Workshop on CUDA and OPENMP

Day 1 - Background

Dave Stampf
CSC/BNL
drs@bnl.gov
November 4, 2013

Slides & Software at github.com/davestampf/BNLWorkshop2013/

BROOKHAVEN NATIONAL LABORATORY

a passion for discovery

U.S. DEPARTMENT OF ENERGY | Office of Science

These days, all but the simplest computers are **parallel computers**

These days, all but the simplest computers are **parallel computers**

More accurately, all but the simplest computers are capable of running parallel programs *concurrently* - that is, they can do more than one thing at a time.

These days, all but the simplest computers are **parallel computers**

More accurately, all but the simplest computers are capable of running parallel programs *concurrently* - that is, they can do more than one thing at a time.

But to do so, requires YOU to do something!

This is not a new development!

- CDC 6600
  - Had 10 “functional units” - boxes of wires and stuff that could process specific types of instructions
    - branching
    - boolean
    - shift
    - long add
    - float add
    - divide
    - multiply
  - The (assembly language) programmer would then know enough to
    - intermix instruction types
    - if one loads data from memory into the register - run 3 other instructions before using that data
    - This trick turned a 2 Mips machine into a 10 Mips machine

This is not a new development!

- CDC 6600
  - Had 10 “functional units” - boxes of wires and stuff that could process specific types of instructions
    - branching
    - boolean
    - shift
    - long add
    - float add
    - divide
    - multiply
  - The (assembly language) programmer would then know enough to
    - intermix instruction types
    - if one loads data from memory into the register - run 3 other instructions before using that data
    - This trick turned a 2 Mips machine into a 10 Mips machine

(Yes Mips!)

More recent CPUs

- Have many (2-16) “cores”
- Even have some “virtual cores”
- Can re-order instructions on its own to keep the functional units busy
- Can follow both halves of an if statement
- Have instructions that work on arrays of values

Modern (Good =? $$$) Compilers

- unroll loops
- extract constant expressions
- re-order statements and inline functions -- hence the good advice that you don't try to debug or profile (line by line) an optimized code
- And in general play many of the same games the hardware does but with greater understanding of the entire problem.

So, what is left to do?

- Well, our languages permit us to do all of the things that one needs to do things like
  - solve differential equations
  - find the edges in an image
  - display and manipulate 3-D structures
  - compose slide shows
- But none of that is “built in”.
- We still have to write the algorithms and to turn our GFlop machines into TFlop machines, we have to add a touch of cleverness.

So what?

- With a modern intel cpu
  - 8 cores
  - 2 hyper-threads/core
  - 8 wide avx instructions (vector operations)
- For some parts of your code a 128x parallelization awaits you

So what?

- With a modern intel cpu
  - 8 cores
  - 2 hyper-threads/core
  - 8 wide avx instructions (vector operations)
- For some parts of your code a 128x parallelization awaits you

But how can you take advantage of it?

So what?

- With a modern intel cpu
  - 8 cores
  - 2 hyper-threads/core
  - 8 wide avx instructions (vector operations)
- For some parts of your code a 128x parallelization awaits you

But how can you take advantage of it?

More importantly - if you don't, with plain serial code, you are using less than 1% of the capability of this CPU!

Our Goals

- This morning - an introduction to some of the terminology and some of the theory that you will need to understand OpenMP and Cuda
- This afternoon - an introduction to Cuda with some examples
- Tuesday - Nick D'Imperio will lead you through a number of exercises to help you learn OpenMP - a technology that might be able to give you an order of magnitude improvement on some areas of your code with most any Intel based computer today
- Wednesday - I'm back with a similar hands on session with Cuda - capable of giving you several orders of magnitude improvement on certain computers.

Daily Schedule

9-10:15	Workshop
10:15 - 10:30	Coffee & Danish Break
10:30 - 12:00	Workshop
12:00 - 1:00	Lunch (on your own)
1:-2:15	Workshop
2:15-2:45	Coffee Break (on your own)
2:45 - 4:00	Workshop

A Quick Message from the Sponsor

- The HPC group at BNL consists of a handful of people (Nick D'Imperio, Meifeng Lin, Len Slatest, Dave Stampf, and Kwangmin Yu) who are willing and able to collaborate with you in an effort to get your programs
  - running in a parallel mode on your desktop
  - running in a parallel mode on local shared clusters
  - and maybe even running at world class computer centers around the US.
- Feel free to contact us!

Administrivia

- You all have accounts on hpc1.csc.bnl.gov that you can reach through ssh.bnl.gov & hopefully you have all verified that. (when you login, type “module load cuda” for cuda to work)
- Please bring your laptops
- There are a number of us here who will be able to help you if you get stuck - ASK FOR HELP.
  - Please note - we are learning this stuff too! We just got a head start and have a bit more experience. Helping you helps us enormously!
- ASK QUESTIONS!!!

Hardware Technologies

Technology	Cost	
Cores	$	It comes in the box.
Intel/Phi	$$	Cluster on a board - numerous pentiums running linux.
GPGPUs	$$$	Using enormous # of processors designed to do linear algebra for games, but we do linear algebra too!
Cluster	$$$$	Rooms full of commodity computing and high performance networking.

Note - these are NOT mutually exclusive!

The Software Technologies

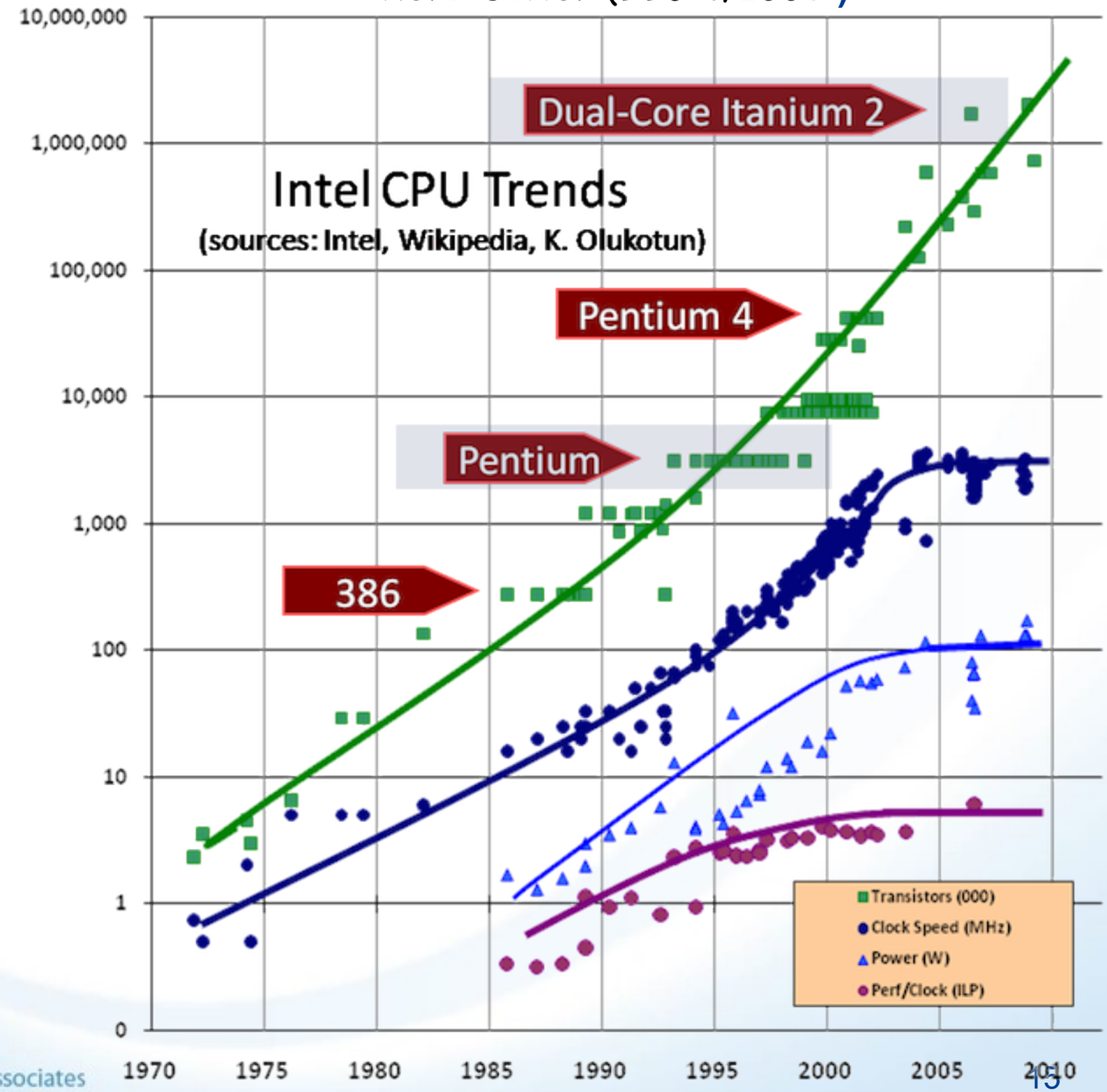
Technology	Cost - H/W & People	Description	Speed-up?
MPI (Message Passing)	$$$/$$	Distributed Memory/Network Intense	?
OpenMP	$/$	Make use of a processor's cores. Compiler Directives. Useful on any device	1x-30x
OpenCL	$/$$$	General shared memory - works on cores and co-processors like gpus	1x-30x-1000x
OpenACC	$/$	OpenMP for GPUs	1x-100x
CUDA	$/$$$	Hardcore programming and nVidia specific	1x-1000x
Intel Phi	$/$	Cluster on a board. Also uses a variant of OpenMP.	100x

Computing Faster - 3 Basic Ways

- Do more at each clock cycle
  - This is an approach that Intel and others have taken with instruction set expansion and the use of virtual cores.
- Use more processors
  - This is the approach we will be covering this week and various scales.
    - Smaller scale (but easier and faster programming) with OpenMP
    - Larger scale (but more difficult programming) with CUDA
- Use a faster computer
  - This had worked up until 2004 or so...

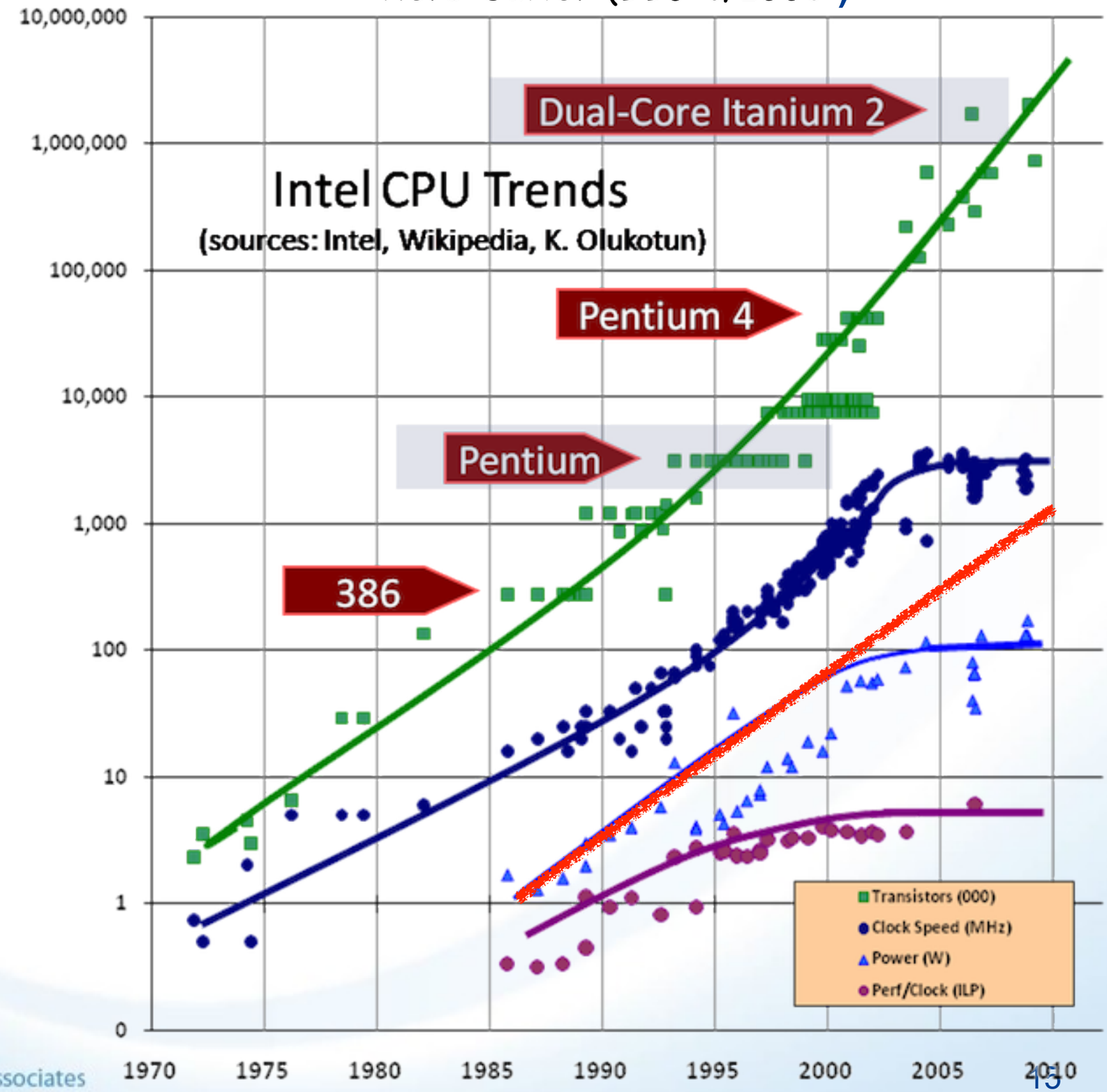
Classic (by now) Performance Graph

(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)



Some Basic Concepts

- Please be patient - some of these concepts will be very familiar to many of you, but to others... Please feel free to ask questions!
  - Latency vs Throughput
  - Transistor Budgets
  - Speed-up
  - Process vs Thread
  - Cores, Caches and Hyper-thread
  - O notation
  - Memory Hierarchy
  - Thinking Different

Minimizing Latency vs Maximizing Throughput

- Latency has units of time ... how long does it take to get something done, or how soon do you get an answer to a question you asked.
  - Traditional CPUs generally try to minimize latency - when you multiply two numbers together, you get the answer quickly.
  - Note that minimizing latency has suffered with the stall of clock speeds!
- Throughput has units of stuff/time - what is the total amount of work you can do over a given time period.
  - GPUs have been designed for maximizing throughput at the expense of latency.

Latency vs Throughput

- Classic Analogy - sports car vs bus crossing country
  - Fast sports car with a driver and passenger crosses county at 100 mph. Total time to cross country (3,000 miles) is 30 hours.
  - Bus with 80 passengers & driver crosses the country at 50 mph. Total time is 60 hours.
- Latency - sports car wins hands down!
- Throughput (passengers delivered/hr)
  - sport car - 1 passenger/30 hours = .033 passengers/hour
  - bus - 80 passengers/60 hours = 1.333 passengers/hour - 40 times greater! Bus wins.
- Note - GPUs optimize for throughput because they are working with pixels/sec.

Important Decision Point!

- Does your application need to optimize for latency or throughput?
  - Do you handle lots of data (passengers?) for which having only part of the result is not useful - you need to increase your throughput. (e.g., image processing, monte carlo, exploring a data domain, etc.) - GPU might be a good fit
  - Is the amount of processing you do per data element small and getting even partial results quickly important (e.g., single result searches, process control, etc.) - OpenMP might be a better fit.

Classic Throughput vs Latency Programing (Map)

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

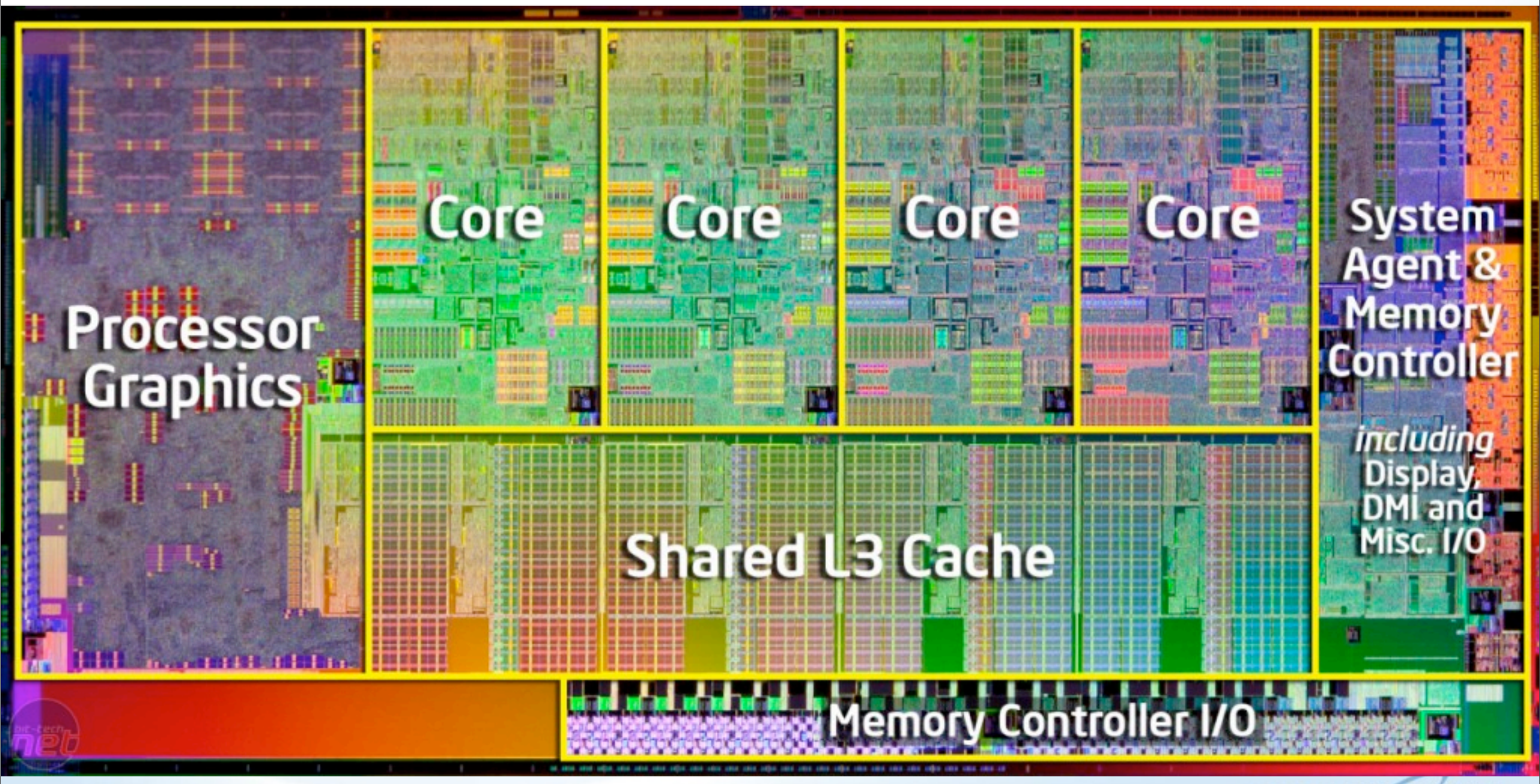
A blazing fast CPU will have the value of data[0] almost instantaneously - lets say Δt. data[N-1] is available at N Δt

A collection of m much slower processors (Δs) will take (N/m) Δs to get all of the results. If Δs < m Δt - it is a win.

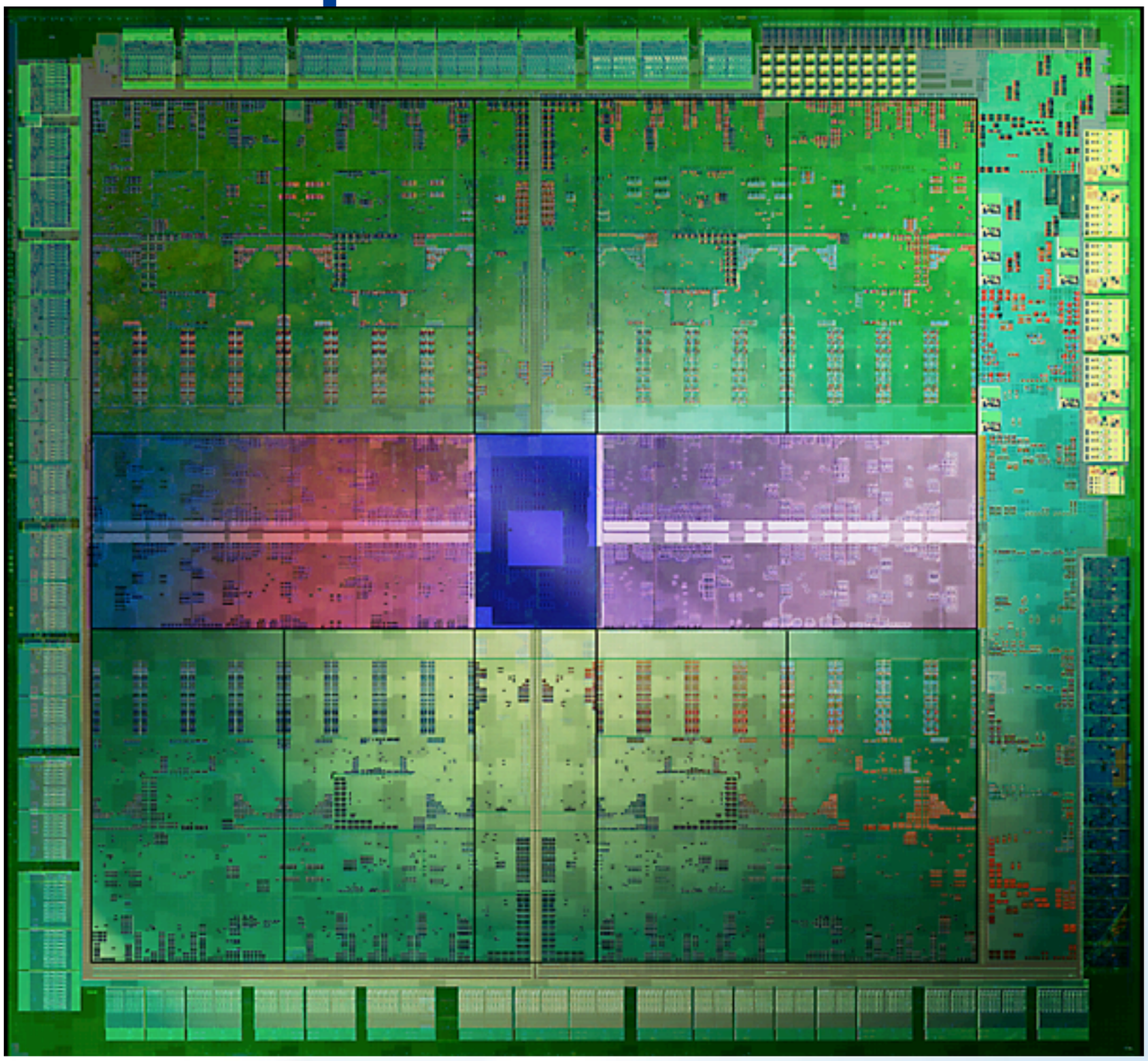
Transistor Budgets

- Modern CPUs and GPUs have billions of transistors
- However, the number of transistors needed to add two numbers has remained the same since... well, the invention of the transistor! So, what do these engineers use with all of the extra transistors?
  - In Intel's case - use them to minimize latency
  - In GPU case - use them to maximize throughput

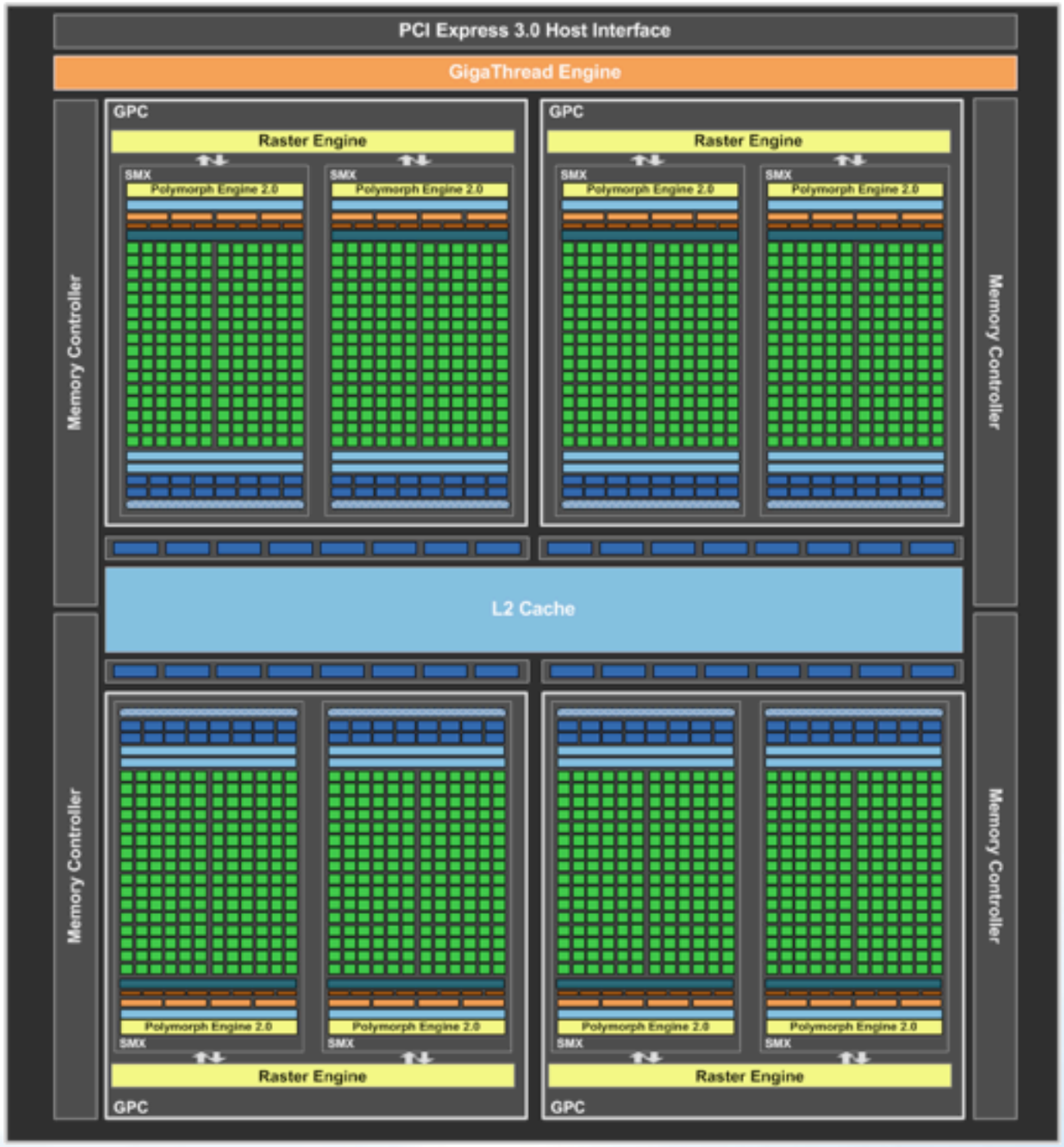
Intel Die Map



Nvidia Die Map



Nvidia die map translated



Speed-up (Amdahl's Law)

- This gives us an idea of
  - Before-hand, what is realistic to expect
  - Afterward, a measure of our success
- Suppose your program had parts that are amenable to optimization and parts that are not (serial)
  - f = fraction of time in optimized code before optimization
  - N = the speed up we can get out of that code (N=1 => no speedup)
  - Then, the over all speed up of the optimized version is given by:

$$Speedup = \frac{1}{1 - f \cdot (1 - \frac{1}{N})}$$

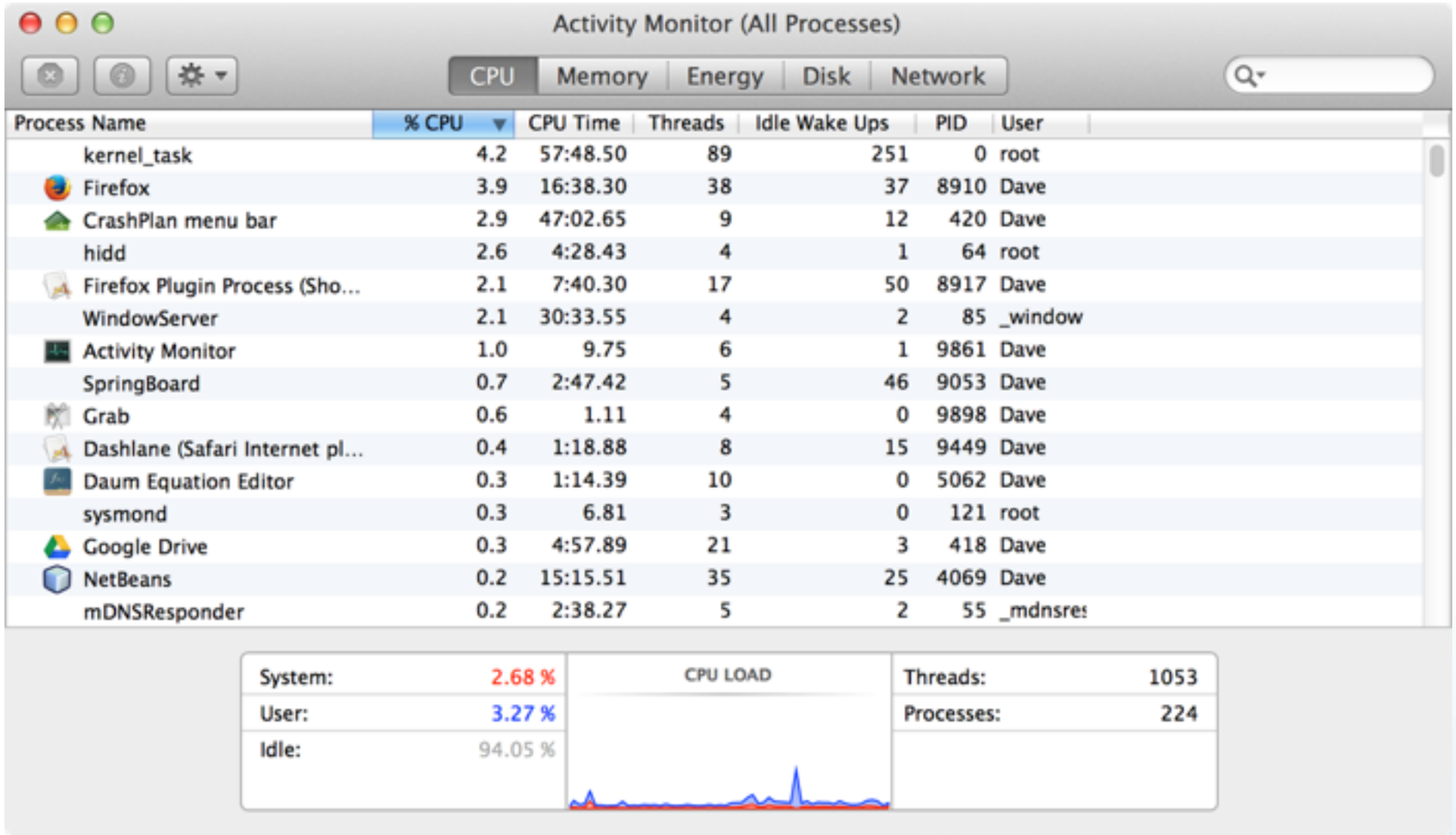
This equation is heartless!

Some (sorry) stories

f (fraction that can be optimized)	N (speedup of optimizable part)	Overall Speedup
0.5	100	1.98
0.5	1000	1.998
0.5	∞	2
0.8	100	4.8
0.8	1000	4.98
0.8	∞	5

Basically, to see a 100x speedup, you had better be able to make 99% of your code faster!

Process vs Thread



Process

- A process is (briefly):
  - A program
  - The memory state
  - The register state
- Each process has (more or less) its memory protected from every other non-root process
- A process has 1 or more threads of control or threads of execution, or simply threads.
- In *nix types of system, you create a new process with the fork system call.

Threads

- A Thread (aka “light-weight process”) is (briefly):
  - A program
  - The memory state
  - The register state
- Each thread belongs to a process and may have many siblings (other threads belonging to the same process). Each has full access to the process’ memory.
- If the computer is capable of concurrent execution of threads, they might run concurrently (needs O/S support). If not, they run in a manner similar to time-shared processes.

Threads

- A Thread (aka “light-weight process”) is (briefly):
  - A program
  - The memory state
  - The register state
- Each thread belongs to a process and may have many siblings (other threads belonging to the same process). Each has full access to the process’ memory.
- If the computer is capable of concurrent execution of threads, they might run concurrently (needs O/S support). If not, they run in a manner similar to time-shared processes.

  But each thread shares its memory (not registers) with every other thread in the process. In that is both the power and danger!

Power and Danger

- Both OpenMP and CUDA make widespread use of threads, but both try to give you access to the power while protecting you from the dangers
- Power
  - You don't have to worry about memory tables etc. when swapping between threads like you do with processes (fast!)
  - You don't have to explicitly share data - all data (except local variables) are shared - (fast and simple!)
  - If you have 8 cores and the threads can fully cooperate, then you can conceivable see an 8x speedup of the threaded code
  - Threaded code is usually concise and if the data is small, it is quite possible that all of the data and instructions are in cache (blazing fast)

Threads are easy to write hard to get right - a (not thread safe) example in Java

```
public class Account {

    private int balance;

    public Account(int initialBalance) {
        balance = initialBalance;
    }

    public void deposit(int amt) {
        balance += amt;
    }

    public void withdraw(int amt) {
        balance -= amt;
    }


    public int getBalance() {
        return balance;
    }

    public String toString() {
        return "" + balance;
    }
}
```

Collector and Depositor

```
public class Collector extends Thread {
    private Account acct;

    public Collector(Account acct) {
        this.acct = acct;
    }

    @Override
    public void run() {
        for (int i = 0; i < 1000000; i++) {
            acct.withdraw(100);
        }
        System.out.println("Collector done "
                + acct);
```

```
public class Depositor extends Thread {

    private Account acct;

    public Depositor(Account acct) {
        this.acct = acct;
    }

    @Override
    public void run() {
        for (int i = 0; i < 2000000; i++) {
            acct.deposit(100);
        }
        System.out.println("Depositor done "
                + acct);
    }
}
```

Collector and Depositor

```
public class Collector extends Thread {
    private Account acct;

    public Collector(Account acct) {
        this.acct = acct;
    }

    @Override
    public void run() {
        for (int i = 0; i < 1000000; i++) {
            acct.withdraw(100);
        }
        System.out.println("Collector done "
                + acct);
```

```
public class Depositor extends Thread {

    private Account acct;

    public Depositor(Account acct) {
        this.acct = acct;
    }

    @Override
    public void run() {
        for (int i = 0; i < 2000000; i++) {
            acct.deposit(100);
        }
        System.out.println("Depositor done "
                + acct);
    }
}
```

Collector and Depositor

```
public class Collector extends Thread {
    private Account acct;

    public Collector(Account acct) {
        this.acct = acct;
    }

    @Override
    public void run() {
        for (int i = 0; i < 1000000; i++) {
            acct.withdraw(100);
        }
        System.out.println("Collector done "
                + acct);
```

```
public class Depositor extends Thread {

    private Account acct;

    public Depositor(Account acct) {
        this.acct = acct;
    }

    @Override
    public void run() {
        for (int i = 0; i < 2000000; i++) {
            acct.deposit(100);
        }
        System.out.println("Depositor done "
                + acct);
    }
}
```

Driving Program

```
public class NotThreadSafe {

    public static void main(String[] args) throws InterruptedException {
        Account acct = new Account(100);
        Depositor p = new Depositor(acct);  // deposits $100 2,000,000 times
        Collector c1 = new Collector(acct); // withdraws $100 1,000,000 times
        Collector c2 = new Collector(acct); // withdraws $100 1,000,000 times
        c1.start(); // start 3 separate threads, all hitting same account
        p.start();
        c2.start();

        Thread.sleep(5000);    // sleep for 5 seconds
        System.out.println("In main account is " + acct);

    }
}
```

As a non-thread safe program...

One Run - a very good day for Dave!

Collector done -101947500
Collector done -101496600
Depositor done 92181300
In main account is 92181300

Another Run - a sadder day for Dave :-(

Collector done -186026100
Collector done -186024300
Depositor done -276500
In main account is -276500

Threads are not to be used thoughtlessly!

- Reading without writing (immutable data) is never a problem.
- But, in our application 3 separate threads are reading AND writing data at roughly the same times - and it is all asynchronously.

operation	thread local value	Global Account
		$100
depositor fetches & adds	$200	$100
collector fetches & subtracts	$0	$100
depositor stores	$200	$200
collector stores	$0	$0 (Should be $100!)

Sometimes, synchronization helps

```
public class Account {

    private int balance;

    public Account(int initialBalance) {
        balance = initialBalance;
    }

    public void deposit(int amt) {
        balance += amt;
    }

    public void withdraw(int amt) {
        balance -= amt;
    }


    public int getBalance() {
        return balance;
    }

    public String toString() {
        return "" + balance;
    }
}
```

```
public class Account {

    private int balance;

    public Account(int initialBalance) {
        balance = initialBalance;
    }

    public synchronized void deposit(int amt) {
        balance += amt;
    }

    public synchronized void withdraw(int amt) {
        balance -= amt;
    }


    public int getBalance() {
        return balance;
    }

    public String toString() {
        return "" + balance;
    }
}
```

But, NOT SO FAST

- The synchronized version works, but if someone is told that Account is now thread safe and writes code like:

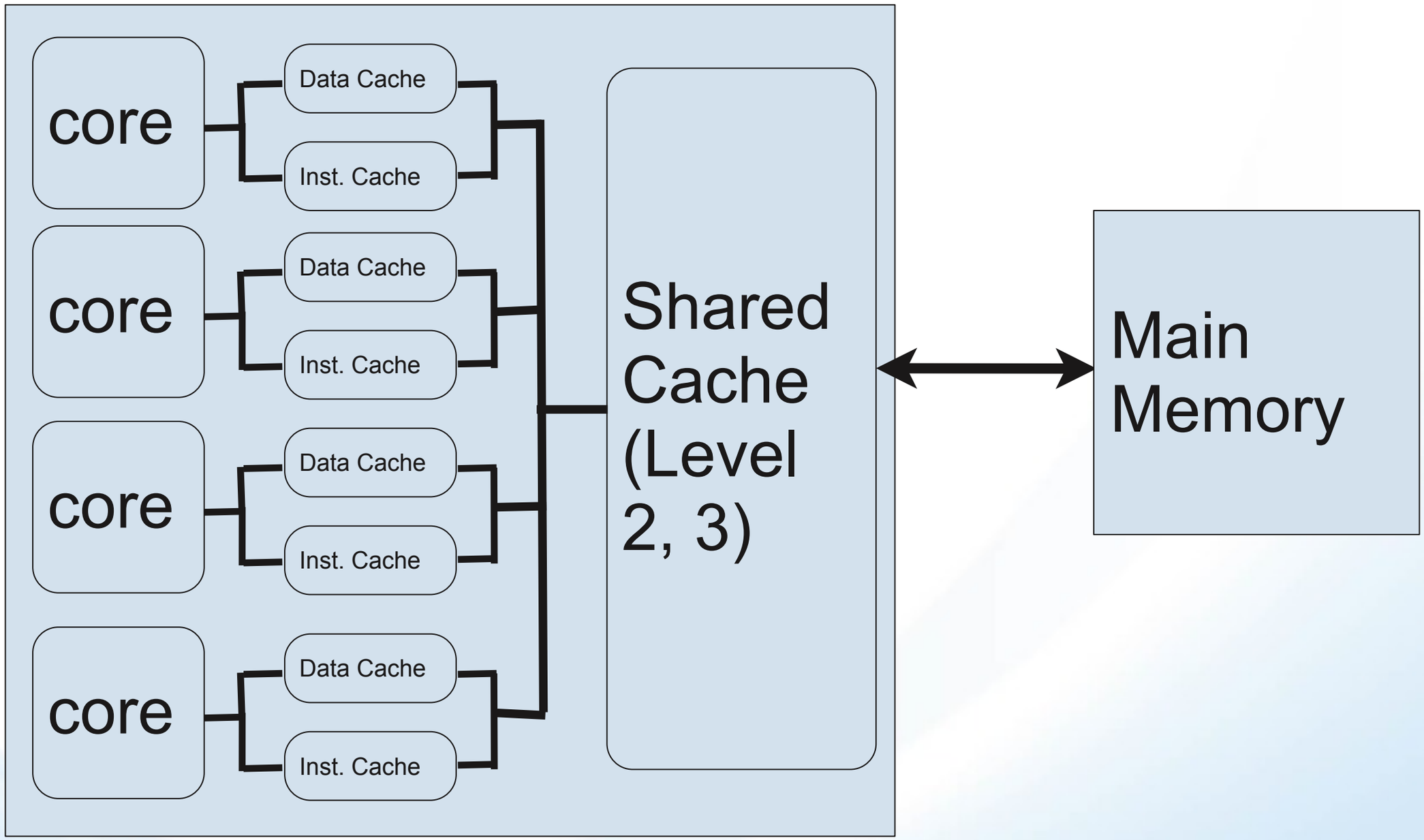
```
if (account.getBalance() > amountToWithdraw)
    account.withdraw(amountToWithdraw);
```

- Then we have broken logic even if the code is thread safe.

Nodes, Processors and Cores and HyperThreads (tm)

- The HPC Cluster that we will be using has:
  - 16 nodes (actually 17 - a head node hpc1.csc.bnl.gov and 16 “work” nodes, node01,..., node16.
  - Each node has 2 processors
    - The first 8 nodes have GPU accelerators
    - The last 8 nodes have Intel Phi accelerators (not discussed here)
  - Each processor has 8 physical cores
  - Each core is capable of running 2 threads supported by physical hardware (hyperthreads) - of course, each core can run many threads managed by software.

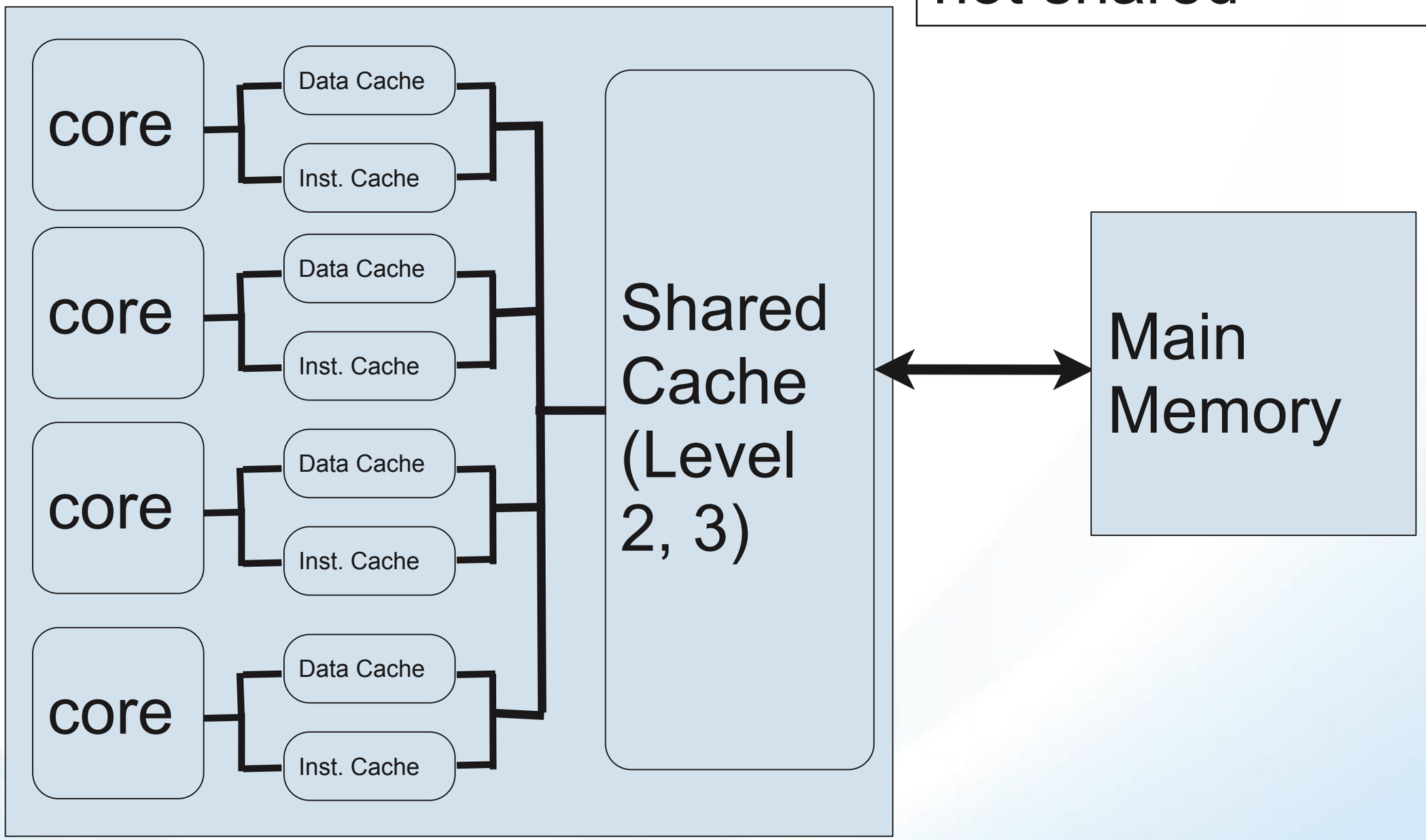
A Simplified View of a Four Core Processor



Level 1 ~ o(10K)

A Simplified View of a Four Core Processor

Level 1 Cache not shared

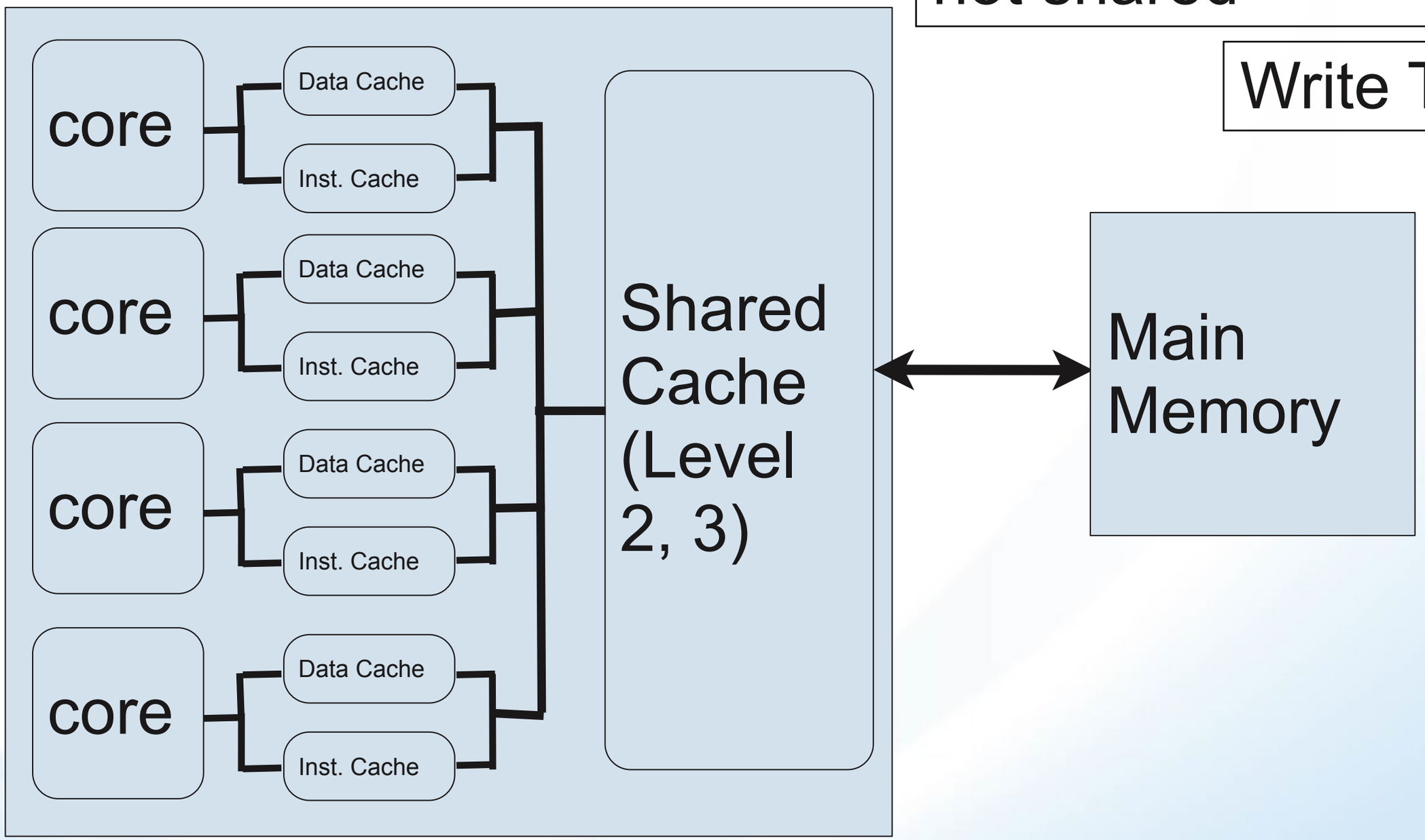


Level 1 ~ o(10K)

A Simplified View of a Four Core Processor

Level 1 Cache not shared

Write Thru

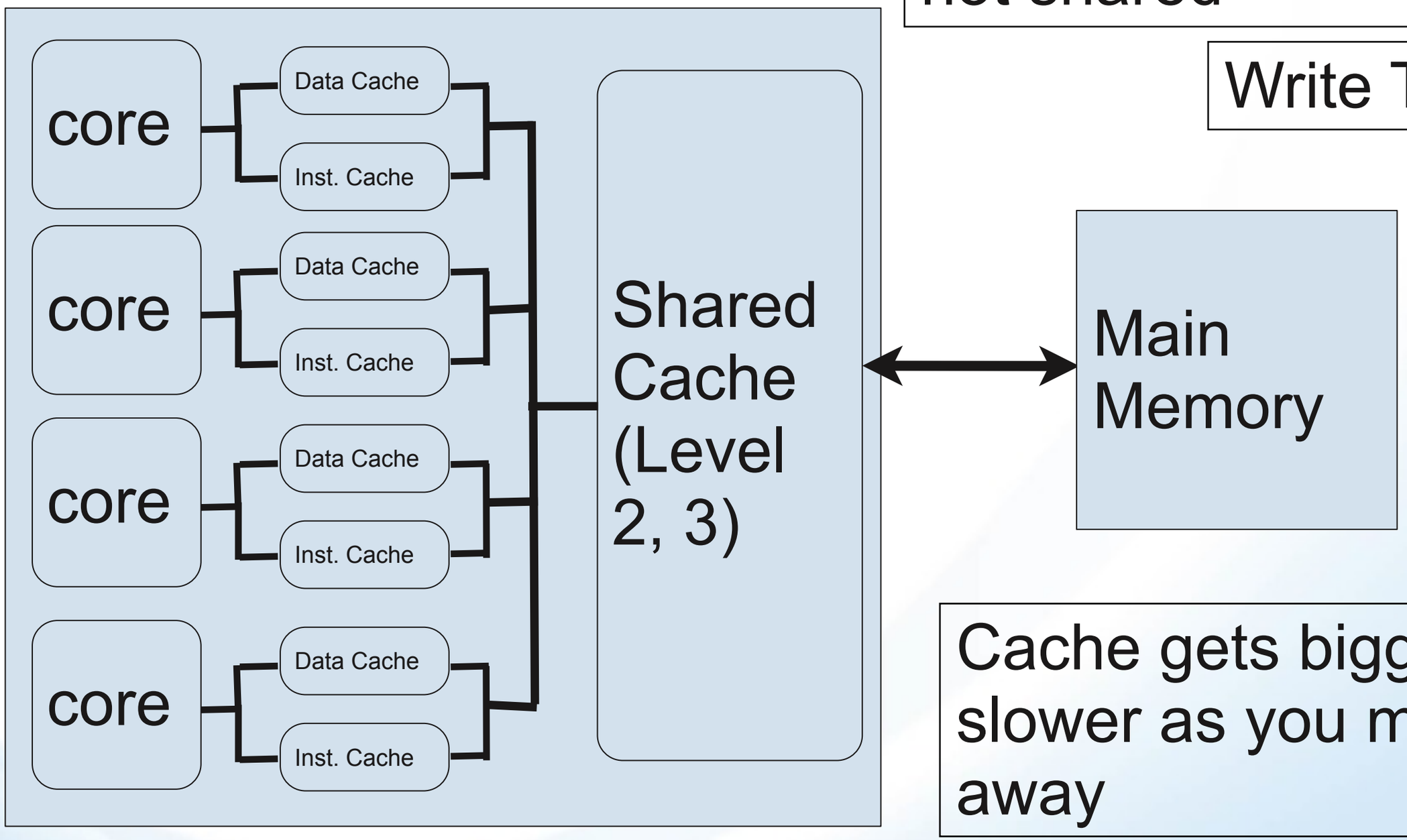


Level 1 ~ o(10K)

A Simplified View of a Four Core Processor

Level 1 Cache not shared

Write Thru



Cache gets bigger and slower as you move away

Level 1 ~ o(10K)

Hyperthreads (tm)

- Like the venerable CDC6600, modern processors have functional units that can be active (or for that matter powered off) simultaneously.
- Think of a core as a bunch of registers (state) and a bunch of functional units (behavior)
  - Registers take up very few transistors
  - Functional units use up lots of transistors
- Why not just swap out the register set when one thread has to stall ... and you have hyperthreading
- IMPORTANT - 1) this is way oversimplified and 2) as an application programmer, you don't have a lot of control over this (but learn more tomorrow!)

Parallel Programming and Algorithms

- Every Algorithms class spends time talking about O notation. The formal (math) definition:

  Let f(x) and g(x) be two functions. We say:

  $f(x) = O(g(x))$

  iff, there are positive numbers M and x0 such that

  $|f(x)| < M|g(x)|$ for all $x > x0$

- Basically, for large x, the functions have the same character

One Difference between Serial and Parallel Computing...

- In the dark arts of Algorithm design, one is mostly interested in finding an algorithm that takes a time that can be compared to a slowly growing function g(x). (x, lg(x), x lg(x), etc.) The value of M is less important and almost never talked about!

- In the world of parallel computing - we are more concerned with the value of M. Work still takes time and more work takes more time, but the time increases at a slower rate, not with a slower character.

Memory Hierarchy

- All programs have to deal with a hierarchy of memory basically layers of memory for which that access times increase along with the size.
  - The times and sizes vary with model numbers, so I won't quote numbers here other than to say that:
    - memory close to the processor (e.g. registers) can be accessed in a single clock cycle
    - RAM is much slower than cache (1000x?)
    - DISK is much much slower than RAM (hence the use of SSDs)
    - Network disk is much much slower than local disk (hence the "caching" of web pages)
- The location of data vis-a-vis the process is a critical element of performance for all parallel programming technologies - it is NOT just a compiler switch

Thinking Different

- Can you remember when this made your head hurt?

Thinking Different

- Can you remember when this made your head hurt?

$$x = x + 1$$

Thinking Different

- Can you remember when this made your head hurt?

$$x = x + 1$$

If not, maybe you learned programming before Algebra?

How about this?

```
for (int i = 0; i < 100; i++)
{
    z[i] = x[i] + y[i];
 }
```

How about this?

```
for (int i = 0; i < 100; i++)
{
    z[i] = x[i] + y[i];
}
```

Do you think vector addition or...

How about this?

```
for (int i = 0; i < 100; i++)
{
    z[i] = x[i] + y[i];
}
```

Do you think vector addition or...

do you think, “what are the other 99 values of x, y, and z doing while this big fancy computer is working with the other one?”

The fact is...

```
for (int i = 0; i < 100; i++)
{
    z[i] = x[i] + y[i];
 }
```

$$\vec{z} = \vec{x} + \vec{y}$$

The fact is...

```
for (int i = 0; i < 100; i++)
{
    z[i] = x[i] + y[i];
 }
```

Is NOT vector addition - it is an algorithm that a paltry serial processor has to follow because it is ... well ... Serial.

$$\vec{z} = \vec{x} + \vec{y}$$

The fact is...

```
for (int i = 0; i < 100; i++)
{
    z[i] = x[i] + y[i];
 }
```

Is NOT vector addition - it is an algorithm that a paltry serial processor has to follow because it is ... well ... Serial.

This is vector addition:

$$\vec{z} = \vec{x} + \vec{y}$$

One more

```
sum = 0;
for (int i = 0; i < N; i++) {
  sum = sum + x[i];
  }
```

- Seems pretty linear - every element of x has to be visited but... - Associative law to the rescue...

sum = ((((x[0] + x[1]) + x[2]) + x[3]) + x[4]) + ...

which is the same as

sum = ((x[0] + x[1]) + (x[2]+x[3])) + ...

Note about speedup

- For the previous problem - IF you have enough processors, you go from an O(N) algorithm to an O(lg(N)) algorithm! (1,000,000 -> 20)

You are invited to think different!

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!

Your options (in increasing pain and performance):

- Ignore the hype (see graph - this is not a real option)
- Buy COTS software (think Adobe) and relax
- Use libraries (cu-fftw, cublas, thrust, etc.)
- 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 workshop
- 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 (in C and Fortran) illustrating:
  - map
  - reduce (e.g., numerical integration)
  - scan (e.g., prefix sum/max/min/...)
  - Matrix Transpose (if time permits)
- Porting techniques (ongoing study)
- Resources

Reality Check

- Many schools (and MOOCs!) have semester courses on these topics while we have a few days.
  - I'm going to resist taking the conversation down a path that loses 50% of the attendees or that really requires 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 workshops

Overview - setup

Overview - setup

- Make sure your system supports the CUDA SDK (see web site at 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

Our System for this Workshop

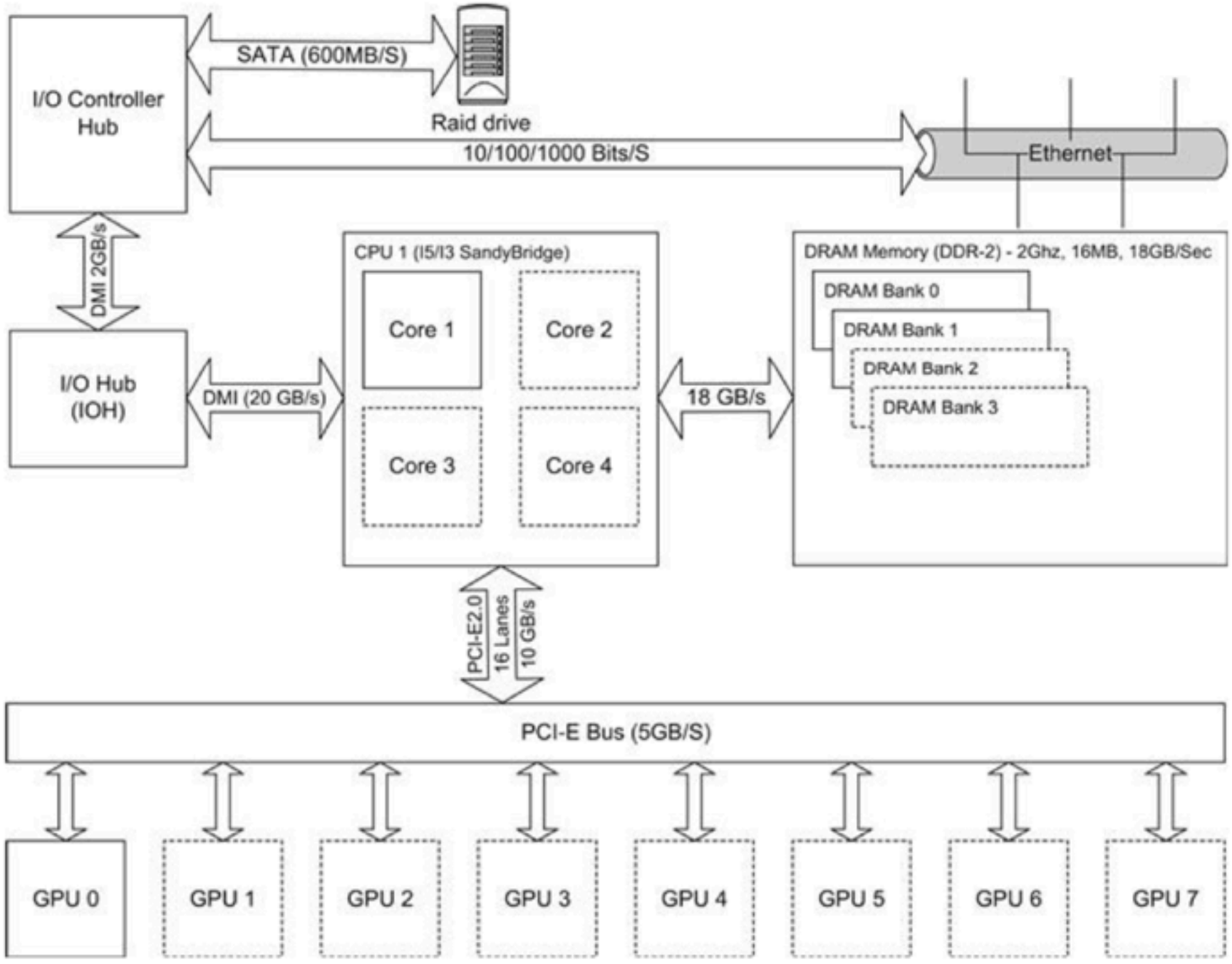
- We have a small 16 node cluster
  - Head node hpc1 which you should use
  - Computer Nodes
    - node01-node08 - have a CUDA capable nVidia board (more on this later)
    - node09-node16 - have Intel/Phi - a new architecture that we have much to learn about.
  - Queuing System
    - Torque/Maui you need to know:
    - qsub -v program="name of program to run",args="arguments to program" gpu.pbs
    - qstat to see the status of your program (you probably won't be fast enough)
  - All of today's code is or will be available on github

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)

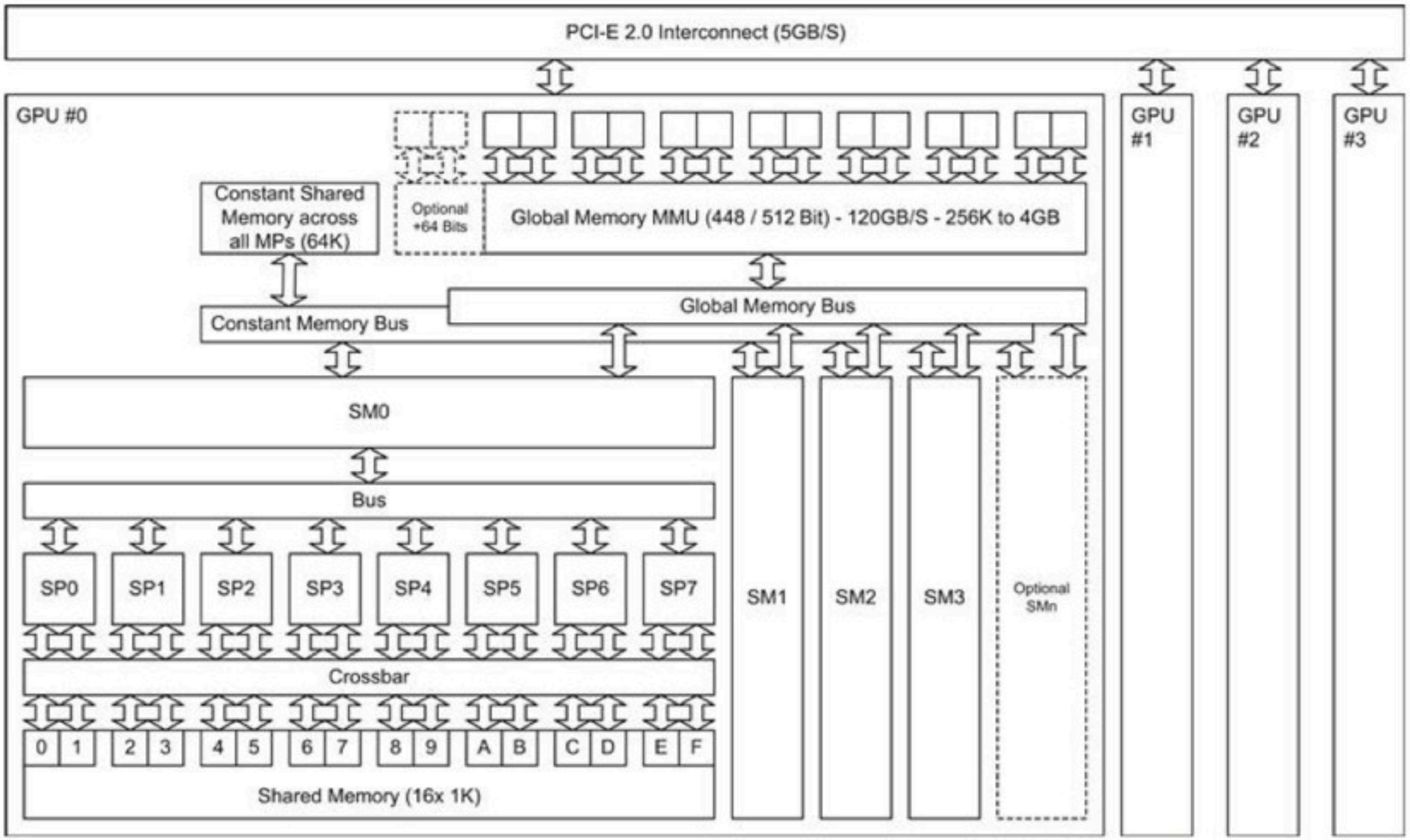


Source: CUDA Programming A Developer's Guide to Parallel Computing with GPUs by Shane Cook, Morgan Kaufmann

Our GPU instance

- Tesla K20Xm
  - 14 Multi-processors, each with 192 cores - a total of 2688 cores
  - Processor clock rate .73GHz
  - Memory clock rate 2.6GHz
  - Memory bus width 384 bits
  - L2 Cache1.5Mbytes
  - Shared Memory/Block (48K) (Yes, K)
  - Peak performance 3.52TFlop
  - Cost - $3550 from newegg.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-1/3

```
/*
 * 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-2/3

```
/* the cudaDeviceProp struct is fairly large - read 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

```
Number of cuda devices = 1
Device # 0
 name = Tesla K20Xm
 version = 3.5
 total global memory = 6039339008
 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 = 14
Max Threads/multiprocessor = 2048
```

Programming Interlude - Discovery-1/3

```
program deviceQuery

        use cudafor
        implicit none

        type(cudaDeviceProp) :: prop
        integer :: nDevices, i, ierr

        ! how many cuda devices on this node?

        ierr = cudaGetDeviceCount(nDevices)

        if (ierr .ne. cudaSuccess) then
                print *, 'Failed to get # of devices'
                stop
        else if (nDevices .eq. 0) then
                print *, 'No Cuda Devices'
                stop
        else
                print *, 'Found ',nDevices,'.'
        end if
```

Programming Interlude - Discovery-2/3

```
do i = 0, nDevices-1

    ierr = cudaGetDeviceProperties(prop,i)
    if (ierr .ne. cudaSuccess) then
        print *, 'Failed to get device properties'
        stop
    endif
    print *, "Device # ",i
    print *, " name = ",trim(prop%name)
    print *, " version = ",prop%major,".",prop%minor
    print *, " total global memory = ",prop%totalGlobalMem
    print *, " shared Memory/Block = ",prop%sharedMemPerBlock
    print *, " registers/block = ",prop%regsPerBlock
    print *, " warp size = ",prop%warpSize
    print *, " Max threads/block = ",prop%maxThreadsPerBlock
    print *, " Max Threads Dim = ",prop%maxThreadsDim(0) ," x ", &
                          prop%maxThreadsDim(1)," x ",prop%maxThreadsDim(2)
    print *, " Max Grid Size = ",prop%maxGridSize(0)," x ", &
                         prop%maxGridSize(1)," x ",prop%maxGridSize(2)
    print *, " Multi-processor count = ",prop%multiProcessorCount
    print *, " Max Threads/multiprocessor=",prop%maxThreadsPerMultiProcessor
end do
end program deviceQuery
```

Programming Interlude - output

```
Found                1 .
Device #                0
 name = Tesla K20Xm
 version =                3 .              5
 total global memory =                        6039339008
 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 =                14
Max Threads/multiprocessor =             2048
```

Overview - Software

Programming CUDA

- In the bad old days, programming your GPU meant that you had to cast your problem as a graphics manipulation - probably using openGL. 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

Rinse and repeat

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] = ...
```

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] = ...
```

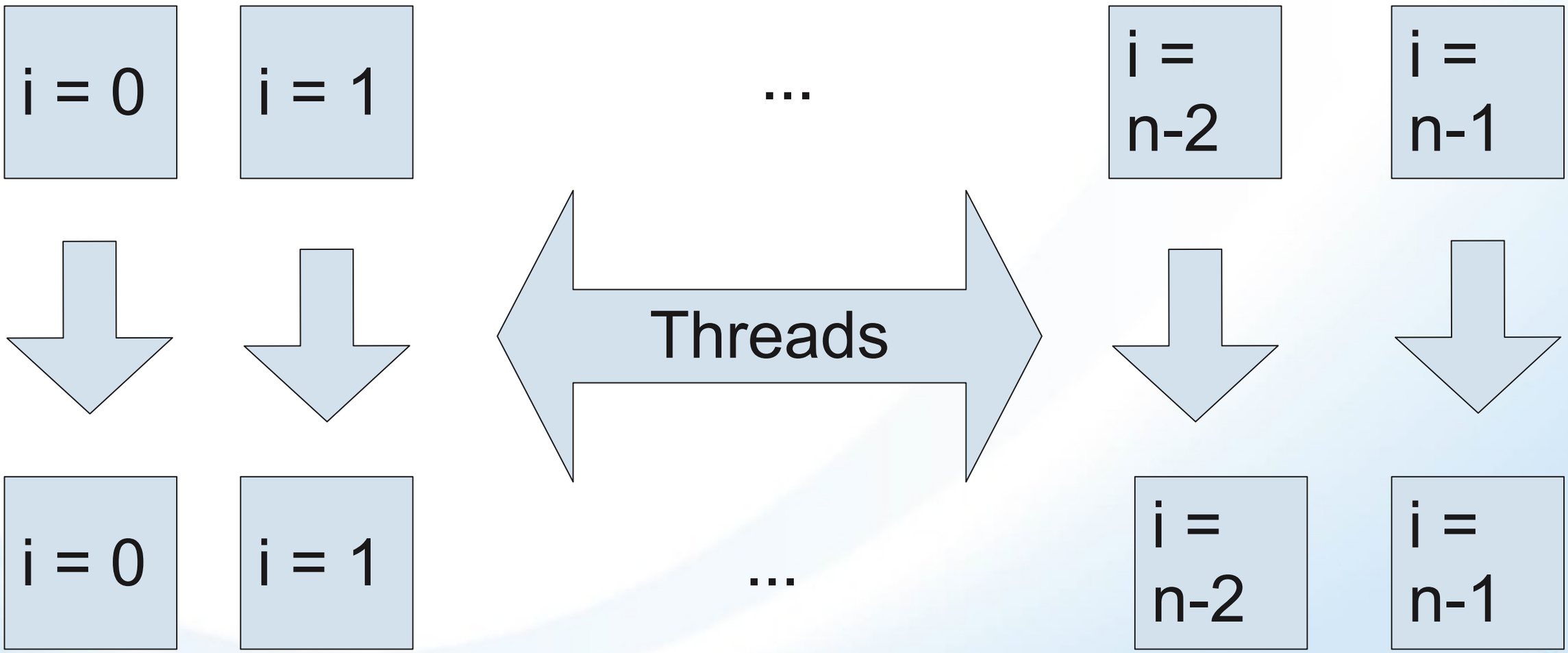
But you only have to write 1 of these!

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.
- If Fortran is your language, a compiler like the PGI Fortran will do the same

Lets get an example in mind...

- Consider a program that has to simply run through a 1-D array and recompute its value: (Map)



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.

What if you don't have enough threads? (array > 2048 elements?)

- This is where blocks come into play:

Block 1 (with 2048 threads)	Block 2 (with 2048 threads)	...	Block m-2 (with 2048 threads)	Block m-1 (with 2048 threads)

It is the job of the GPU to run these blocks as best it can. No order is specified. There is no simple communication between blocks.

Remember this?

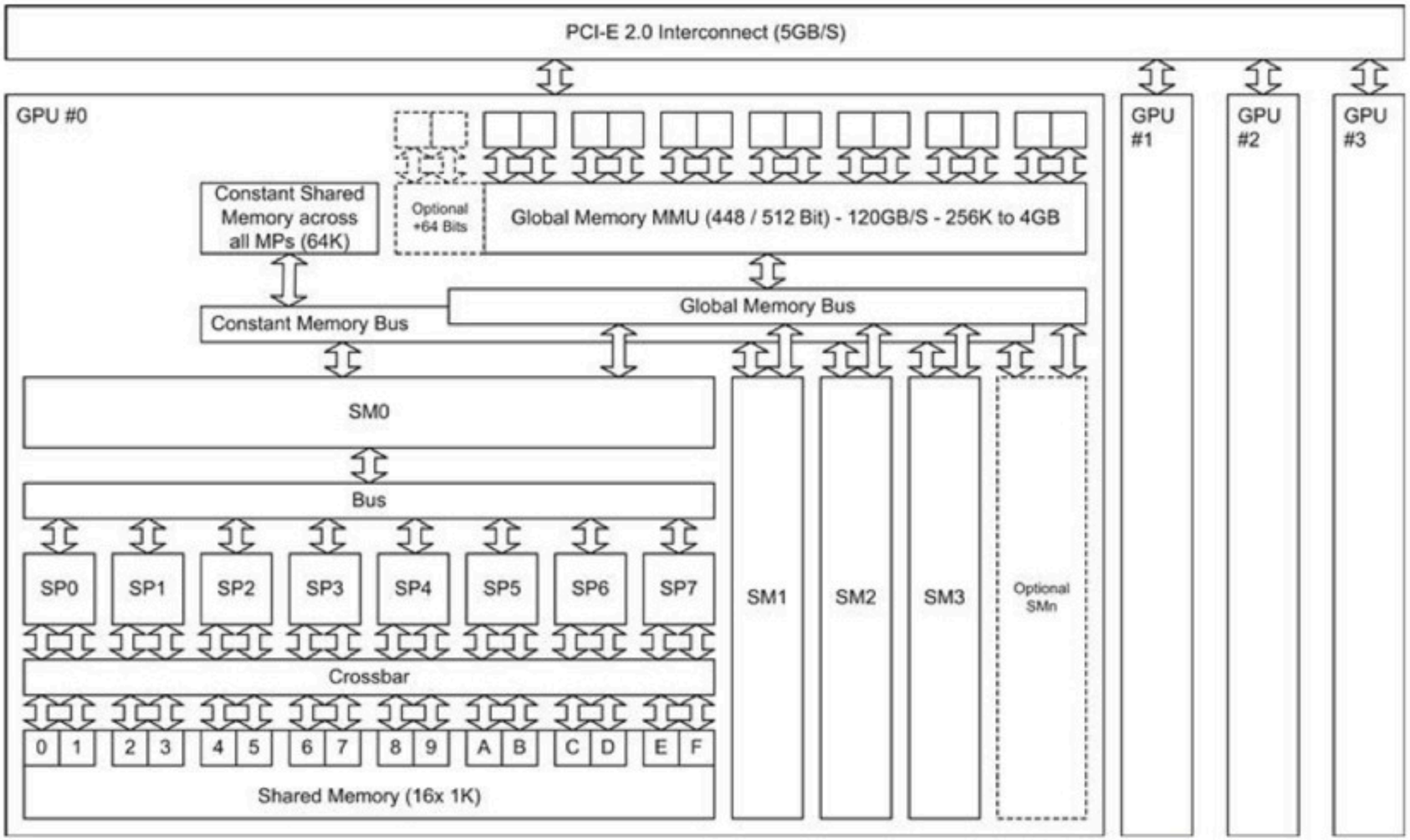
```
Max threads/block =           1024
Max Threads Dim =          1024  x          1024  x             64
Max Grid Size =    2147483647  x         65535  x          65535
```

- Blocks can run up to 1024 threads
- You can treat the threads running in a block as a 1, 2 or 3 dimensional array of blocks - to total however is limited (e.g.1024x1x1 or 32x32x1 or 16x16x4) - it is all just bookkeeping
- You create a grid of blocks which have a nearly unlimited size in 1, 2 or 3 dimensions.

Overview - Programming Model First Pass - Blocks

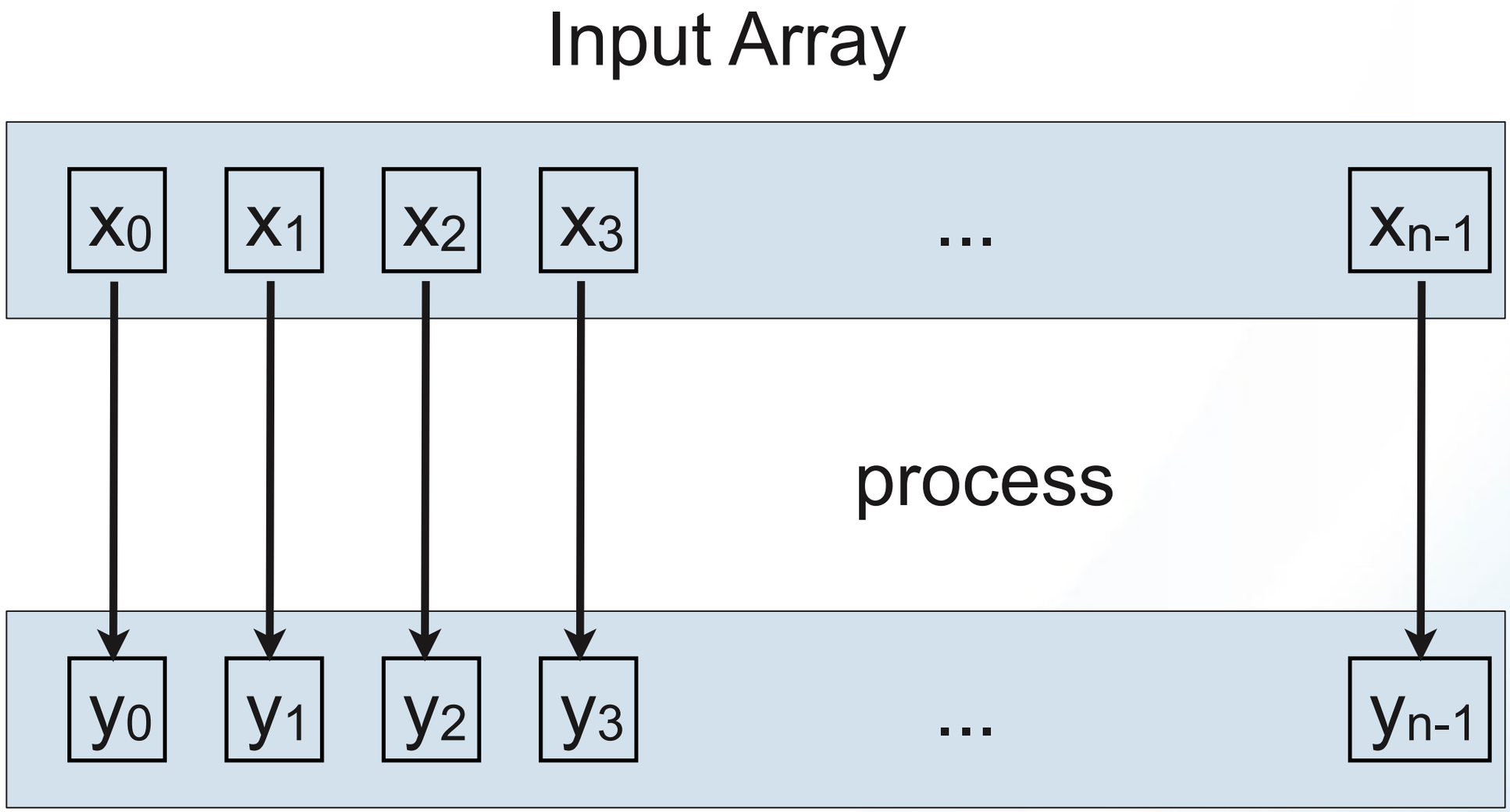
- A Block is a bunch of threads (up to 1024 on modern devices)
- When you launch a kernel, you create 1 or more blocks.
  - It is your choice for the number of blocks and the number of threads/block - you choose to fit the problem & for performance
- The block is the unit of scheduling for the GPU
  - 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?

If you don’t have enough what is the “O” speed?

Code Interlude - sequential map 1/2

```
/*
 * Map the square function sqeezing all of this data through 1 CPU.
 */

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

void nonCudaMap(float* out, float *in, int size);
void startClock(char*);
void stopClock(char*);
void printClock(char*);

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


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

Code Interlude - sequential map 2/2

```
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, *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");
}
```

Code Interlude - CUDA map (host) 1/3

```
/*
 * A very simple cuda implementation of map
 */

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

__global__ void map(float* out, float* in, int size);

void startClock(char*);
void stopClock(char*);
void printClock(char*);

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);
   }
```

Code Interlude - CUDA map (host) 2/3

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

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");
```

Code Interlude - CUDA map (host) 3/3

```
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)

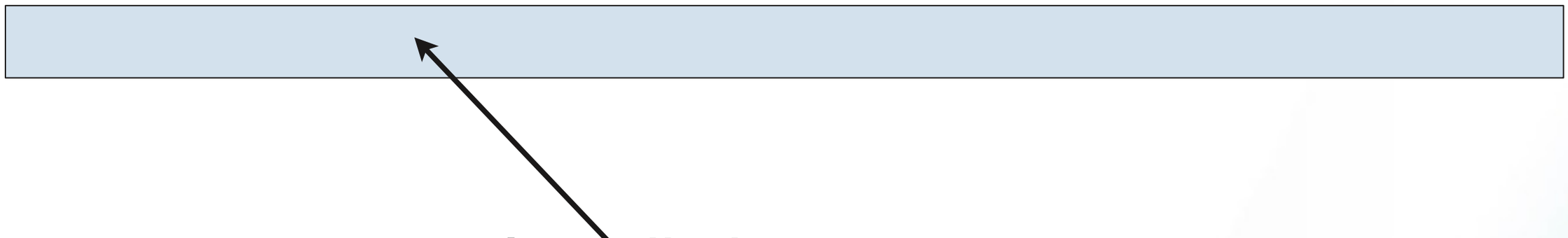
```
/*
 * squaring map kernel that runs in 1 block
 */

/*
 * 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

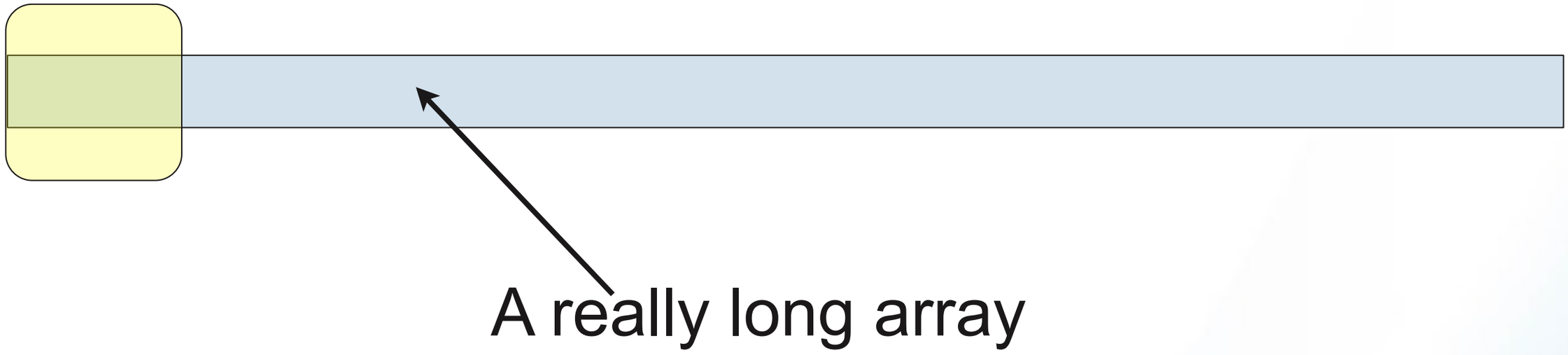
Size of array	Sequential Time in micro seconds	CUDA Data Copy Time in micro seconds	Cuda Compute Time in micro seconds
256	10	79	200
512	13	80	245
1024	20	84	204

So, How to deal with arrays > 1024?



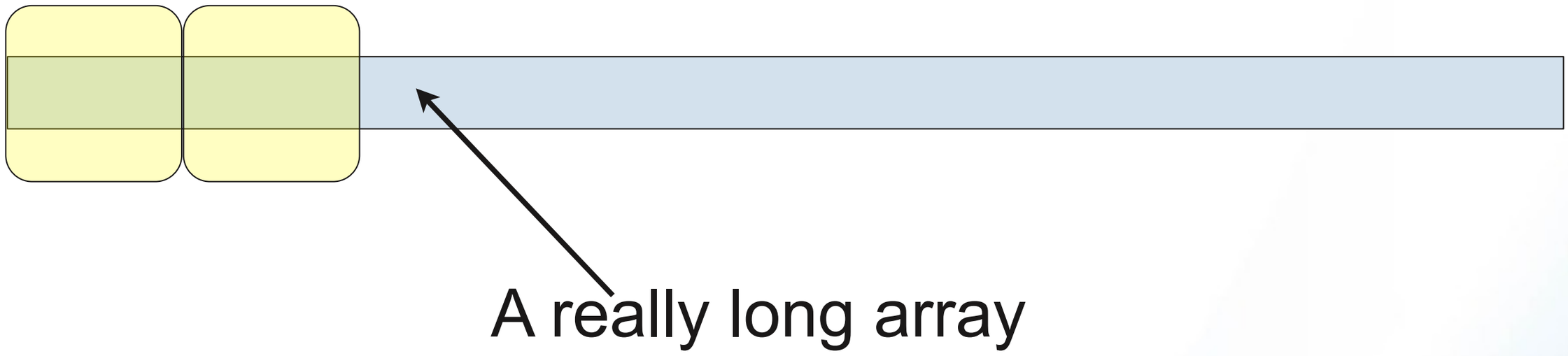
Then, (somehow) each kernel knows which block it belongs to and which thread within each block, so it can easily compute it’s index into the array.

So, How to deal with arrays > 1024?



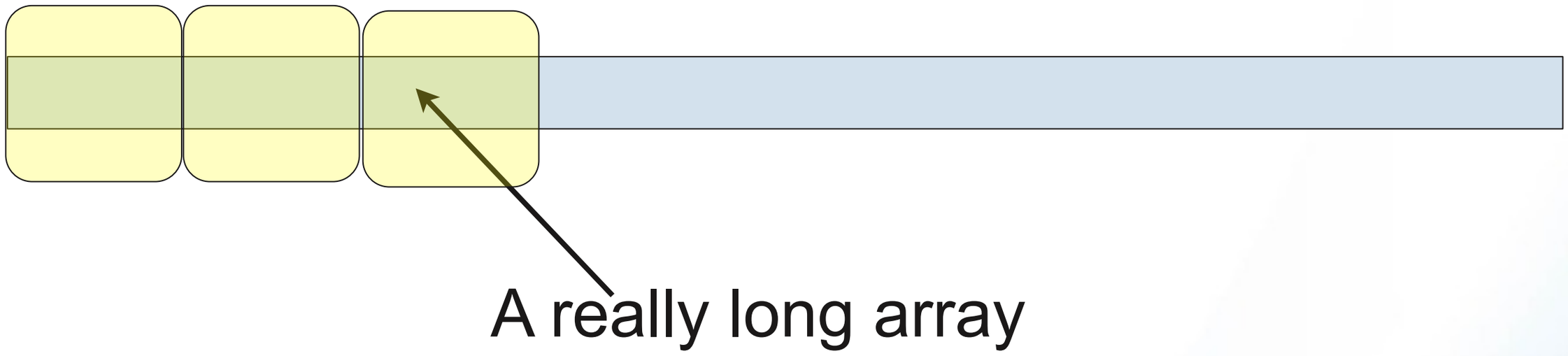
Then, (somehow) each kernel knows which block it belongs to and which thread within each block, so it can easily compute it’s index into the array.

So, How to deal with arrays > 1024?



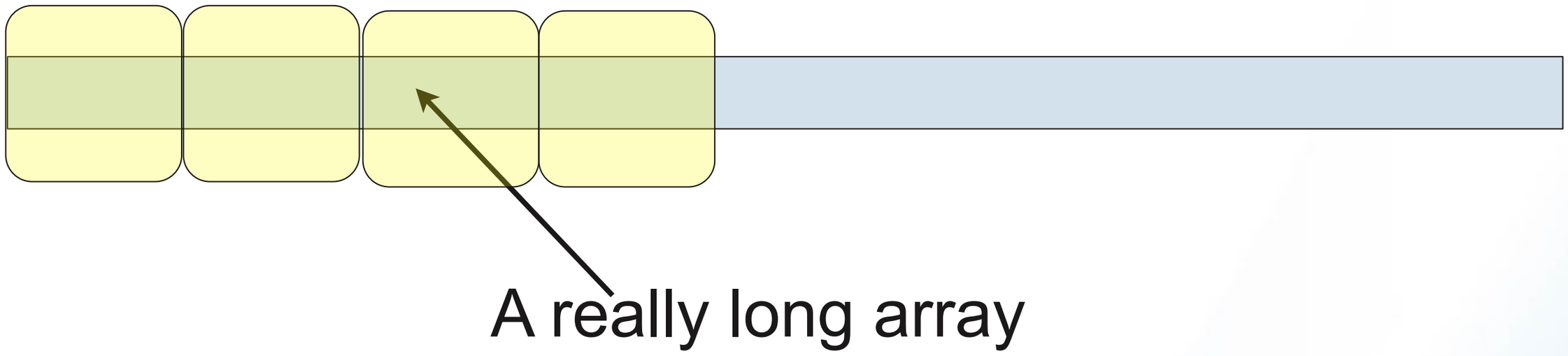
Then, (somehow) each kernel knows which block it belongs to and which thread within each block, so it can easily compute it’s index into the array.

So, How to deal with arrays > 1024?



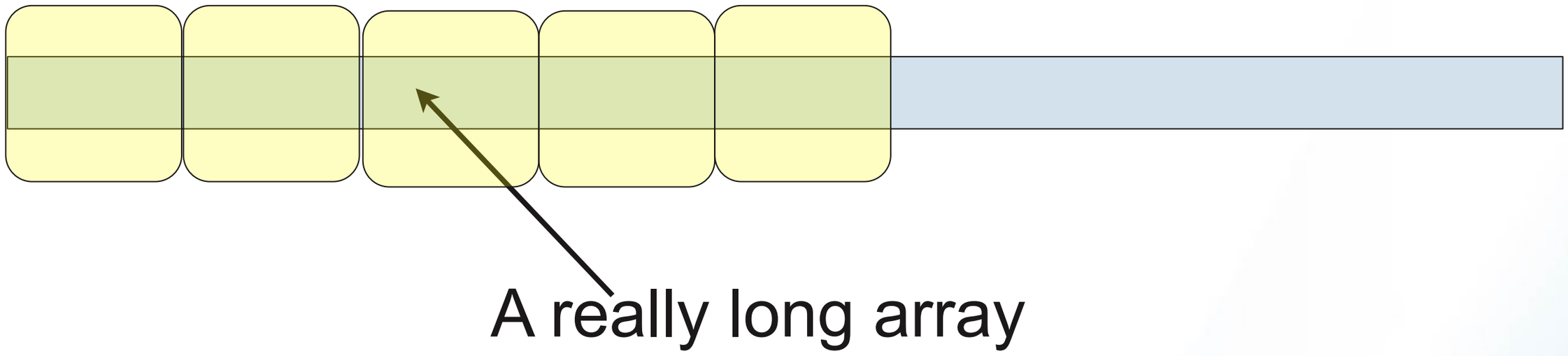
Then, (somehow) each kernel knows which block it belongs to and which thread within each block, so it can easily compute it’s index into the array.

So, How to deal with arrays > 1024?



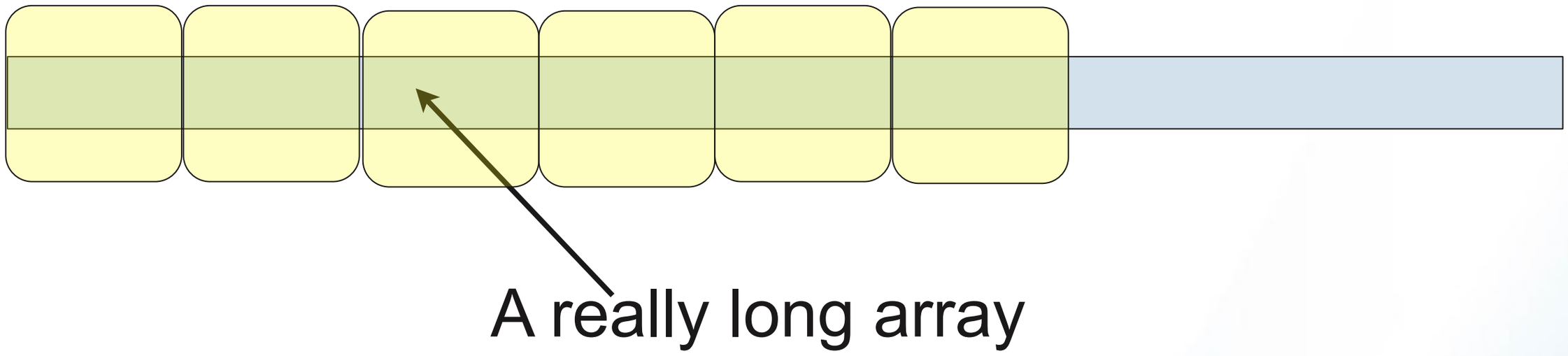
Then, (somehow) each kernel knows which block it belongs to and which thread within each block, so it can easily compute it’s index into the array.

So, How to deal with arrays > 1024?



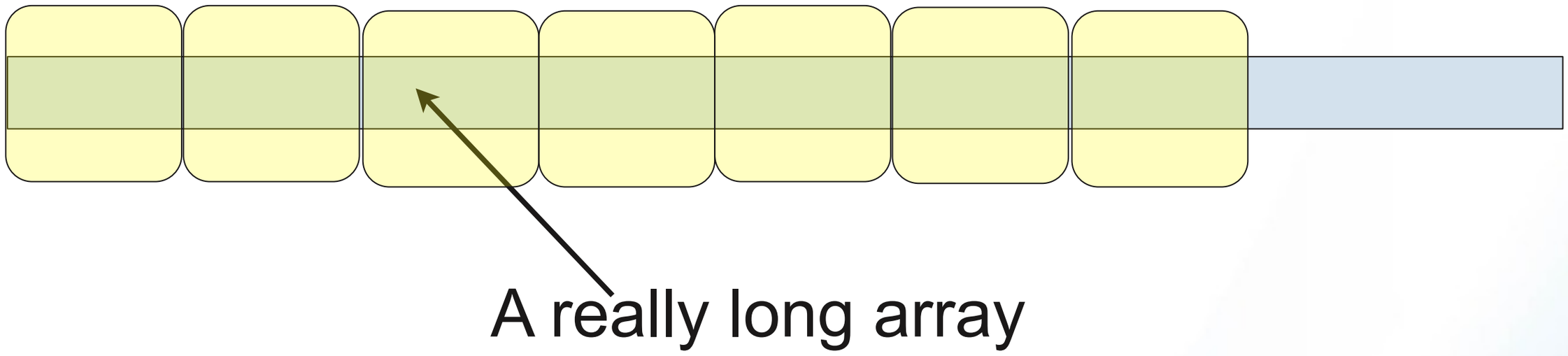
Then, (somehow) each kernel knows which block it belongs to and which thread within each block, so it can easily compute it’s index into the array.

So, How to deal with arrays > 1024?



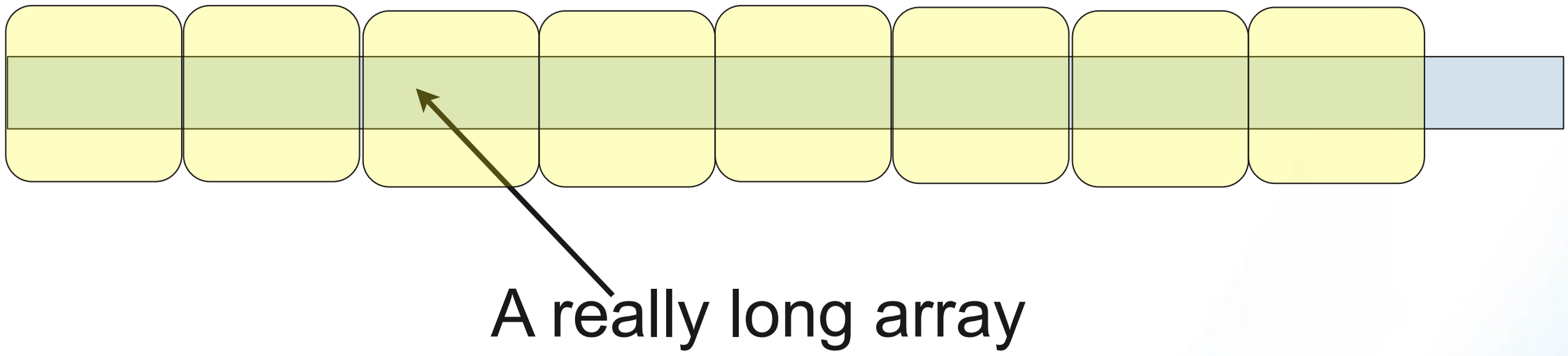
Then, (somehow) each kernel knows which block it belongs to and which thread within each block, so it can easily compute it’s index into the array.

So, How to deal with arrays > 1024?



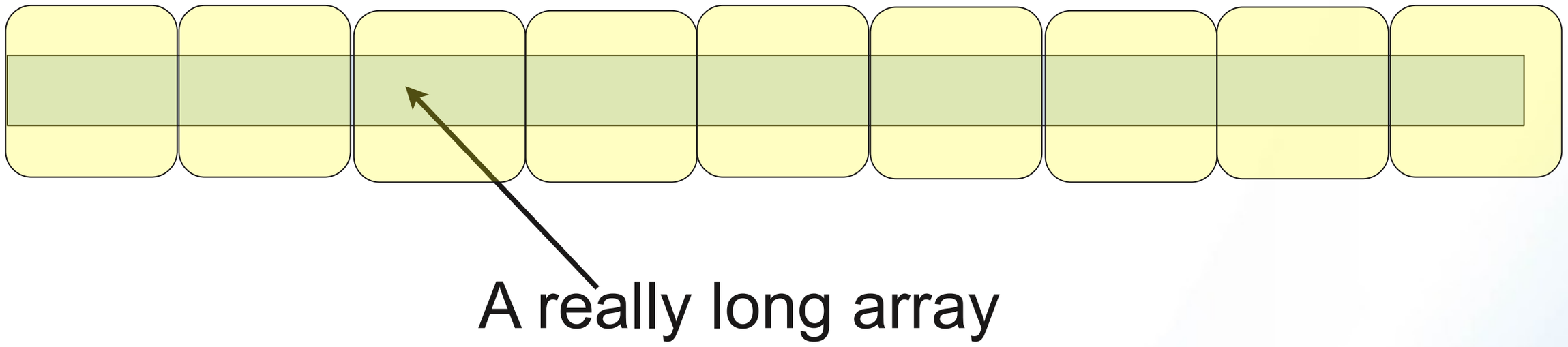
Then, (somehow) each kernel knows which block it belongs to and which thread within each block, so it can easily compute it’s index into the array.

So, How to deal with arrays > 1024?



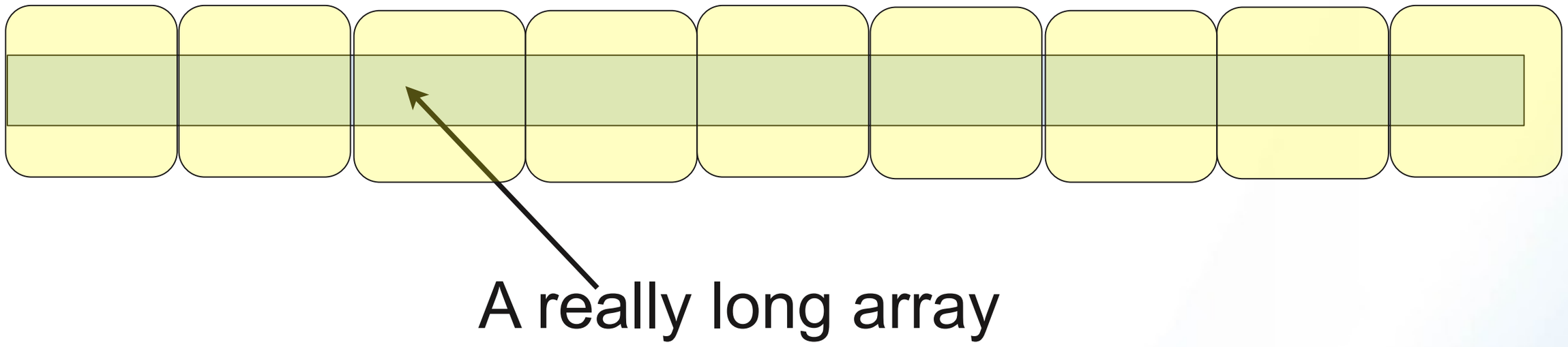
Then, (somehow) each kernel knows which block it belongs to and which thread within each block, so it can easily compute it’s index into the array.

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.

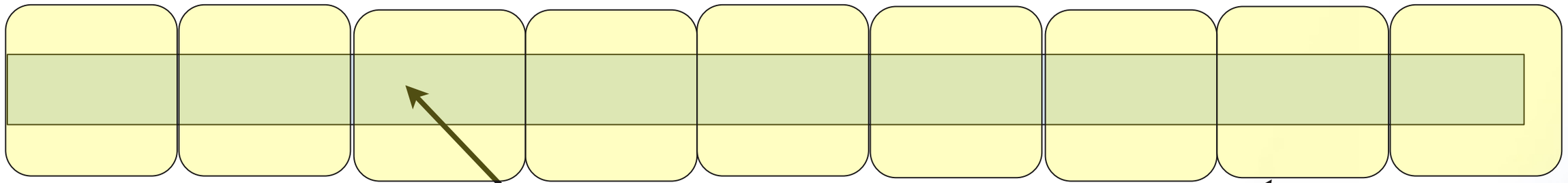
So, How to deal with arrays > 1024?



Blocks, each with 1024 threads (last might be short)...

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?

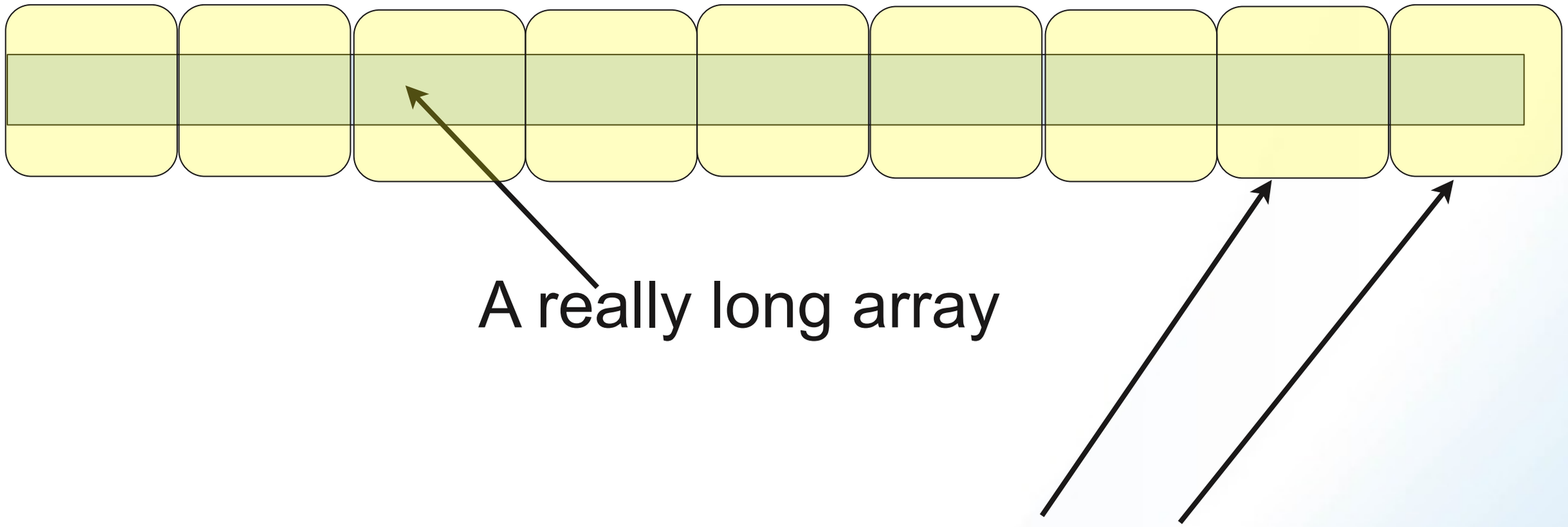


A really long array

Blocks, each with 1024 threads (last might be short)...

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.

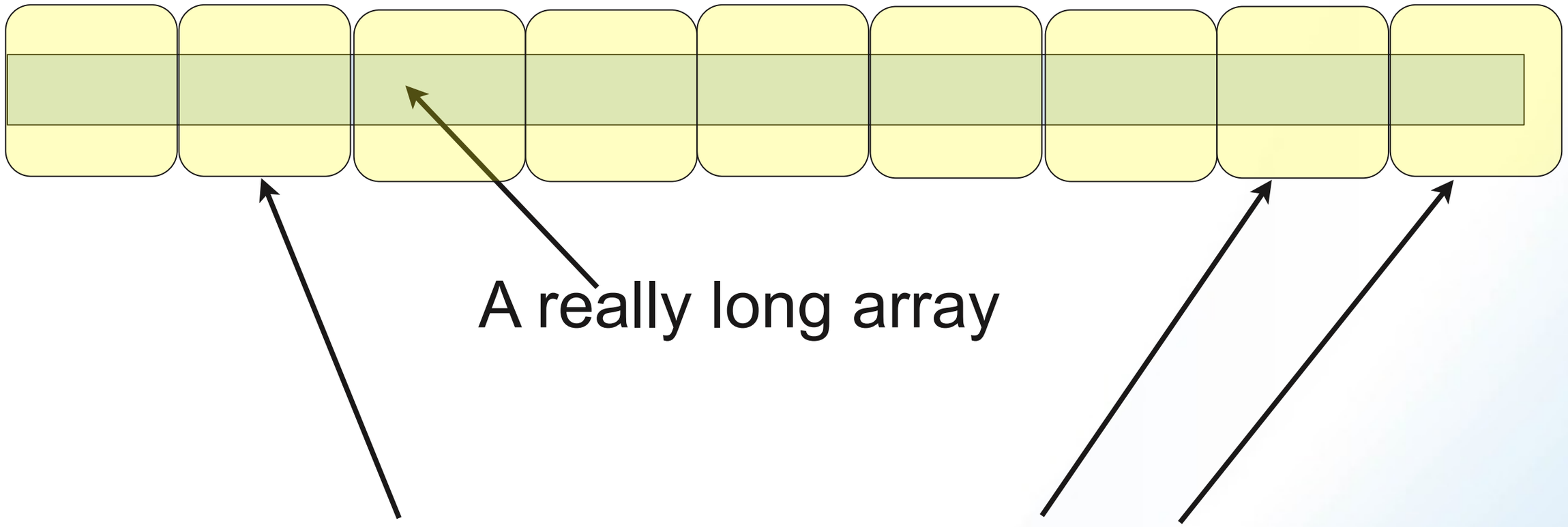
So, How to deal with arrays > 1024?



Blocks, each with 1024 threads (last might be short)...

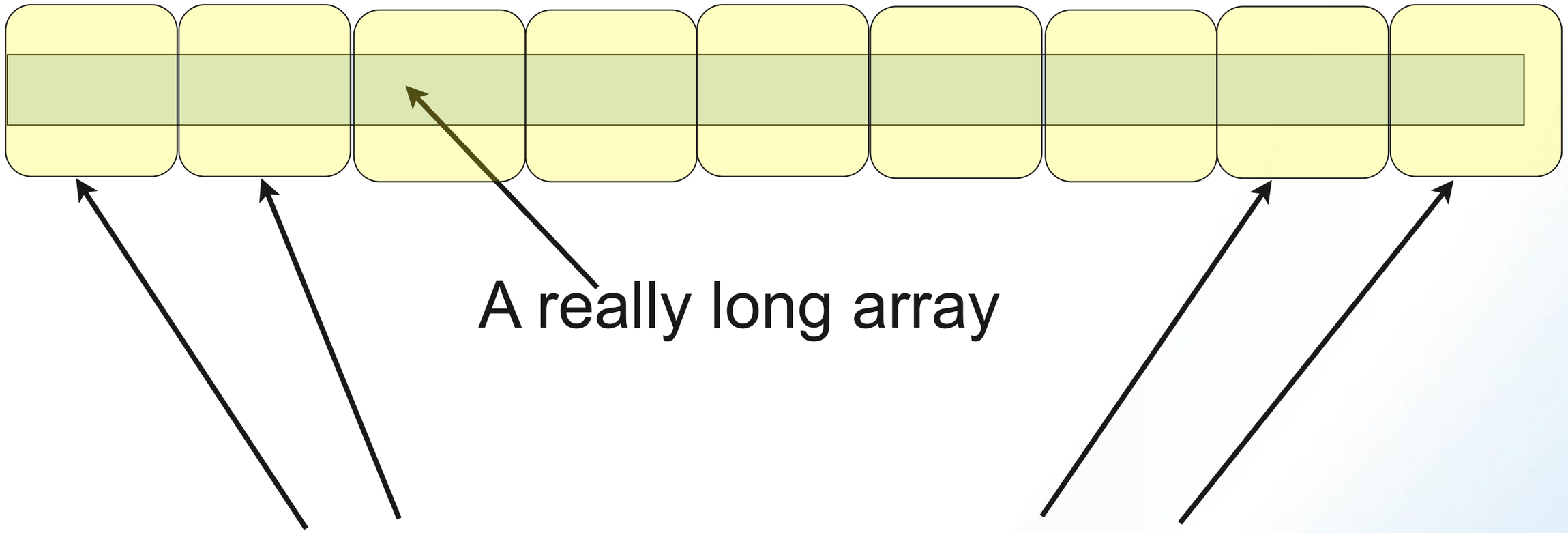
Then, (somehow) each kernel knows which block it belongs to and which thread within each block, so it can easily compute it's index into the array.

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 threads (last might be short)...

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

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	10	79	200
512	13	80	245
1024	20	84	204
2048	36	90	204
4096	69	115	202
8192	129	153	205
16384	271	214	211
32768	530	346	227
65536	1060	621	251

For completeness, Fortran versions (Serial) (1/3)

```
!
! A serial map program
!
! Map the square function sqeezing all of this data through 1 CPU.


subroutine nonCudaMap(in,out,size)

real, dimension(1:size), intent(in) :: in
real, dimension(1:size), intent(out) :: out
integer, intent(in) :: size

        do i = 1, size
                out(i) = in(i)*in(i)
        end do

end subroutine nonCudaMap
```

For completeness, Fortran versions (Serial) (2/3)

```
program SerialMap

        implicit none
        integer i, size, iargc
        character(len=32) :: arg
        real, dimension(:), allocatable :: h_in, h_out

        if (iargc() .ne. 1) then
                print *,'Usage: map-non-cuda #'
                stop
        end if

        call getarg(1,arg)
        read(arg,*) size

        allocate(h_in(1:size));
        allocate(h_out(1:size));

        do i = 1, size
                h_in(i) = i
        end do
```

For completeness, Fortran versions (Serial) (3/3)

```
        call nonCudaMap(h_in, h_out,size)

        do  i = 1, size
                print *,i, h_in(i), h_out(i)
        end do

        deallocate(h_in);
        deallocate(h_out);

end program SerialMap
```

The small cuda map version (1/4)

```
! A very simple CUDA implementation of Map

module map_module
contains

attributes(global) subroutine map(in,out,size)
        implicit none
        real, dimension(:), intent(in) :: in
        real, dimension(:), intent(out) :: out

        integer, value :: size
        integer i

        i = threadIdx%x
        out(i) = in(i)*in(i)

end subroutine map

end module map_module
```

The small cuda map version (2/4)

```
program Map_Cuda_Small

        use cudafor
        use map_module

        implicit none
        integer size, i, iargc, ierr
        character (len=32) arg
        type(cudaDeviceProp) :: prop

        real, dimension(:), allocatable :: h_in, h_out
        real, dimension(:), allocatable, device :: d_in, d_out

        if (iargc() .ne. 1) then
                print *, 'Usage map-cuda-small #'
                stop
        end if

        call getarg(1,arg)
        read(arg,*) size
```

The small cuda map version (3/4)

```
!
! make sure size is <= 1024 for the small case
!

        ierr = cudaGetDeviceProperties(prop,0)
        if (prop%maxThreadsPerBlock .lt.size) then
                print *, 'Size must be <= ',prop%maxThreadsPerBlock
                stop
        endif

        allocate(h_in(1:size))
        allocate(h_out(1:size))
        allocate(d_in(1:size))
        allocate(d_out(1:size))

        do i = 1, size
                h_in(i) = i
        end do

        ! transfer data to device

        d_in = h_in
```

The small cuda map version (4/4)

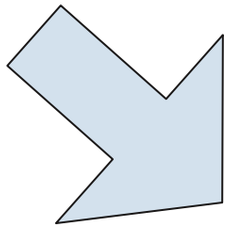
```
! Launch the kernel

call map<<<1,size>>>(d_in, d_out, size)

! transfer data back to host

h_out = d_out

do i = 1, size
        print *, i, h_in(i), h_out(i)
end do

deallocate(h_in, h_out, d_in, d_out)

end program Map_Cuda_Small
```

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 (e.g., gcc -fprofile-arcs -ftest-coverage along with tcov)
  - gpu (production)
  - 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

A Porting Example - here is the critical piece from tcov

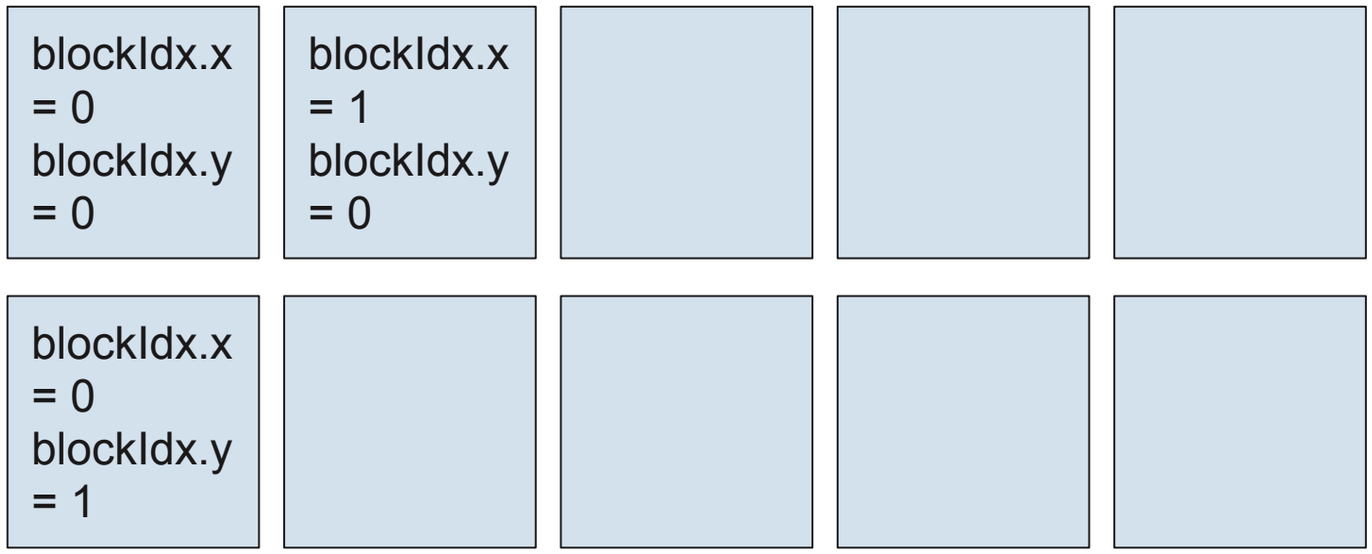
```
    3610:  142:       for (o=0; o<pixel_n; o++)^M
       -:  143:       {
 1299600:  144:           for (p=0; p<pixel_n; p++)^M
       -:  145:           {
       -:  146:
955152000:  147:               for(a=0; a<M; a++)^M
       -:  148:               {
       -:  149:
3815424000:  150:                   Amp = Amp + cexp(2*PI*I* (Qx[o*pixel_n
+p]*U_L[a] + Qy[o*pixel_n+p]*V_L[a] + Qz[o*pixel_n+p]*W_L[a]));^M
       -:  151:
       -:  152:               }
       -:  153:
 3888000:  154:               Int_2D[o*pixel_n+p] = Int_2D[o*pixel_n+p] +
(Amp* conj(Amp));
 1296000:  155:                   Amp = 0;
       -:  156:           }
       -:  157:       }
```

Analysis

- We are filling in the values of a 2D array, Int_2D.
  - First guess - lets use one thread per element of Int_2D
  - That means our grid and our thread blocks should be 2D arrays.
  - The overall size of the array is pixel_n x pixel_n.
  - Threads - we are allow 1024 threads per block - 32x32 is 1024
  - So, lets use thread blocks that are 32x32 and lets use enough blocks of that size to cover the array (opixel_n + 31)/32 square.
  - Each kernel computes the index of the thread that it is interested in via:

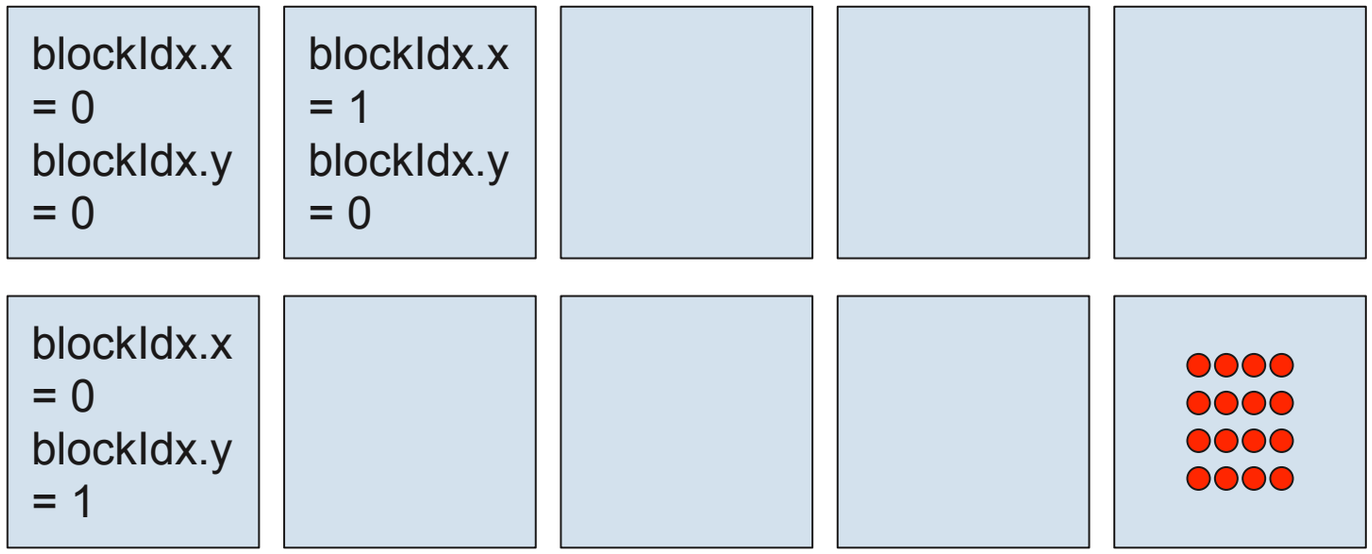
column-index =threadIdx.x + blockIdx.x*blockDim.x
row-index = threadIdx.y + blockIdx.y*blockDim.y

Blocks & Threads



Grid is a 5x2 collection of blocks
gridDim = (5,2,1). Each block has a unique (for the grid) blockIdx.x and blockIdx.y

Blocks & Threads



Grid is a 5x2 collection of blocks
gridDim = (5,2,1). Each block has a unique (for the grid) blockIdx.x and blockIdx.y

Blocks & Threads

Grid is a 5x2 collection of blocks
gridDim = (5,2,1). Each block has a unique (for the grid) blockIdx.x and blockIdx.y

Each Block has a 4x4 matrix of threads, each thread as a unique (for the block) threadIdx.x and threadIdx.y.
Here, blockDim = (4,4,1)

The critical section transforms from:

```
for (o=0; o<pixel_n; o++)
{
    for (p=0; p<pixel_n; p++)
    {

        for(a=0; a<M; a++)
        {

                Amp = Amp + cexp(2*PI*I* (Qx[o*pixel_n+p]*U_L[a] + Qy[o*pixel_n
+p]*V_L[a] + Qz[o*pixel_n+p]*W_L[a]));

        }

        Int_2D[o*pixel_n+p] = Int_2D[o*pixel_n+p] + (Amp* conj(Amp));
        Amp = 0;
    }
}
```

To the Kernel:

```
__global__ void cudaComp(double* Qx,double* Qy,double* Qz,double* U_L,double*
V_L,double* W_L,double* Int_2D,int pixel_n,int M) {

        // compute the value for o and p based upon the blockIdx, blockIdy,
threadIdx, threadIdy
        int p = threadIdx.x + blockIdx.x*blockDim.x;
        int o = threadIdx.y + blockIdx.y*blockDim.y;
        if (o >= pixel_n || p >= pixel_n) return;
        double qx, qy, qz;
        qx = Qx[o*pixel_n + p];
        qy = Qy[o*pixel_n + p];
        qz = Qz[o*pixel_n + p];

        cuDoubleComplex Amp = make_cuDoubleComplex(0,0);
        double PI = 4.*atan(1.0);
        for (int a = 0; a < M; a++) {
                double exp = 2*PI*(qx*U_L[a] + qy*V_L[a] + qz*W_L[a]);
                double cosValue, sinValue;
                sincos(exp,&sinValue,&cosValue);
                Amp = cuCadd(Amp,make_cuDoubleComplex(cosValue,sinValue));
        }
        Int_2D[o*pixel_n+p] = Int_2D[o*pixel_n+p] + cuCreal(cuCmul(Amp,
cuConj(Amp)));
}
```

And the execution time?

- Serial - 128 seconds
- parallel - 12 seconds for 100x the work

This is a BIG win and we haven't even tried to optimize the use of the GPU or use multiple GPUs!

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 1024.
  - 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.

And 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 (3+G)
  - 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

Wrap up - day 1

- 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 talks - 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