summer Project

- Serial Program
  - On a good server, as written, takes 80 seconds/iteration.
  - User wanted at least 100 iterations
  - Also wanted to solve a problem 4x bigger
  - Was looking at 9 hours execution time. (>24 hours on his laptop)
- IMPORTANT - This program was dying to be parallelized!
  - CUDA
  - OpenMP
  - MPI



Or any combination!

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 (referring to a MOOC)...

“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.

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?

- Back in 1974, to program a world class mainframe like a CDC-7600, a programmer had to write much more “machine aware” programs:
  - moving data from external memory to main memory
  - ordering data so as to minimize memory latency
  - order instructions in order to take advantage of multiple functional units
  - Organize one’s computation and I/O to take advantage of “peripheral processing units” (parallel computational elements)
- It was painful, but as time (decades) passed, compilers (and later, silicon) assumed more and more of the responsibility of optimizing the program and let the programmer focus on process.
- The CDC-7600 was a disruptive architecture back then. GPU’s are today’s disruptive architecture - both are worth the effort!

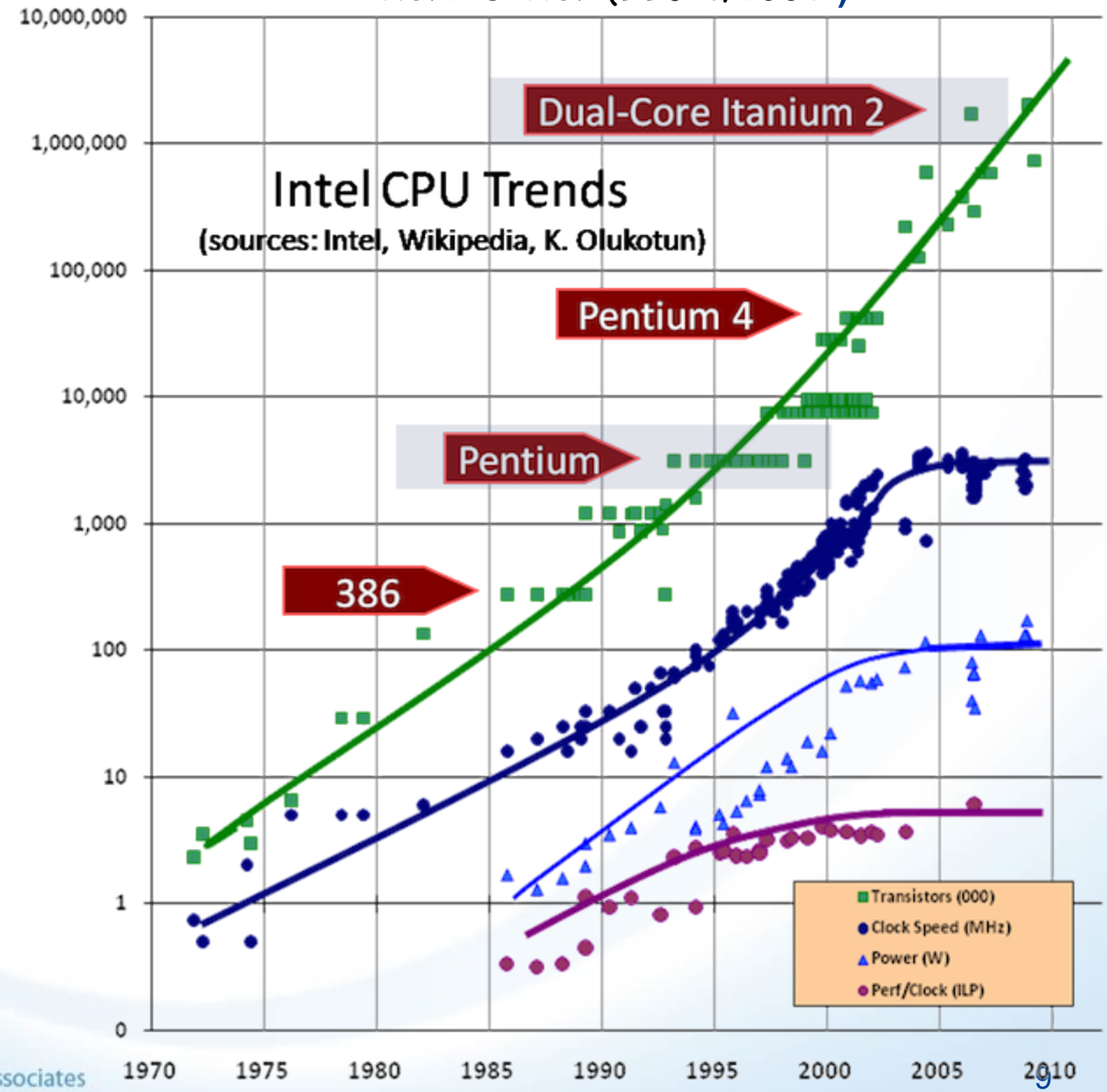
Disruptive?

- A CPU's (single core) performance has stagnated for the past 10(!!!) years.
- Exponential speed increases have flatlined.
- GPU/Coprocessor technologies offer a disruptive opportunity to get back on the exponential track.
- While the imaging industry (Adobe, Apple, games, 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!

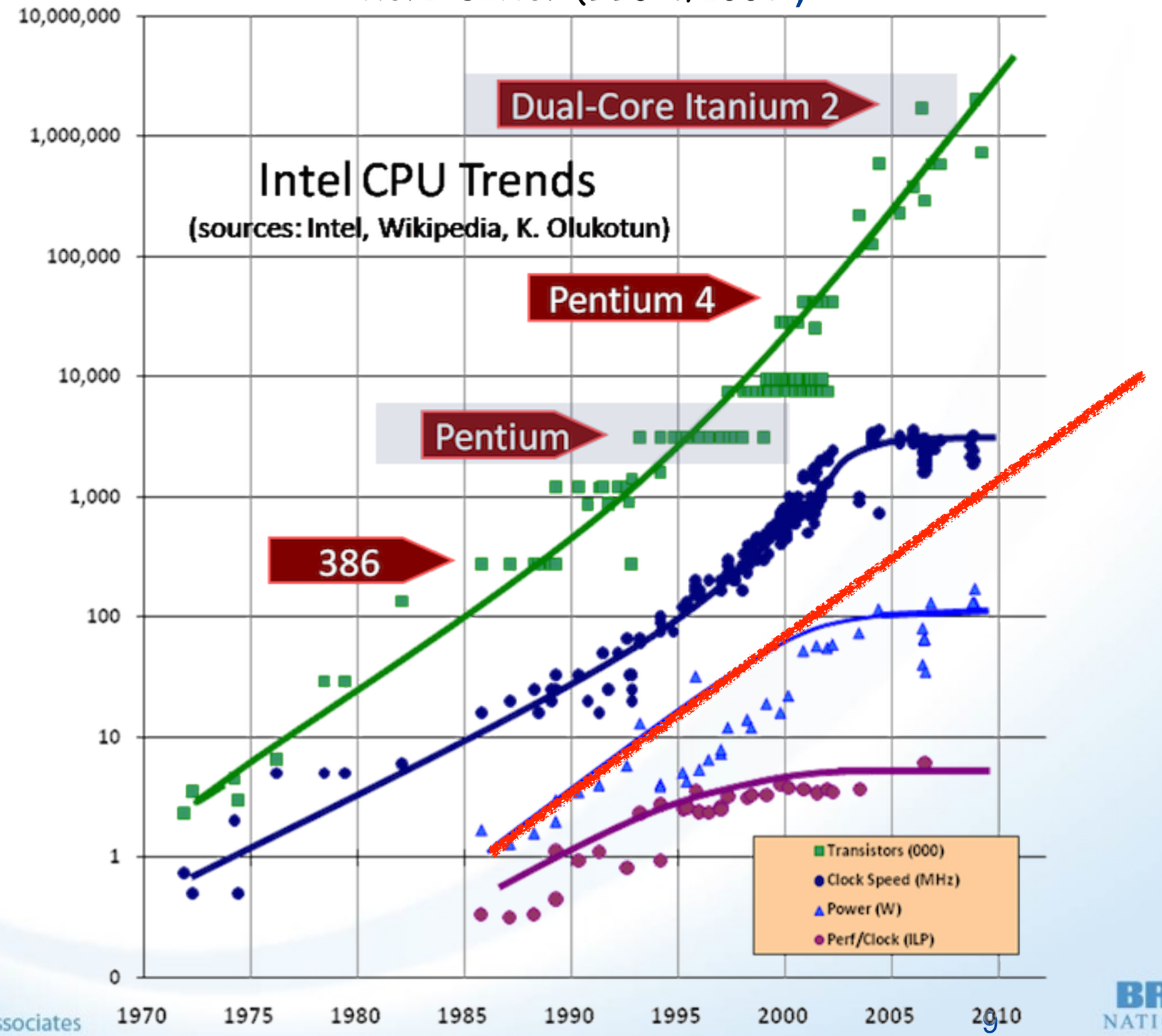
Classic (by now) Performance Graph

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



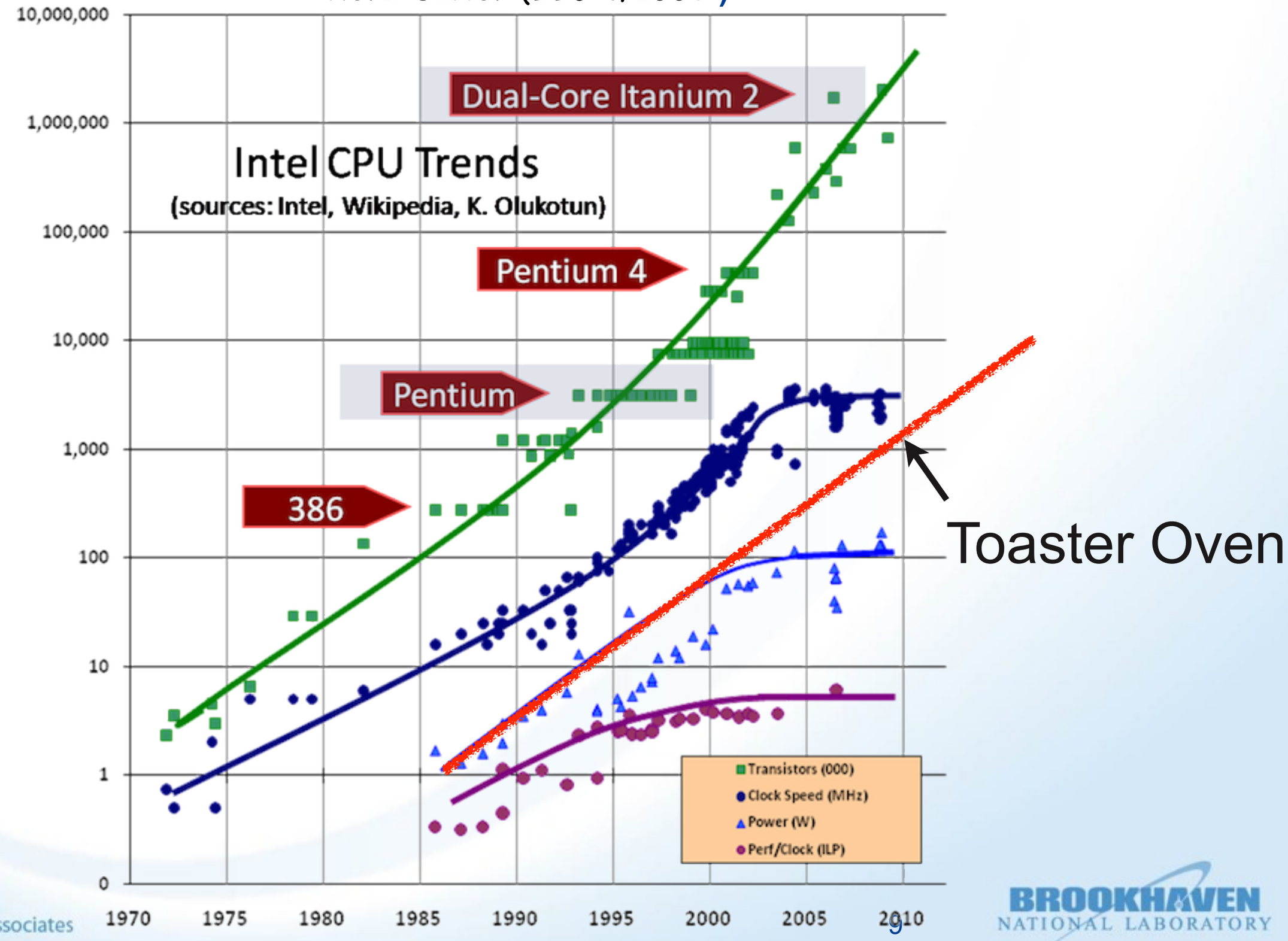
Classic (by now) Performance Graph

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



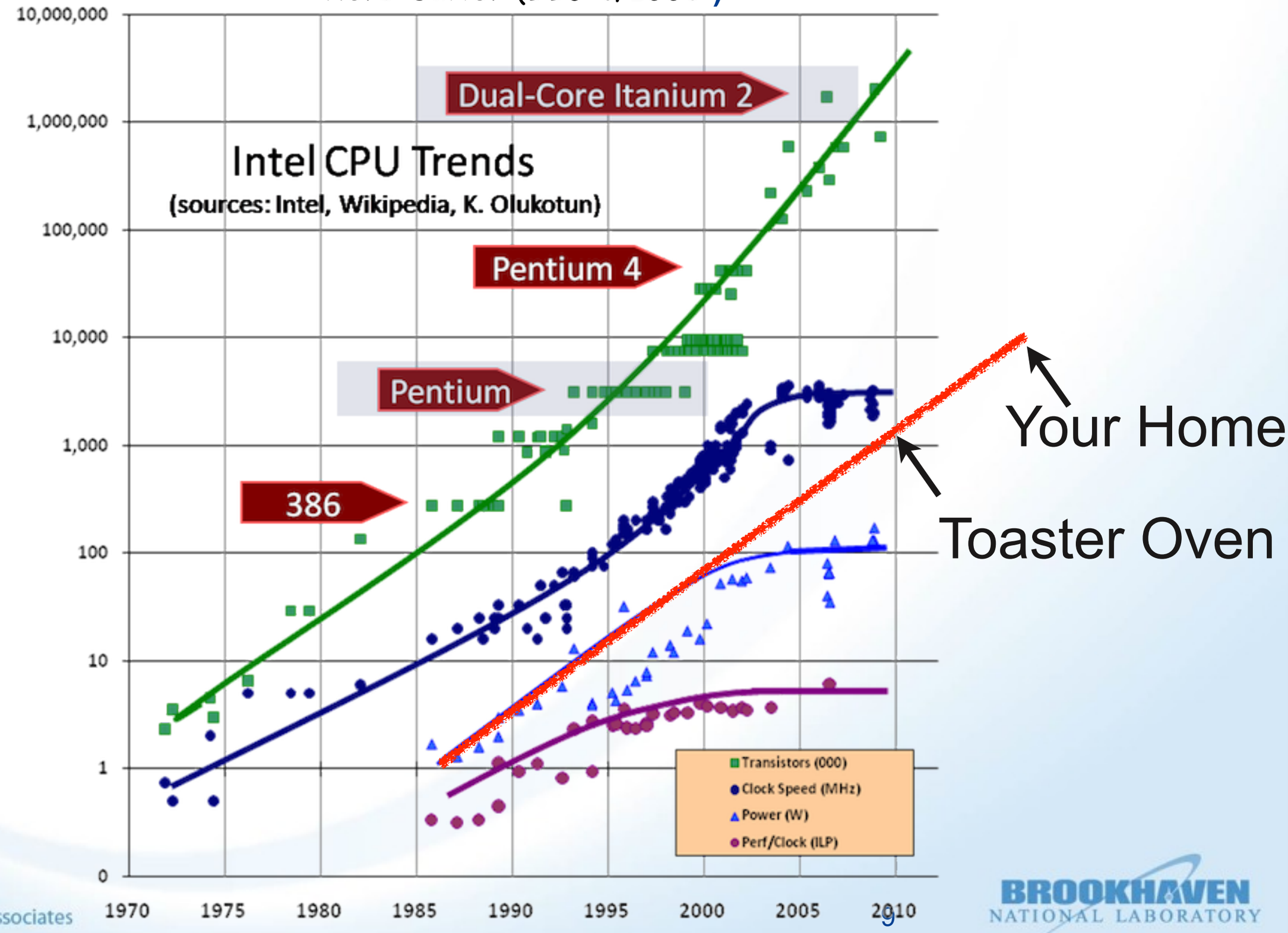
Classic (by now) Performance Graph

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



Classic (by now) Performance Graph

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



Quick Definition

- GPU - a computer component/chip/board whose design goal is to “do” graphics as quickly and smoothly as possible.
  - “do” meaning, scanning, rotating, lighting
  - Essentially - computing y = Ax + b (A is a matrix, y, x, b vectors) for the entire display.
- Luckily, a lot of scientific numerical computation takes the form of computing y = Ax + b!
  - How great would it be if scientists could leverage the GPU engine for their work?

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

CPU

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

GPU

- basically compute c = Ax + b
- many simple processors each with 1 thread (devoted to computing the above)
- One instruction (program) queue (it only ran 1 program but with changing data)
- many, many cores (has the space)
- *Strives for maximizing throughput*

Latency vs Throughput

- Classic example
  - Low Latency/low throughput
    - 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.
    - Latency = 24 hours. Throughput = .041 people/hour.
  - high latency/high throughput
    - bus with 80 people traveling the same distance @ 50mph. The first person arrives 48 hours later, but she has a whole orchestra with her!
    - Latency = 48 hours. Throughput = 1.67 people/hour (40x!)
- Latency is a measure of speed - how many seconds to get a single (or first) task done.
- Throughput is a measure of work performed/unit time. The work may be distributed across the unit time or just accumulated at the end.

Latency vs Throughput

- Computer example
  - Low latency/low throughput
    - imagine you have an array with 80 elements. You pass the array to a function to perform some operation on all of them. The CPU can perform that operation on each element in 24 ns.
    - 24ns later, you have the first element done. 48ns later the second, ..., 1920ns later you have all 80 done.
    - Latency, 24ns. Throughput, 80/1920 = .04 operations/ns.
  - High latency/high throughput
    - Now imagine the same array, but you have 80 threads, running at only half the speed of the CPU, but 80 all at once convert one element.
    - 24 ns into the task, nothing is done. 48ns into the task, all are done.
    - Latency - 48ns. Throughput 80/48 = 1.67 operations/ns.
- Classical CPUs deliver low latency - they want to get the first answer to you as quickly as possible
- For the GPU - computing the top left pixel of a screen is not useful if the bottom right pixel of the screen is computed too late. You want all of the pixels about the same time - just not late

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.)
  - pretty good speed up IF you happen to need the library routine at the program's hot-spot.
- Use directives (openACC, openMP, etc.)
  - inexpensive but modest (although improving) gains.
- 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
- Really buckle down & learn parallel patterns & algorithms
  - 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/...)
- Porting techniques
- Resources
- Afternoon - (your!) hands on. Either some canned problems or real ones.

Overview

CUDA is...

- A Language
  - A (very) small extension to the C language and a compiler (nvcc)
    - A ".cu" file is compiled to host code (using gcc or ...) and device code (ptx)
  - A small(-ish) API, that is pretty crude
    - Allows the CPU to command the GPU
- A *Highly Scalable* Computation Model
  - A Programming Model that can applied to a large variety of nVidia products

Reality Check

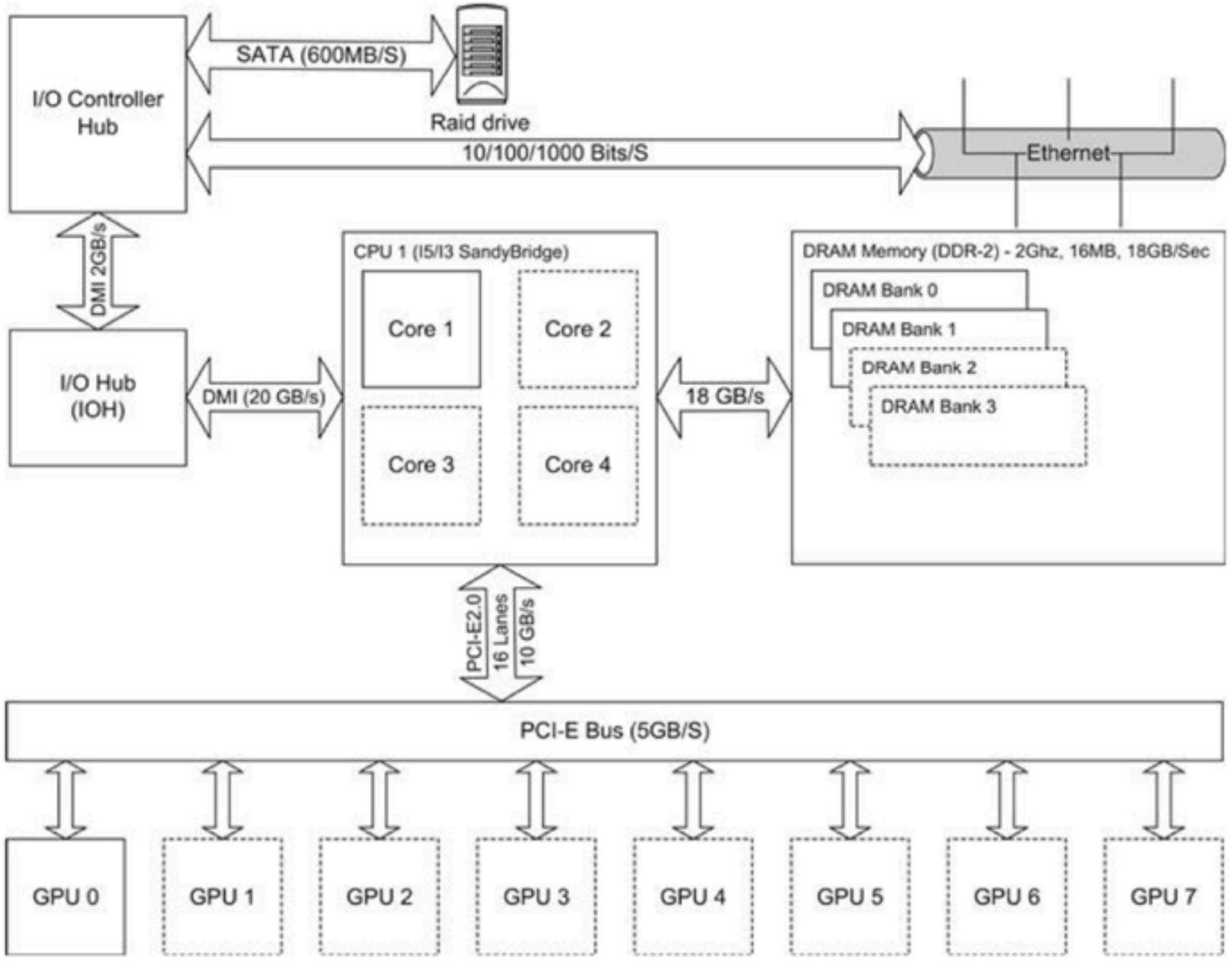
- Many schools (and MOOCs!) have semester courses on these topics while we have a few hours.
  - I'm going to resist taking the conversation down a path that loses 50% of the attendees or that really requires it own 3 hours to cover properly.
  - I will be taking some simplified shortcuts aimed to maximize 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

Overview - setup

- Make sure your system supports the CUDA SDK (see web site nvidia.com)
- 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
- Let's take a look...

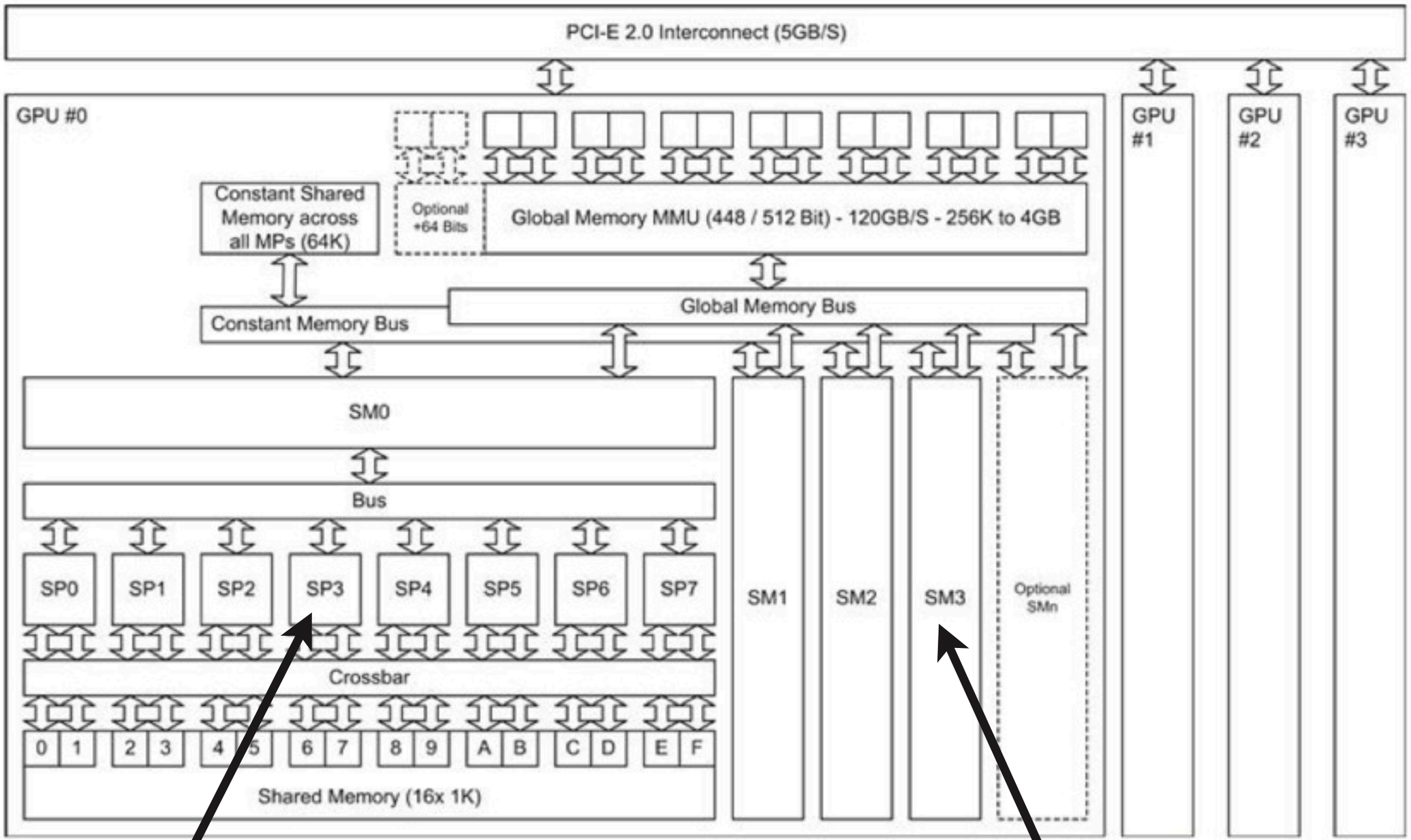
Overview Hardware

A Typical System (Sandy Bridge)



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

Block Diagram of GPU (G80/GT200)



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

SP = Streaming Processor
a.k.a. "Cuda Core

SM = Streaming Multiprocessor

Three instances of CUDA devices

- The Jetson - dev kit from nVidia ($192) - GK20A
  - 1 MP, 192 cores/MP, 192 CUDA cores, ~2G Memory
  - 2048 threads/MP, 1024 threads/block
  - 852 MHz GPU Clock/924 MHz memory clock, 64 bit mem bus
- Macbook Pro (Retina - 15”) - GeForce GT 750M
  - 2 MP, 192 cores/MP, 384 CUDA cores, 2G Memory
  - 2048 threads/MP, 1024 threads/block
  - 926 MHz GPU Clock/2.5 GHz memory clock
  - 128 bit memory bus width
- nvidia1 - Tesla K20m (2 GPUs)
  - 13 MP, 192 cores/MP, 2496 cuda cores, 5G Memory
  - 2048 threads/MP, 1024 threads/block
  - 700 MHz GPU clock/ 2.6 GHz memory clock
  - 320 bit memory bandwidth

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

```
[guest39@nvidia1 GPUTalk]$ ./cuda-devices
Number of cuda devices = 2
Device # 0
 name = Tesla K20m
 version = 3.5
 total global memory = 5032706048
 shared Memory/Block = 49152
 registers/block = 65536
 warp size = 32
 Max threads/block = 1024
 Max Threads Dim = 1024 x 1024 x 64
 Max Grid Size = 2147483647 x 65535 x 65535
 Multi-processor count = 13
 Max Threads/multiprocessor = 2048
Device # 1 ...
```

Overview - Software - basic programming model

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 and are invoked from the host are called in English “kernels”, but in CUDA, __global__
- 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 passing only a few scalars and which operate on the data already on the device.
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 uses or produces an array of data, each thread will be used to deal with just 1 element of the array.
- So, basically, you unwrap a for-loop and replace 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] = ...
```

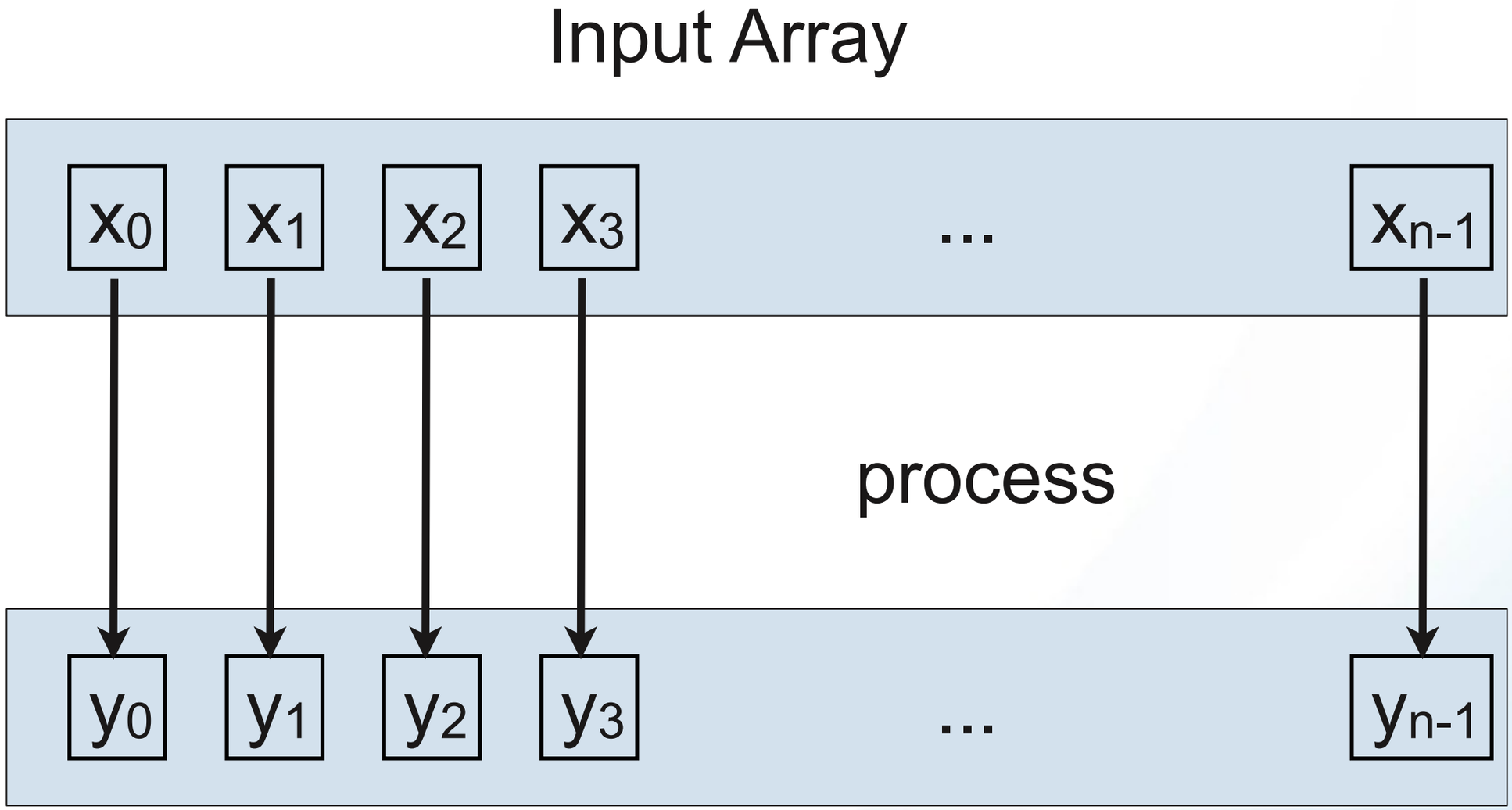
```
get threadId
result[threadId] = ...
```

...

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

But you only have to write 1 of these!

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

Dealing with arrays longer than 1024

- When you “launch” a kernel, you give it 3 launch parameters
  - # of “blocks”
  - # of threads in each of those blocks
  - amount of memory shared by the threads in any single block
- So if you need more than 1024 threads (the maximum allowed in a block) you simply ask for more blocks.
- Blocks are the scheduling unit for a GPU. The GPU matches up blocks to available MPs.
- Blocks run to completion, but there are no other guarantees.

Kernel's View

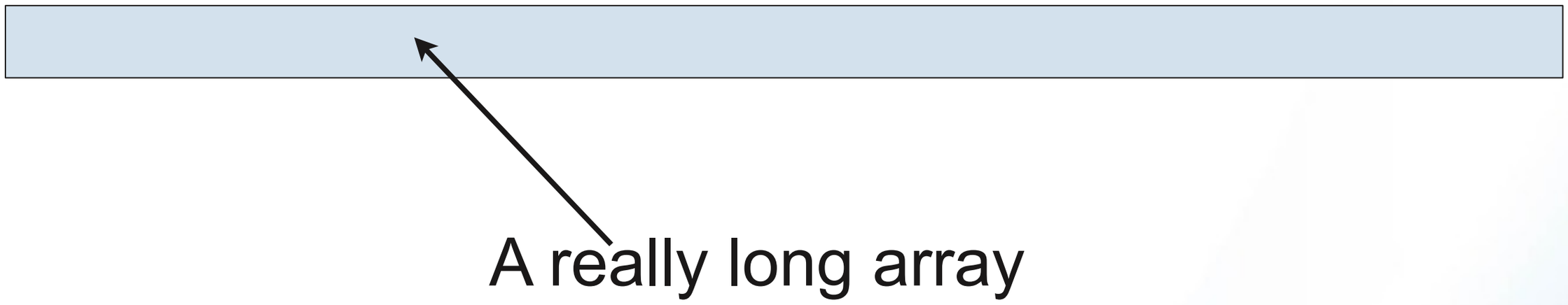
- Each kernel has 4 magic data structures
  - blockIdx.x - the id of the block it belongs to
  - blockDim.x - the number of blocks created
  - threadIdx.x = the id of the thread within a block [0..1024)
  - threadDim.x = the total number of threads in this block
- If you are mapping a thread to an array index,
  - int index = threadIdx.x + threadDim.x*blockIdx.x

Two Sides of Blocks

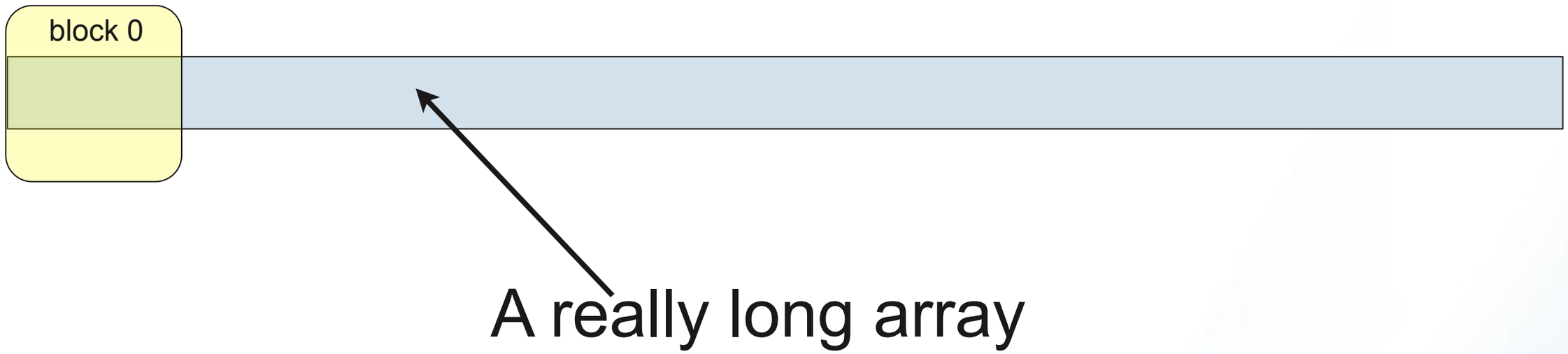
- As programmers, we use blocks to “cover” an linear array
  - There are also ways to arrange blocks in a 2-D pattern to cover a rectangle (e.g., image)
  - And of course ways to arrange blocks in a 3-D pattern to fill a solid prism.
- But to a GPU under CUDA, blocks are simply the mechanism for distributing work among the SMs.
  - All blocks associated with a kernel will execute before another kernel is started
  - The order that the blocks are run in is undetermined.
  - Once a block starts, it runs in that SM until complete
  - When complete, it is replaced by the next available block.

So, How to deal with arrays > 1024?

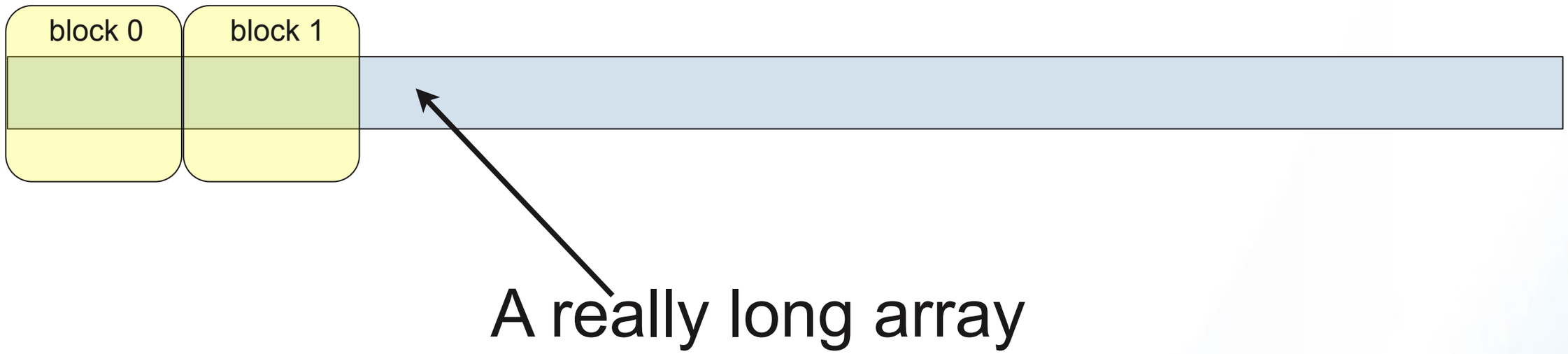
So, How to deal with arrays > 1024?



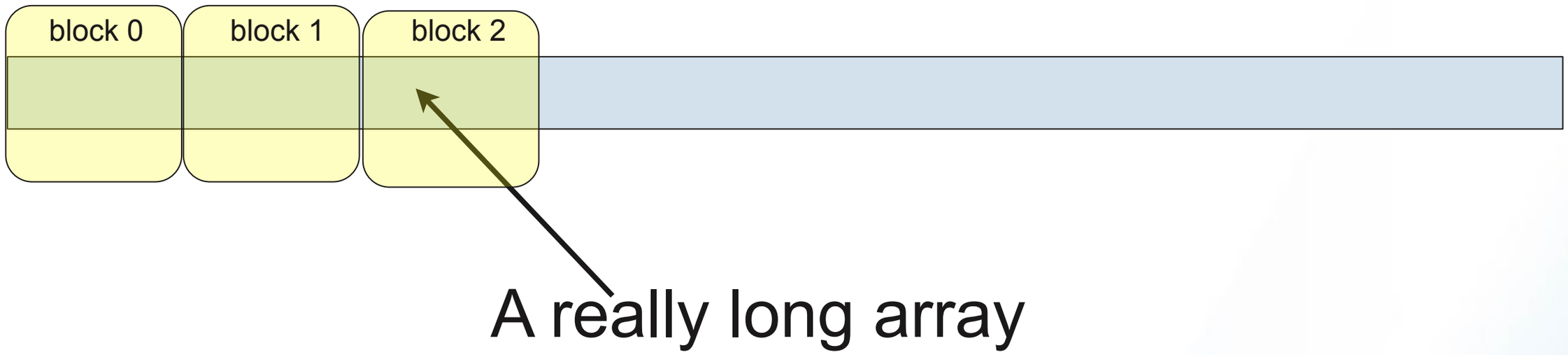
So, How to deal with arrays > 1024?



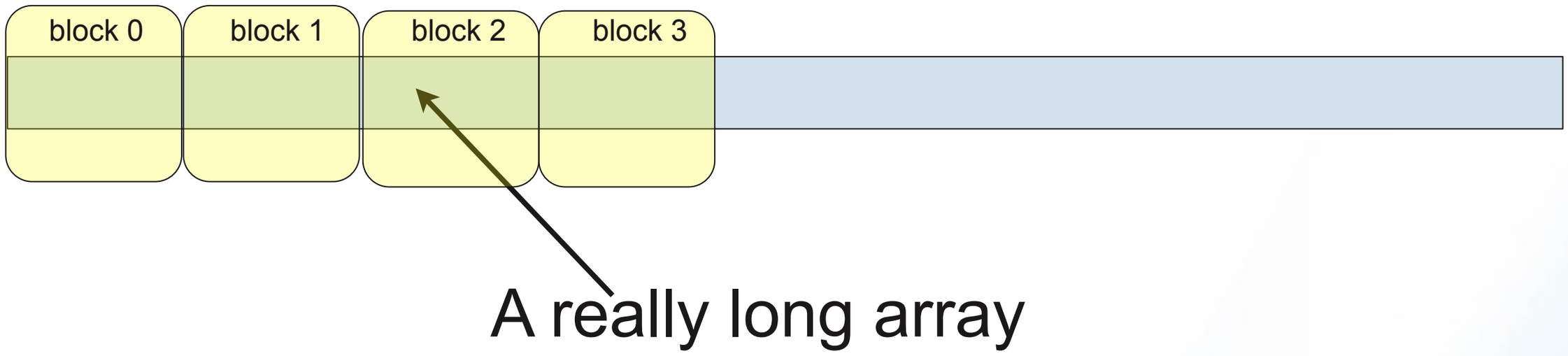
So, How to deal with arrays > 1024?



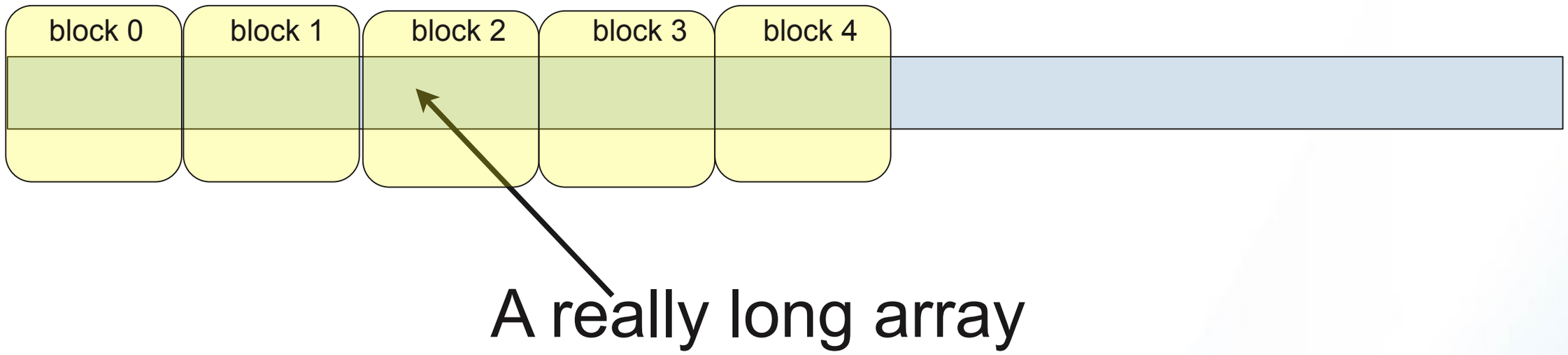
So, How to deal with arrays > 1024?



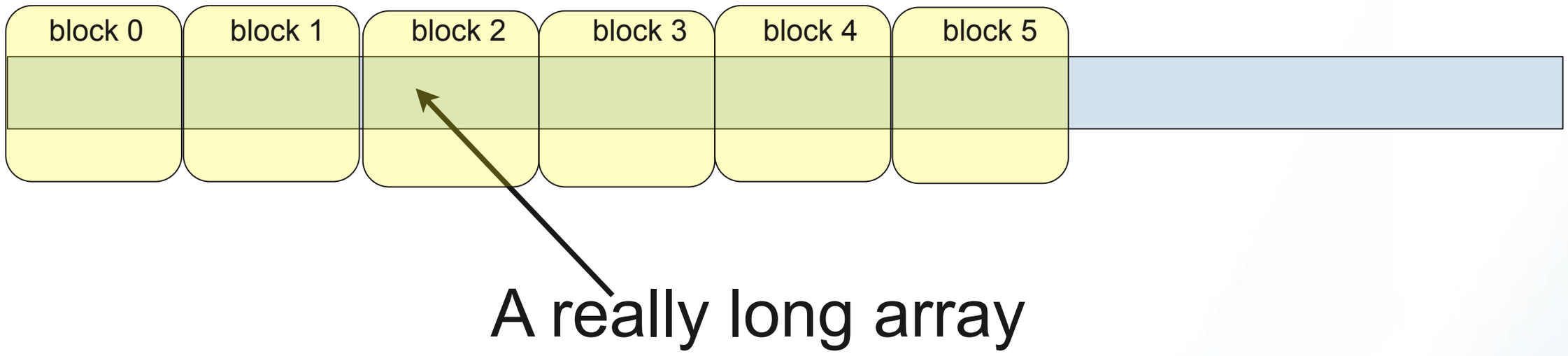
So, How to deal with arrays > 1024?



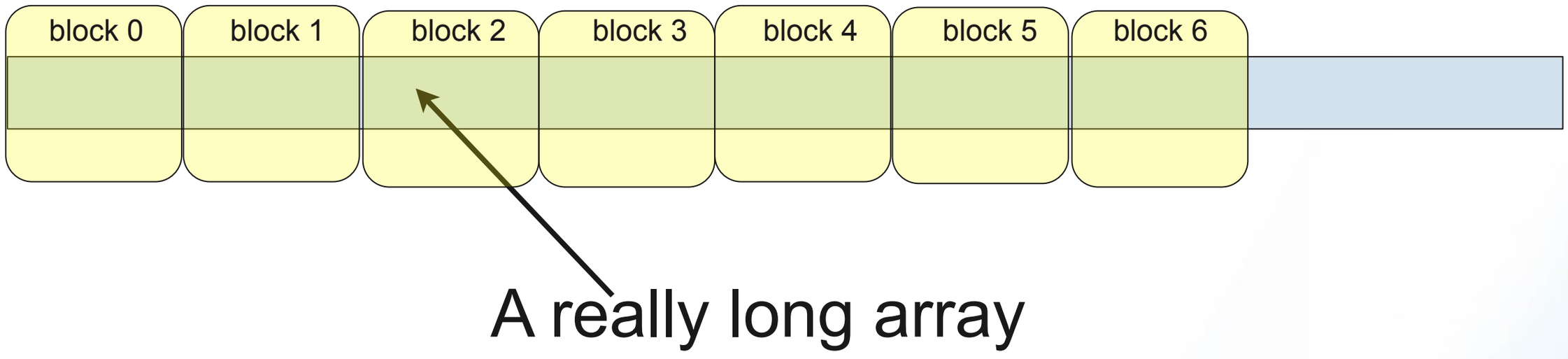
So, How to deal with arrays > 1024?



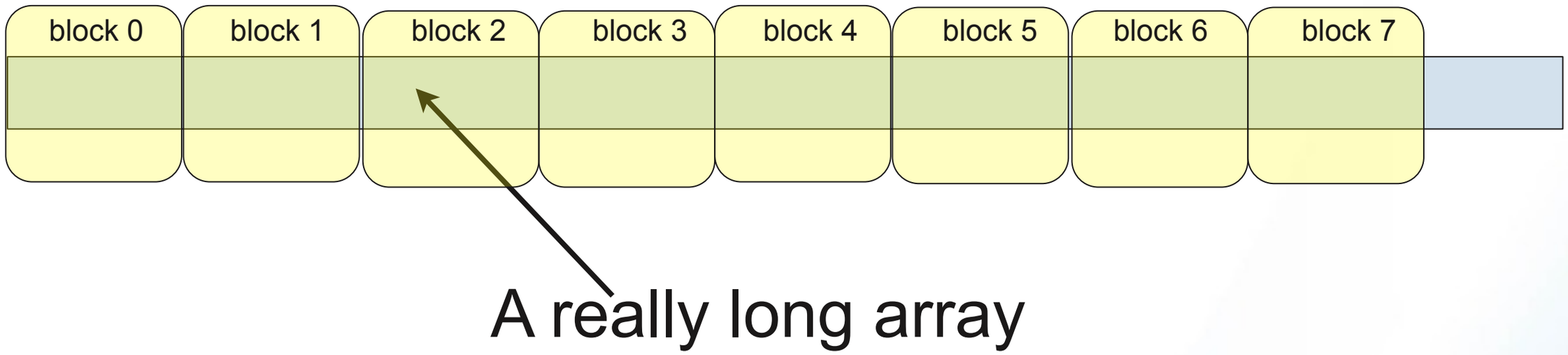
So, How to deal with arrays > 1024?



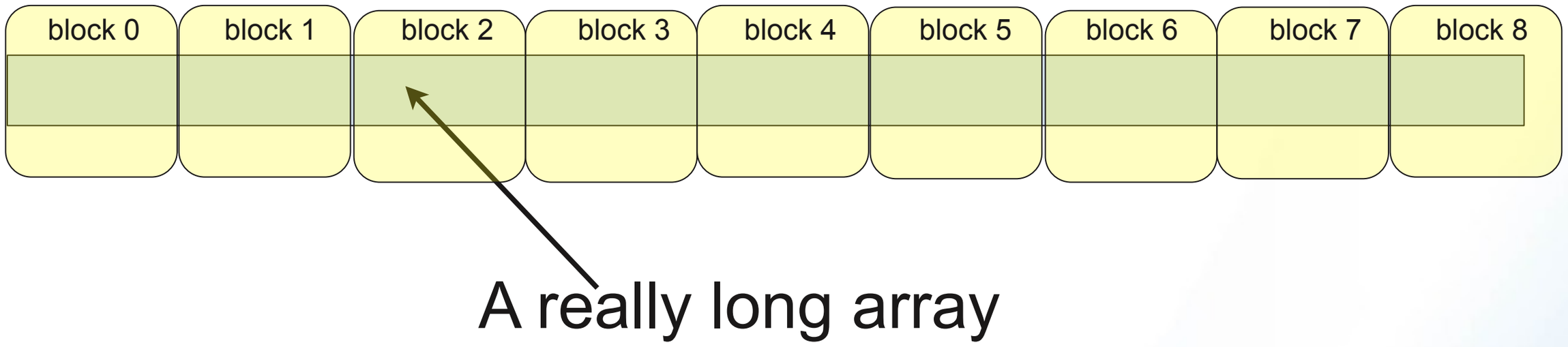
So, How to deal with arrays > 1024?



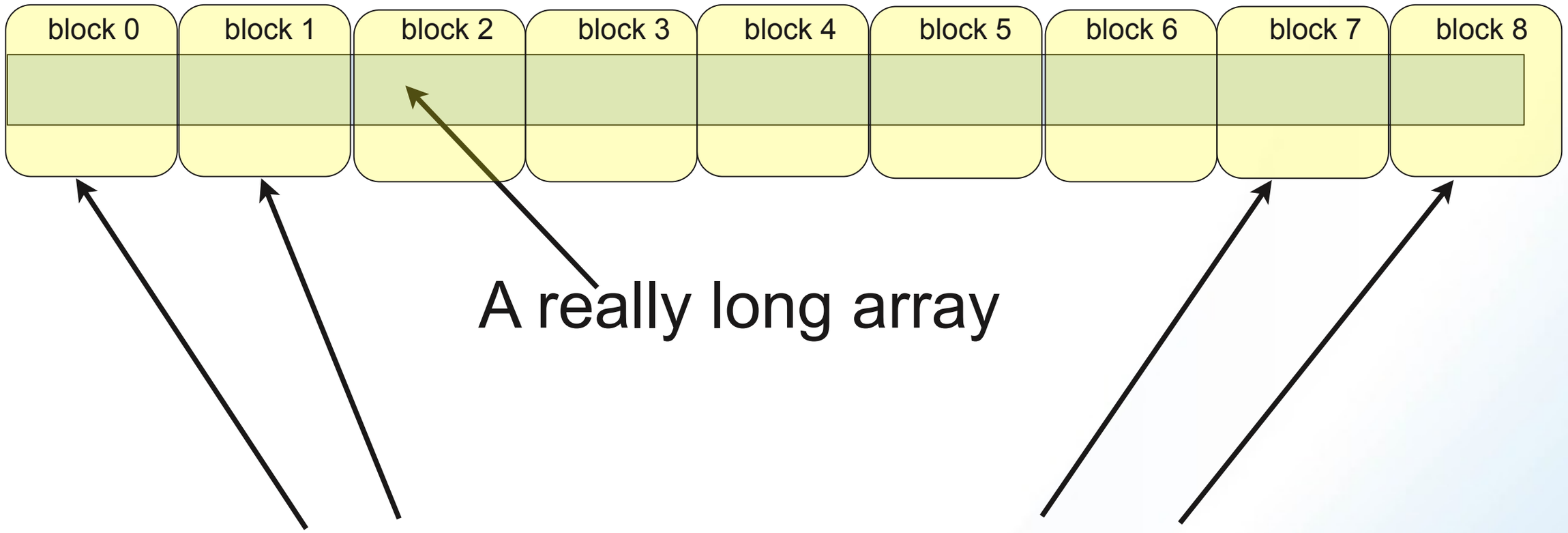
So, How to deal with arrays > 1024?



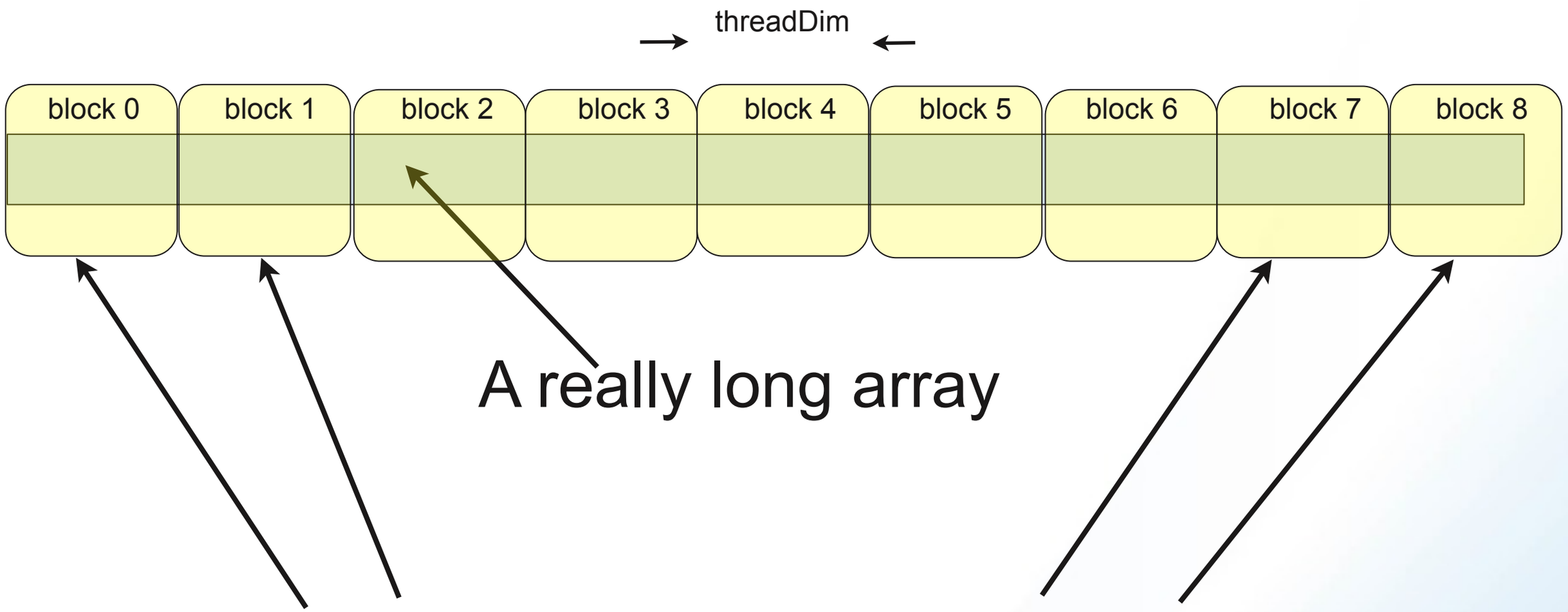
So, How to deal with arrays > 1024?



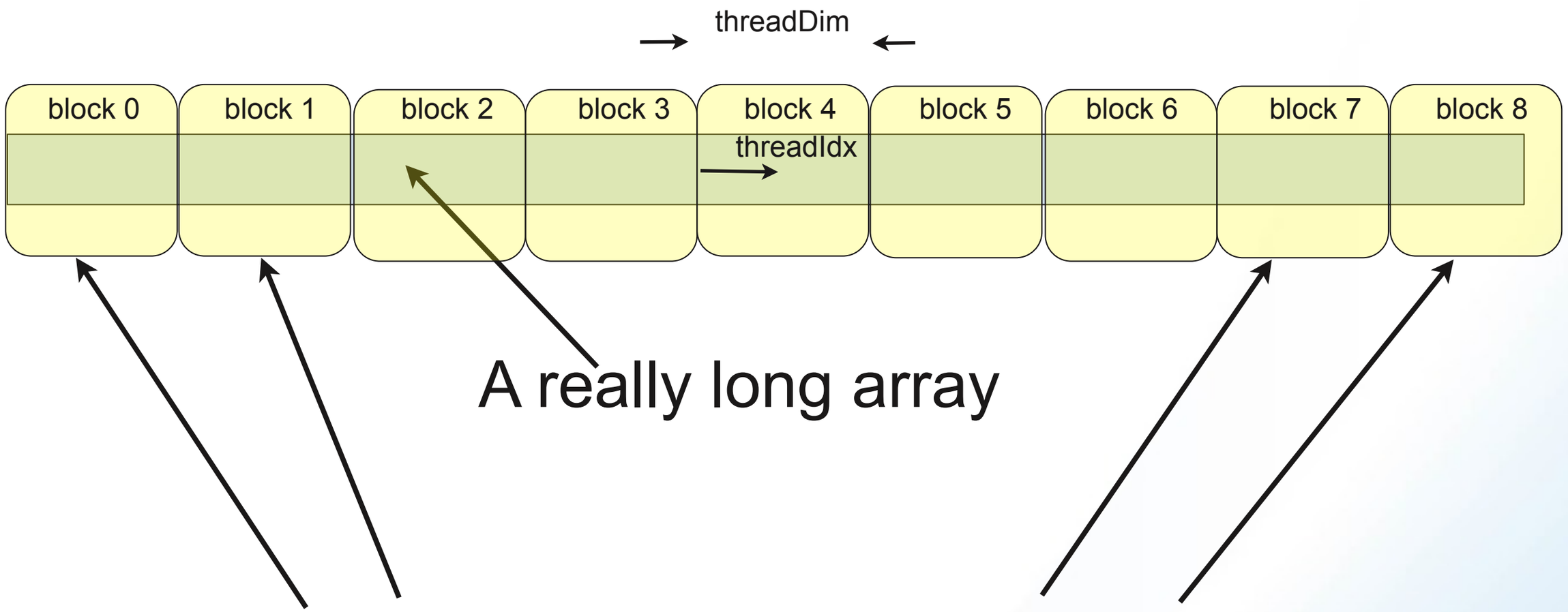
So, How to deal with arrays > 1024?



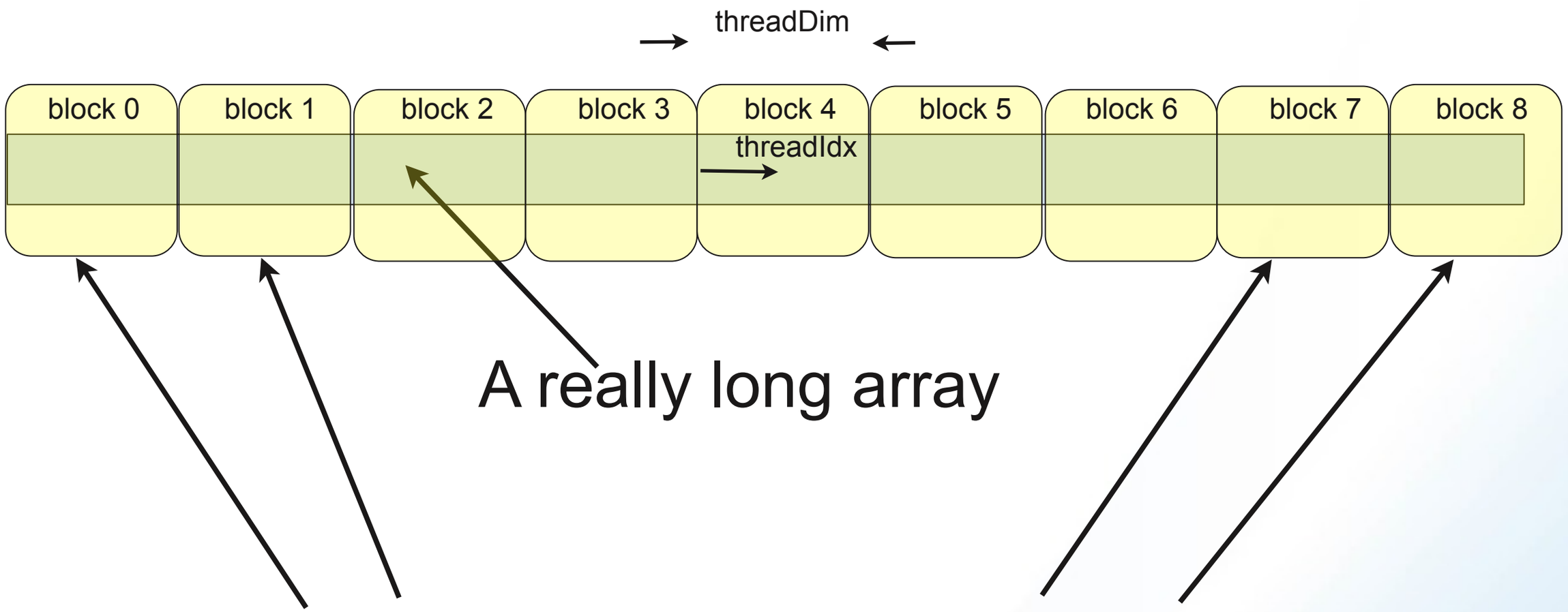
So, How to deal with arrays > 1024?



So, How to deal with arrays > 1024?



So, How to deal with arrays > 1024?



Each thread knows which block it belongs to and which thread it is within each block, so it can easily compute it's index into the array as

```
int index = threadIdx.x + threadDim.x*blockIdx.x
```

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]);
}
```

Review - 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)

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

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

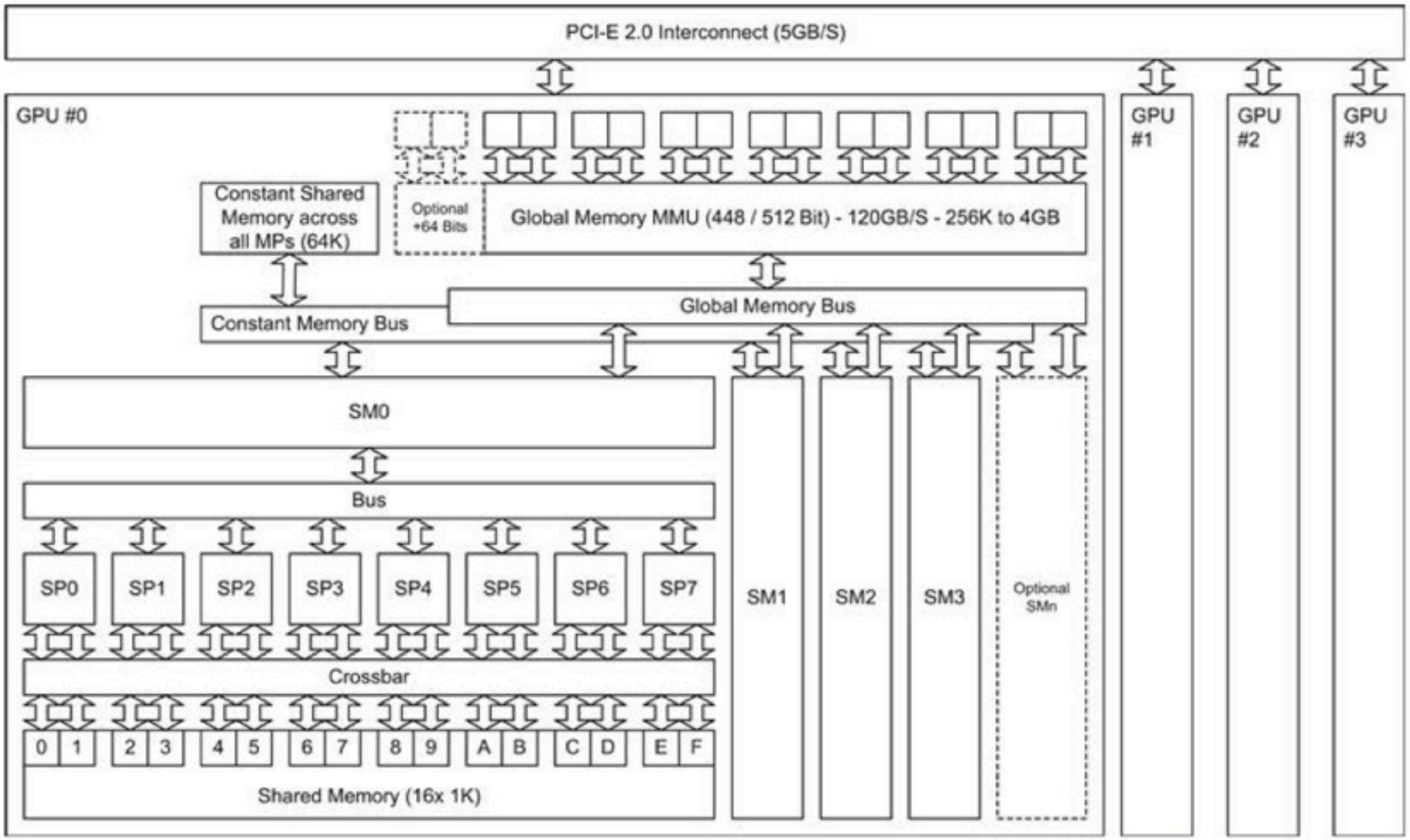
- C
  - void* d_data;
    float* h_data;
    cudaMemcpy(d_data,h_data,numberOfBytes,
    cudaMemcpyHostToDevice);
    //...

    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.

Block Diagram of GPU (G80/GT200)



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

Types of memory

- Global
  - Largest in size (1-6G)
  - Slowest in access (100's of clock cycles)
  - Widest memory width (384 bits/48 bytes/12 floats)
  - accessible to all threads
- Shared (between threads in a block)
  - Rather tight (48K)
  - Much faster than Global
  - Segmented and connected to the processors by a crossbar switch (need to be aware of contention - advanced)
- Register
  - Very tight (32K)
  - Single cycle access
  - Private to thread

Let's Talk Threads

- 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.

(non?) Cooperating Threads

- You have to be especially careful when threads cooperate on a task.
  - e.g. - lets compute 0 + 1 + 2 + ... + 1023. We will just have every thread in a block add its threadIdx into an accumulator
  - This requires and atomicAdd
- Race conditions are especially tricky
  - The occurs when you have multiple threads (we do!)
  - One thread (A) is anticipating that another thread (B) has written a piece of data that the thread (A) needs.
    - if thread B is too slow, thread A will have the wrong value.
    - if thread B is always fast enough, thread A will always have the right value
    - if thread B is sometimes slow, sometimes fast - you have random behavior.

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];
}
```

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];
            }
}
```

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**20 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.
- This version requires the cooperation of threads within a block

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

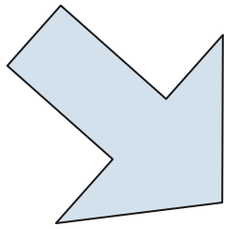
- For 1,000,000 elements
  - Serial Code 4.775 ms
  - Cuda 1.148 (including copy - .7ms is the actual computation)
- This can be greatly improved if the operator is more complex or if the input data can be computed on the device!
- A deeper study of the hardware can also have a significant impact on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

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 5 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 - more on this later. (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

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!)
  - (Coding this up is left as an exercise)

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.