Looking Forward

Goal: Consider some of the parallel opportunities on the horizon

Grid Computing

- Define a cluster computer as an connected set of independent computers … it usually implies "connected by a dedicated network"
- Grid: the Internet is the only network; "resources not in a single administrative unit"
  - Greater latency -- locality, a bigger deal
  - Interference by other traffic
  - Nodes more autonomous -- configuration, security, heterogeneity, etc all issues
- Bottomline: Grid has all of the problems of || architectures, but worse and more

Grid As || Computation

- SETI and BOINC type problems work well because (a) the problems are independent, and (b) have limited need for communication
- Other computations are proportionally tougher -- best to consider Grid as distributed resource rather than a || computer

Attached Processors

- Several forms of attached processors are popular:
  - FPGA
  - Cell
  - GPGPU
- These machines do not match the CTA model -- why? -- and require different programming concepts
  - There is a machine model (type architecture) by Ebeling, van Essen and Ylvisaker

Attached Processor Key Properties

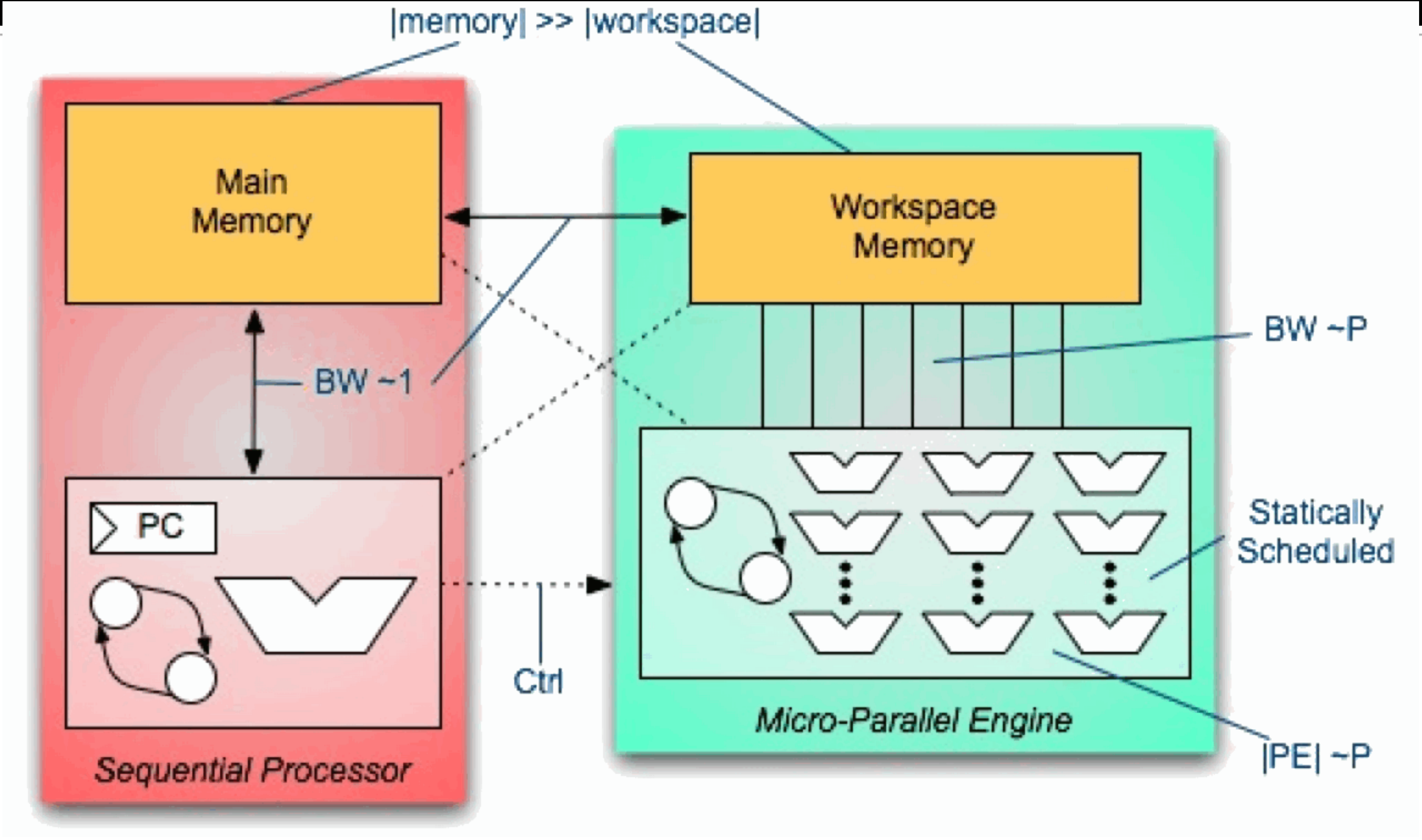
- These engines benefit from Si advances more than standard processors
- They exploit …
  - Kernel processing … only speeding inner loop; leave all other processing/orchestrating to CPU
  - Data streaming from memory to get high throughput … moving lots of data fast permits cheap computations on each item
  - Specialized circuit designs at the expense of generality, making VLSI pay big

Attached FPGA

- Field Programmable Gate Array (Xilinx, Altera)
  - "Programmable hardware"
  - Ideal for ints, bit-twiddling, etc.
  - Very dramatic "regime change" between CPU and attached engine
  - FPGAs are supported "well" for circuit designers, but tools low level compared to IDE
  - Drop into "Opteron slot" for fast system build

Mostly for special purpose

Type Architecture for Attached Procs



A Type Architecture for Hybrid Micro-Parallel Computers (search UW)

General Principle

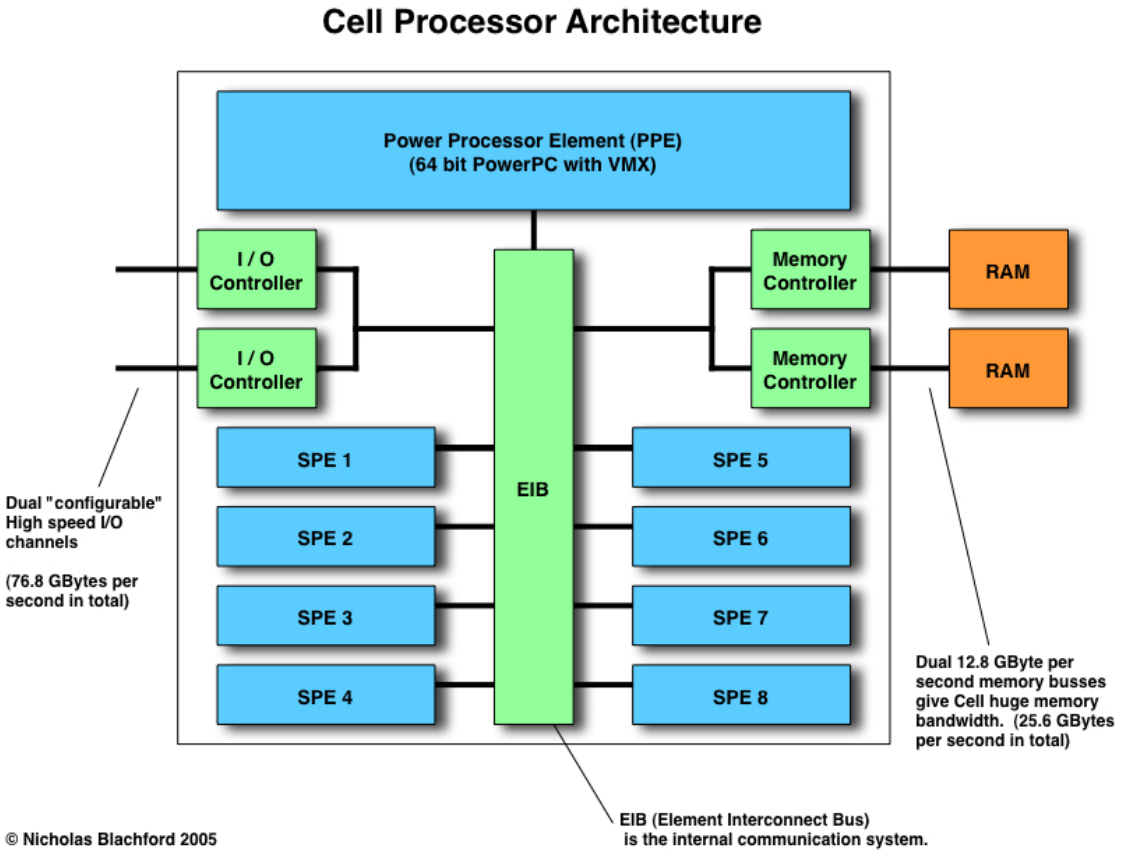
- When programming a family of machines that departs significantly from the "usual suspects," work out a type architecture
- The TA will be the simplest logical machine that exhibits the important costs
- Notice
  - How the CTA does this
  - How the EVY Machine Does This

Cell Processor

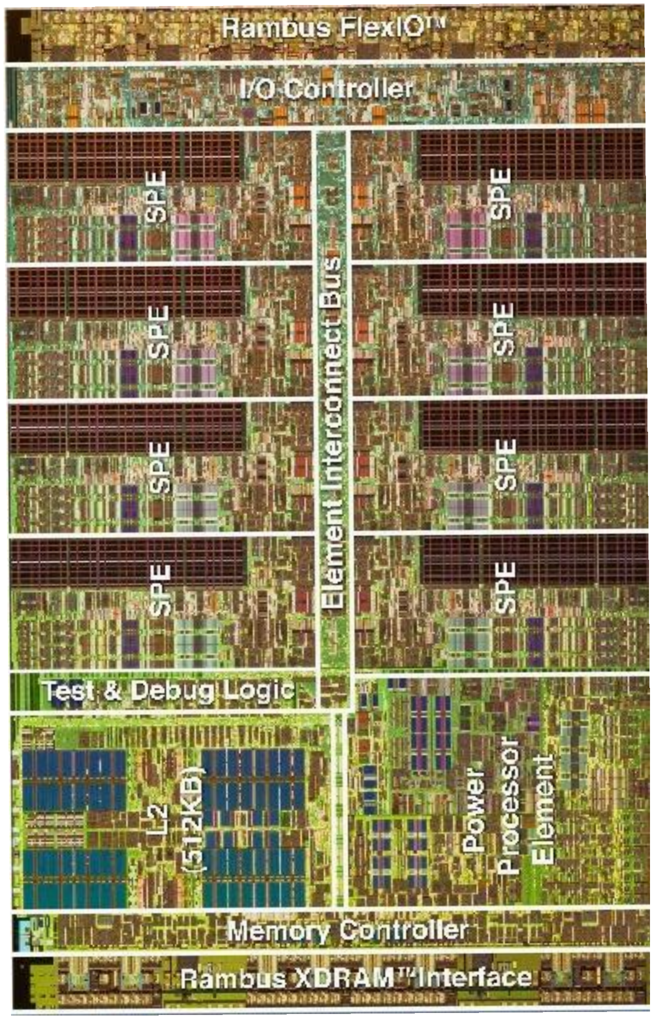
- Though IBM canceled Cell development last November, the ideas in it are crucial to our parallel computation discussion
- (And it's still a fire breather)

Consider Cell

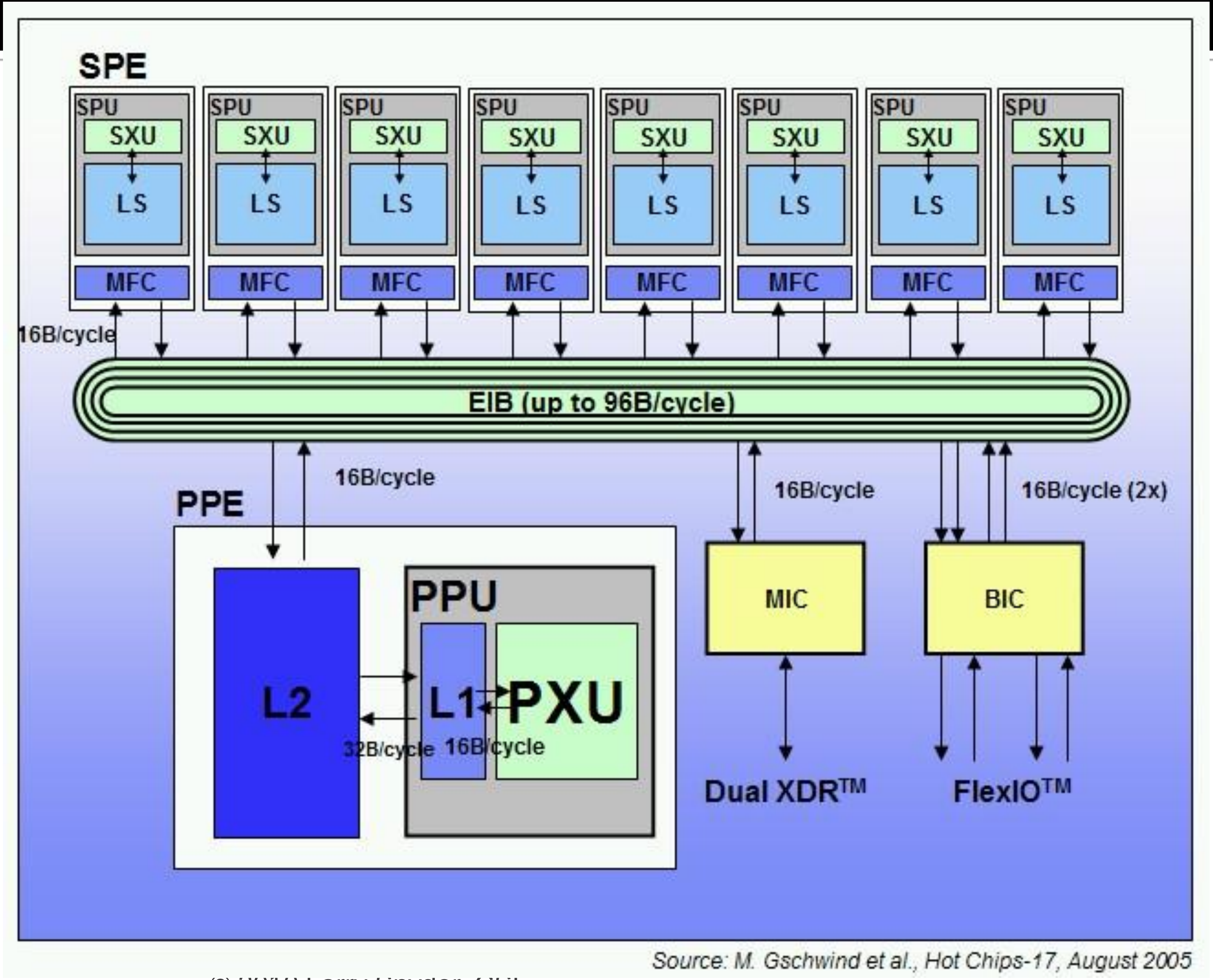
- Basic Floor plan and photo



Cell Processor Architecture
Power Processor Element (PPE)
(64 bit PowerPC with VMX)
I / O Controller
I / O Controller
Memory Controller
Memory Controller
RAM
RAM
EIB
SPE 1
SPE 2
SPE 3
SPE 4
SPE 5
SPE 6
SPE 7
SPE 8
Dual "configurable" High speed I/O channels
(76.8 GBytes per second in total)
Dual 12.8 GByte per second memory busses give Cell huge memory bandwidth. (25.6 GBytes per second in total)
EIB (Element Interconnect Bus) is the internal communication system.
© Nicholas Blachford 2005



Logical Level … A TA

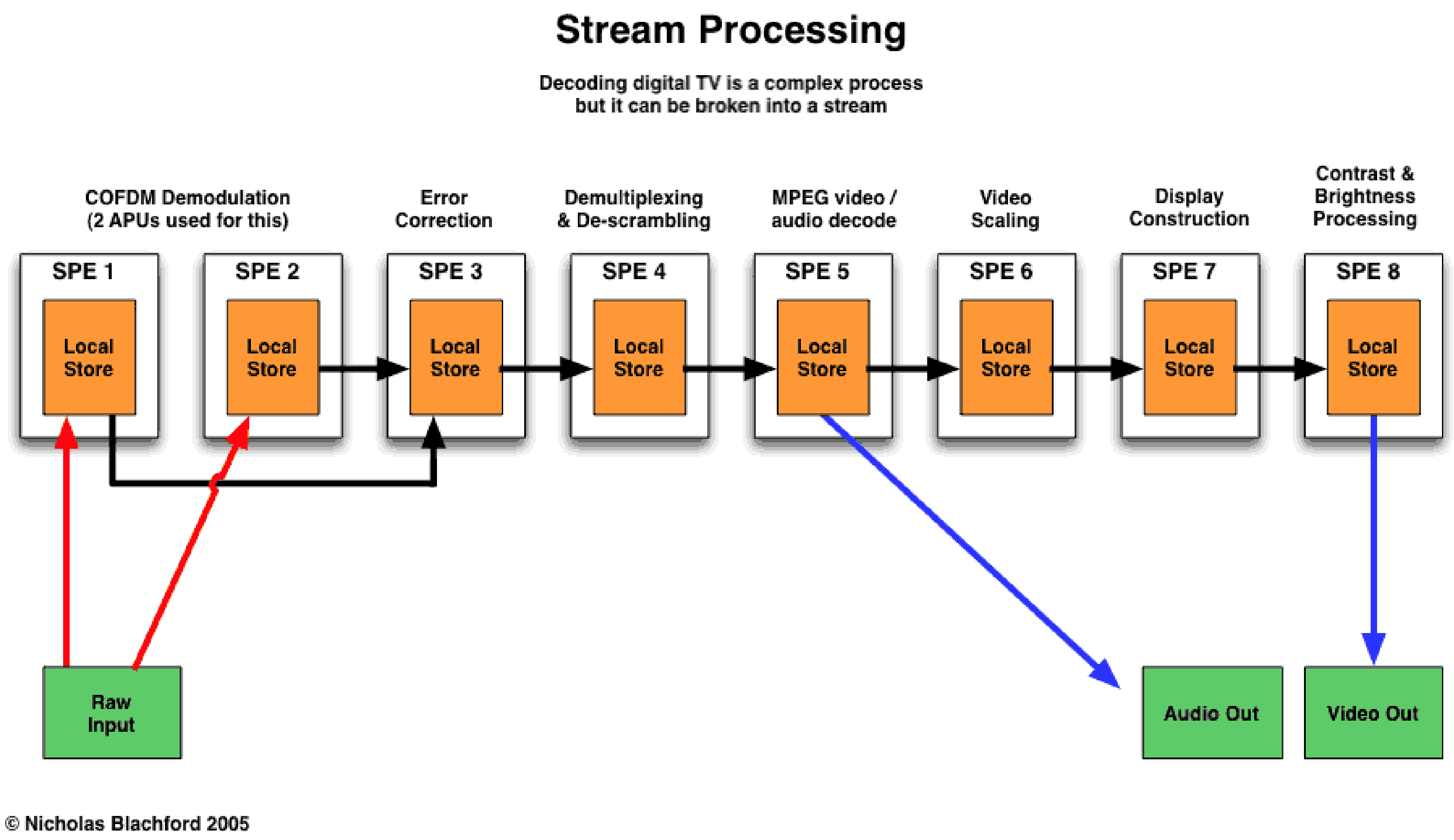


Source: M. Gschwind et al., Hot Chips-17, August 2005

Cell Programming

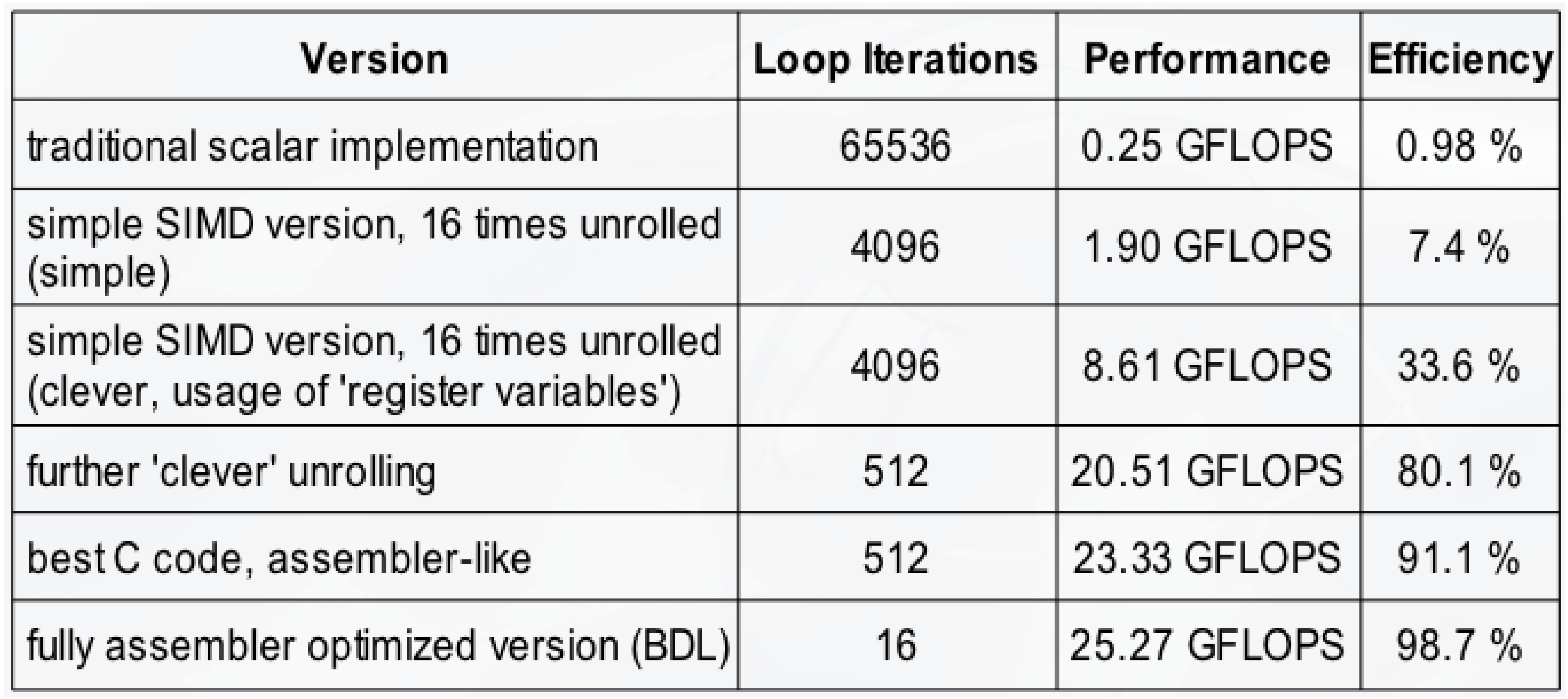
- Blachford describes cell programming
  - *Cell however, has gone against the grain and actually removed a level of abstraction. The programming model for the Cell will be concrete, when you program an SPE you will be programming what is in the SPE itself, not some abstraction. You will be "hitting the hardware" so to speak. The SPEs programming model will include 256K of local store and 128 registers, the SPE itself will include 128 registers and 256K of local store, no less, no more.* [Feb, 2005]

Cell As A Pipeline



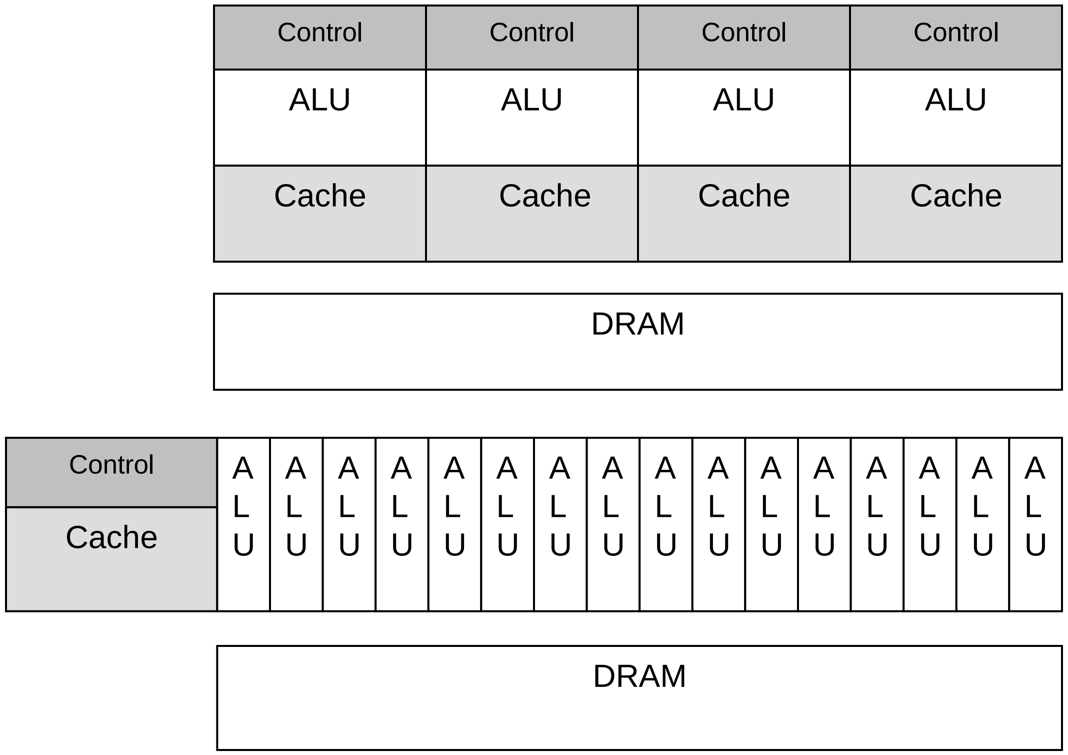
Programming Matrix Multiplication

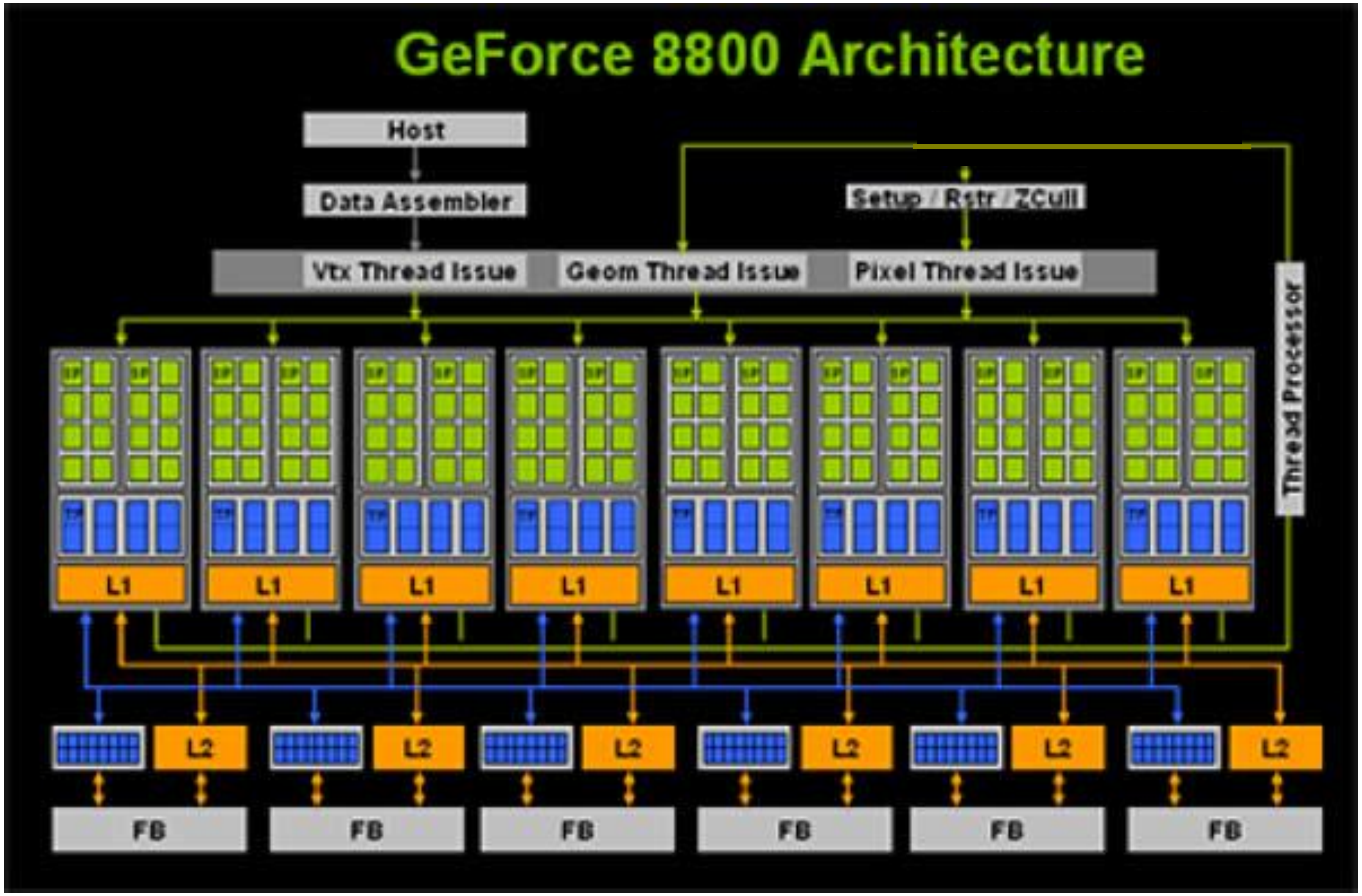
- Complexity seems dramatic

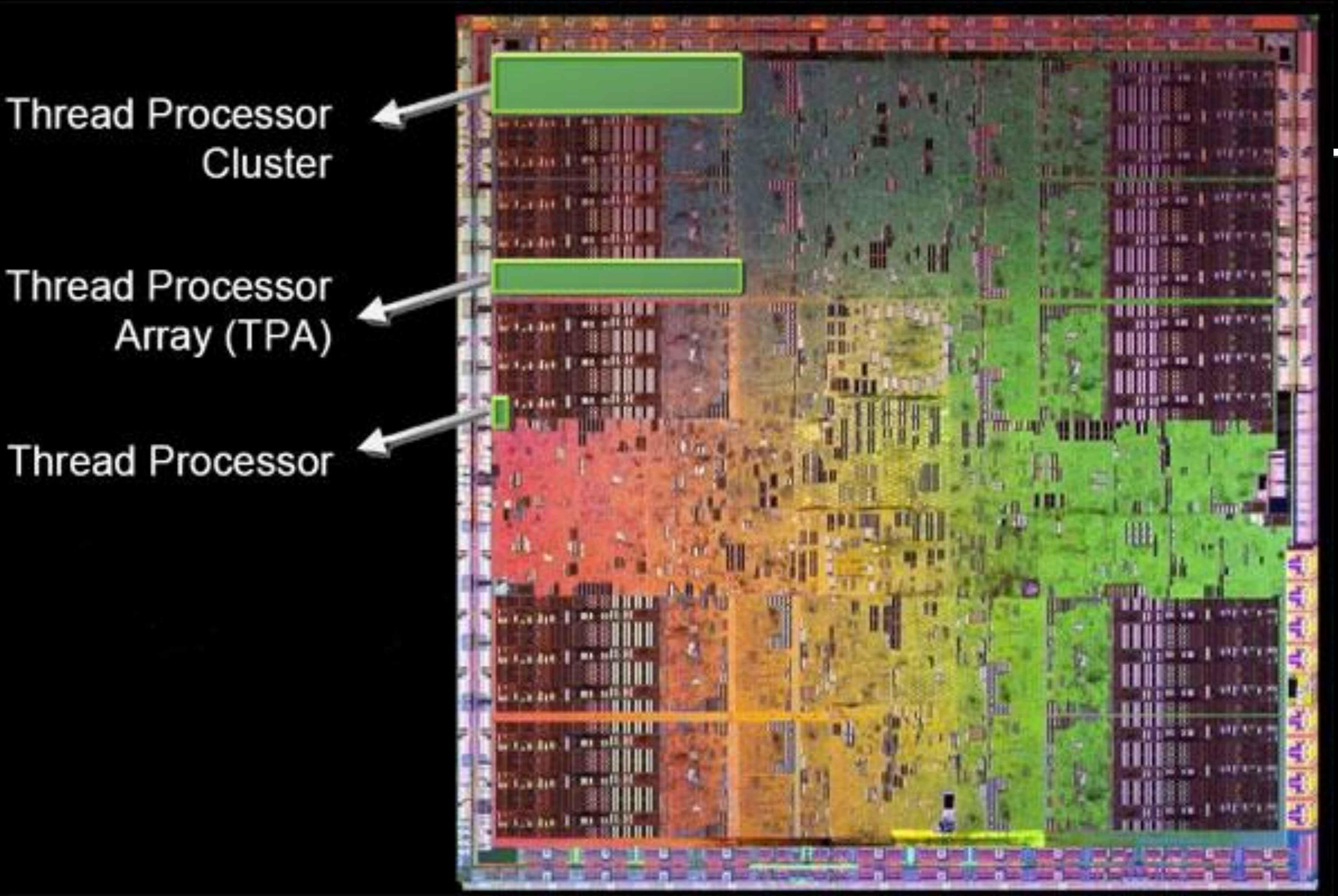


Version	Loop Iterations	Performance	Efficiency
traditional scalar implementation	65536	0.25 GFLOPS	0.98 %
simple SIMD version, 16 times unrolled (simple)	4096	1.90 GFLOPS	7.4 %
simple SIMD version, 16 times unrolled (clever, usage of 'register variables')	4096	8.61 GFLOPS	33.6 %
further 'clever' unrolling	512	20.51 GFLOPS	80.1 %
best C code, assembler-like	512	23.33 GFLOPS	91.1 %
fully assembler optimized version (BDL)	16	25.27 GFLOPS	98.7 %

GPU Compared To Multicore




GeForce 8800 Architecture
Host
Data Assembler
Setup / Rstr / ZCull
Vtx Thread Issue
Geom Thread Issue
Pixel Thread Issue
Thread Processor
SP
TF
L1
L2
FB

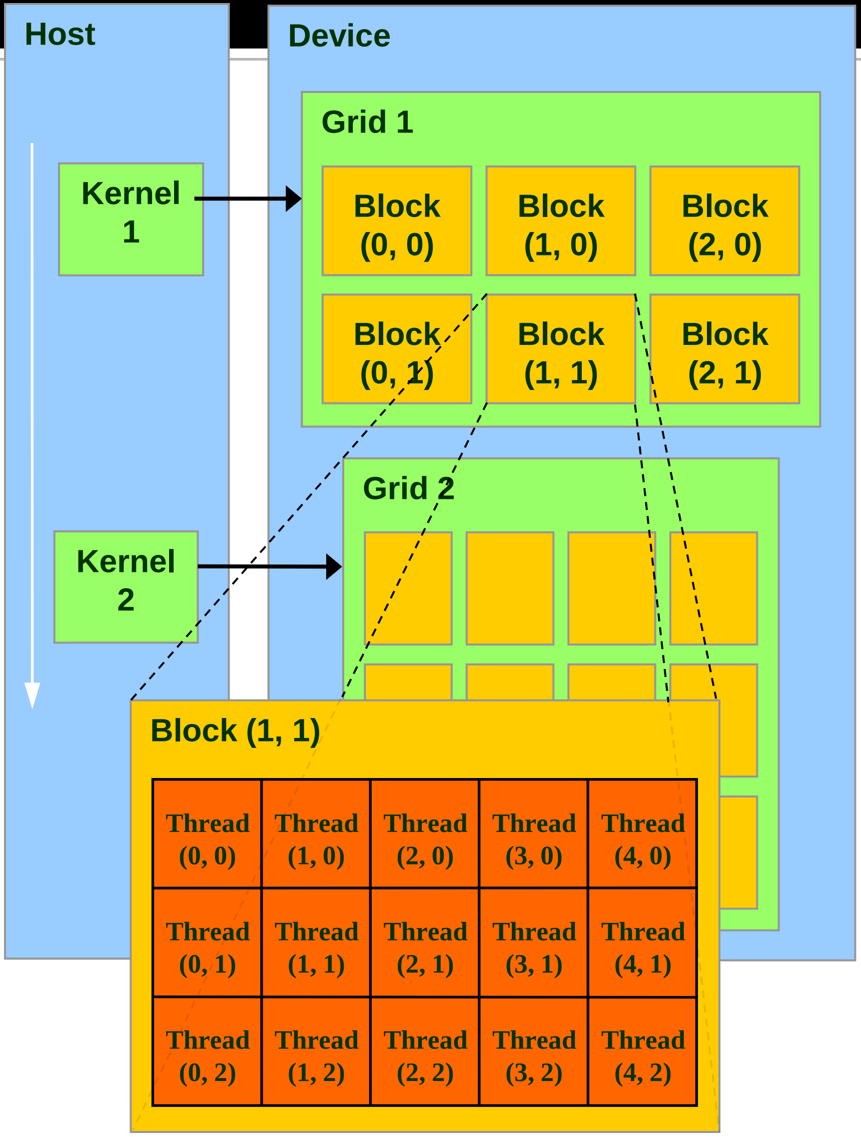

Thread Processor Cluster
Thread Processor Array (TPA)
Thread Processor

CUDA Programming Model

- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
- Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few

Thread Batching: Grids and Blocks

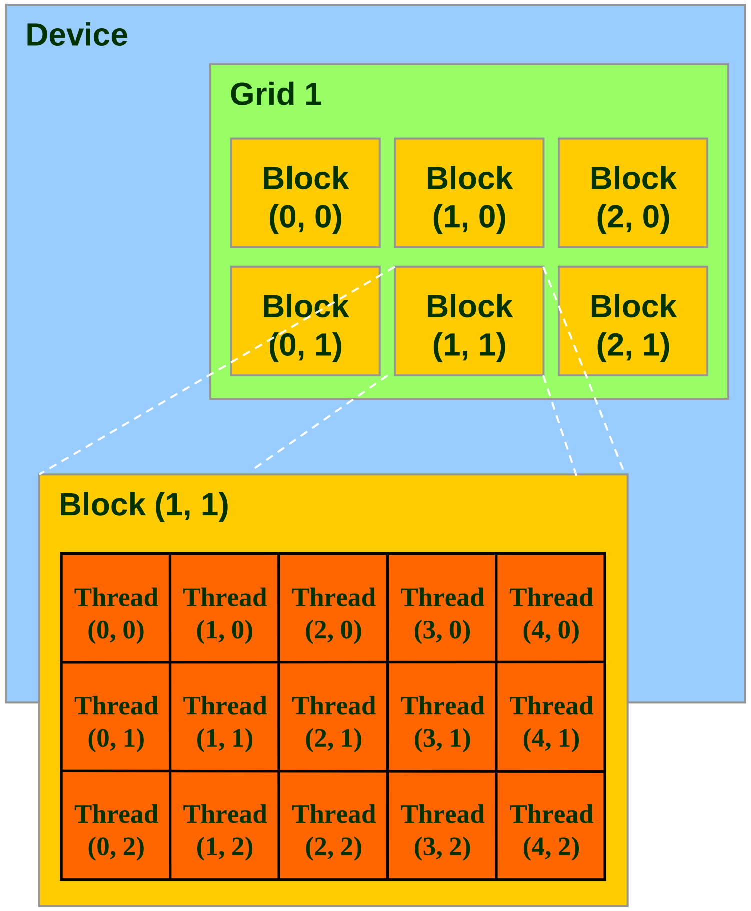
- A kernel is executed as a grid of thread blocks
  - All threads share data memory space
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate



Courtesy: NDVIA

Block and Thread IDs

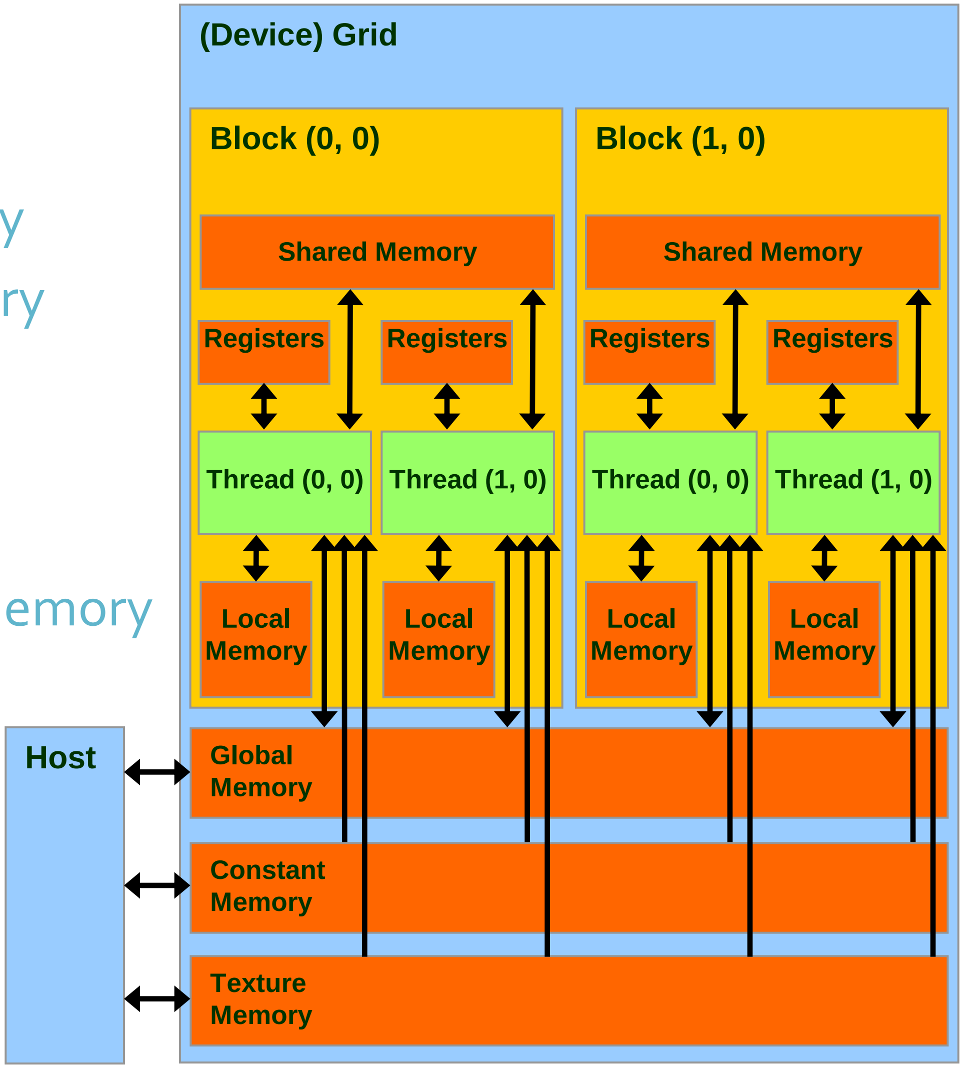
- Threads and blocks have IDs
  - So each thread can decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



Courtesy: NDVIA

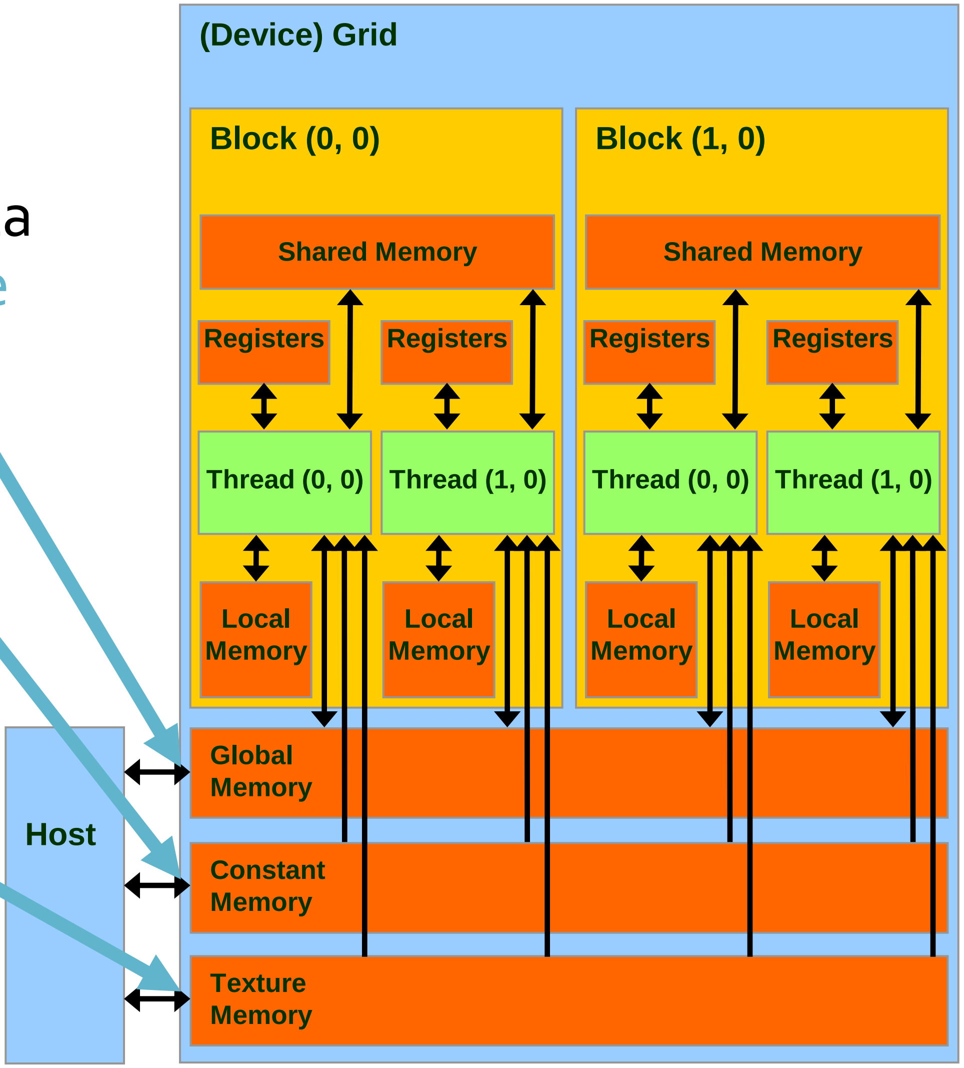
CUDA Device Memory Space

- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory
- The host can R/W global, constant, and texture memories



Global, Constant, and Texture Memories

- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
- Texture and Constant Memories
  - Constants initialized by host
  - Contents visible to all threads



Courtesy: NDVIA

CSE524 Bottom Line

- Attached processors provide enormous power for low price, but they …
  - Use a different programming paradigm than our CTA-based view
  - Have no pretense of being general purpose
  - Should be programmed as though configuring hardware or building a hybrid machine
- Strengths are fast data streaming and leveraging VLSI
- Liabilities are programming challenges and rigidity

New Abstractions: Look for More

- A goal of || abstractions is to specify efficient computation without specifying unnecessary order
- *Problem Space Promotion* is One Way
- Consider a counting sort:

```
for (j=0; j<n; j++) {
   B[j] = 0;          // Init
}
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    if (A[i]<A[j]) // Count #
       B[j]++;     // larger
  }
}
for (j=0; j<n; j++) {
  B[j] = A[B[j]];  // Reorder
}
for (j=0; j<n; j++) {
  A[j] = B[j];     // Restore
}
```

Alternative: PSP

- Specify the operations w/o ordering

C□$S^T < S$	compare all pairs,(1 if true, else 0)
P□sum_cols(C)	column sums give sorted index positions
S□S[P]	permute elements of S into order

- The solution "promotes" the operand to be 2D, allowing all pairs to be tested at once
  - This technique works a lot … 3D MM is instance
  - Promotion is logical … no actual copy needed

Bradford L. Chamberlain, E Christopher Lewis, and Lawrence Snyder. **Problem space promotion and its evaluation as a technique for efficient parallel computation.** In *Proceedings of the ACM International Conference on Supercomputing*, 1999.

Transactional Memory-A Hot Idea

- You all know parallel programming is tough
- How to make it easier? Raise abstractions!
- Transactional Memory: Return of old idea
  - Databases concurrently manage external data consistently using multiple computers; well studied
  - Apply idea to concurrent management of the internal memory image
  - Transaction: Atomically change memory to new state or do nothing at all
  - Say the goal not how to achieve it

Idea: David Lomet in 1977

Atomic

An *easier-to-use* and *harder-to-implement* primitive

```
void deposit(int x){
synchronized(this){
  int tmp = balance;
  tmp += x;
  balance = tmp;
}}
```

```
void deposit(int x){
atomic {
  int tmp = balance;
  tmp += x;
  balance = tmp;
}}
```

lock acquire/release

(behave as if)
no interleaved computation;
no unfair starvation

Transactions

- The code executing atomically is everything (dynamically) between braces, including **`foo().`**

```
atomic {
  if (x != null) x.foo();
      y = true;
}
```

- Three choices: commit; abort; not terminate
- Optimistic: Little overhead if no conflict
- Avoid races and deadlocks due to lock acquisition

Transactions Advantages

- In DB transactions have ACID properties
  - A = atomicity … sequence of operations never interrupted or incomplete; commit or abort
  - C= consistency …changes leave memory in consistent state relative to application; for example new_balance==old_balance+deposit
  - I = isolation … transaction works correctly with any combination of other transactions
  - D = durability … result persists; not appropriate for multithreading memory case

DB & TM Transactions Are Different

- DBs use disks, meaning the SW support for DB transactions not time-critical; referencing memory is too brief (and frequent) to allow for heavy-weight protection
- TM need not be durable (last) since data doesn't outlast execution; simplifying
- TM must retrofit into a rich world of legacy code … must coexist with all other mechanisms; pervasive changes not feasible

Atomic Doesn't Solve Everything

- It's not difficult to mess up using `atomic`

```
Thread 1                    Thread 2
atomic {                    atomic {
while (!flagA)                flagA = true;
  flagB = true; while (!flagB);
}                       }
```

  flags have to be true at the start of block
- This code not serializable, i.e. there is no correct serial execution

TM Promising, But NRFPT

- Transactions are no panacea
  - Neither hardware nor software implementations have proved themselves
- Very Nice Monograph: Larus & Rajwar

http://www.morganclaypool.com/doi/abs/10.2200/S00070ED1V01Y200611CAC002

- A fundamental problem TM will not solve: How to scale shared memory computations to architectures with much larger λ, which are inevitable

Break

An Alternate Concurrency Primitive

- Rather than using the Test&Set to guard shared data, use Fetch&Add
  - Fetch&Add is an atomic read-modify-write operation on memory -- requires special hardware, to be discussed
  - Use Fetch&Add as a semaphore and as a scheduler
- Operation: Fetch&Add(V,e)
  - V is a memory location
  - e is an integer expression
  - Contents of V are returned
  - New value of V is V+e
  - Operation is atomic

V: 0

Fetch&Add(V,1)

V: 1

0 is returned

Concurrent Fetch&Adds

- When multiple Fetch&Adds are executed simultaneously, they are serializable
- Assume Fetch&Add(V, e1), Fetch&Add(V, e2) execute simultaneously
    - Assuming an initial value of e0
    - Final value is e0+e1+e2
    - The 1st process receives either e0 or e0+e2, implying it was first (e0) or second (e0+e2)
    - The 2nd process receives either e0 or e0+e1, implying it was second (e0+e1) or first (e0)

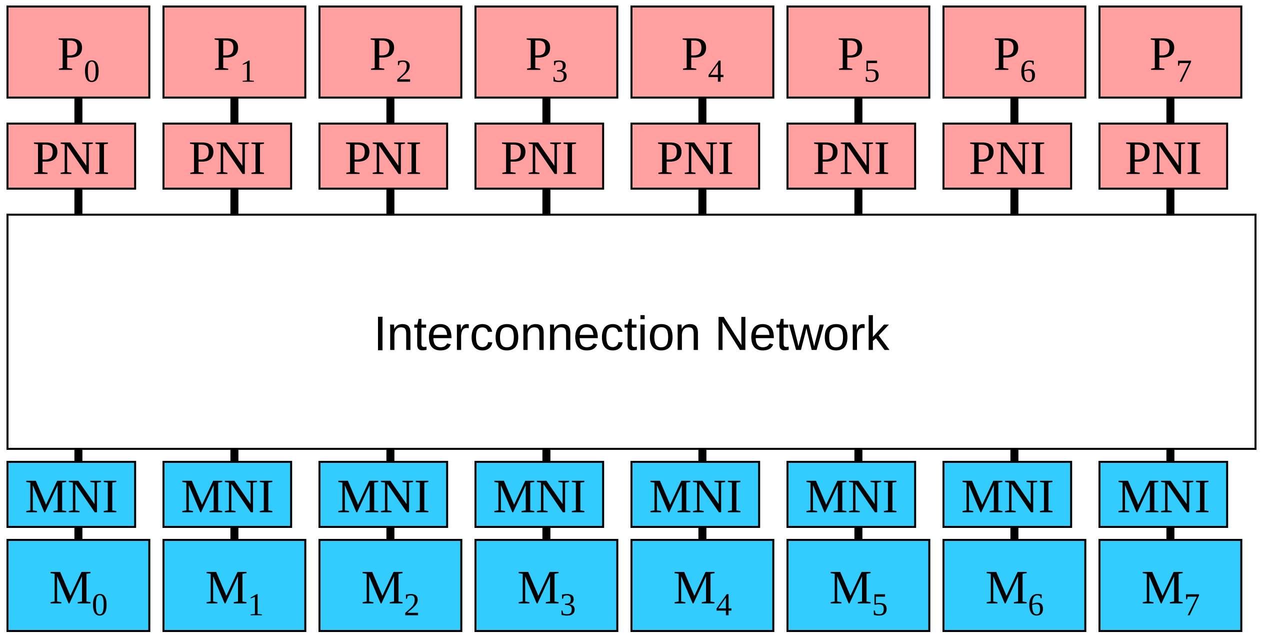
Suppose both execute Fetch&Add(I,1), then one gets I back, the other I+1, and final is I+2

Fetch&Add Exploits Sharing

- Though earlier solutions attempt to reduce sharing to reduce the amount of invalidation and acknowledgment, Fetch&Add does better with greater sharing
- Sharing is used to schedule or allocate, which is then independent activity
  - Sharing is concentrated in a few variables
  - Fine grain size is possible
- Since load/store, Test&Set, etc. are implementable, it is a “sufficient” primitive

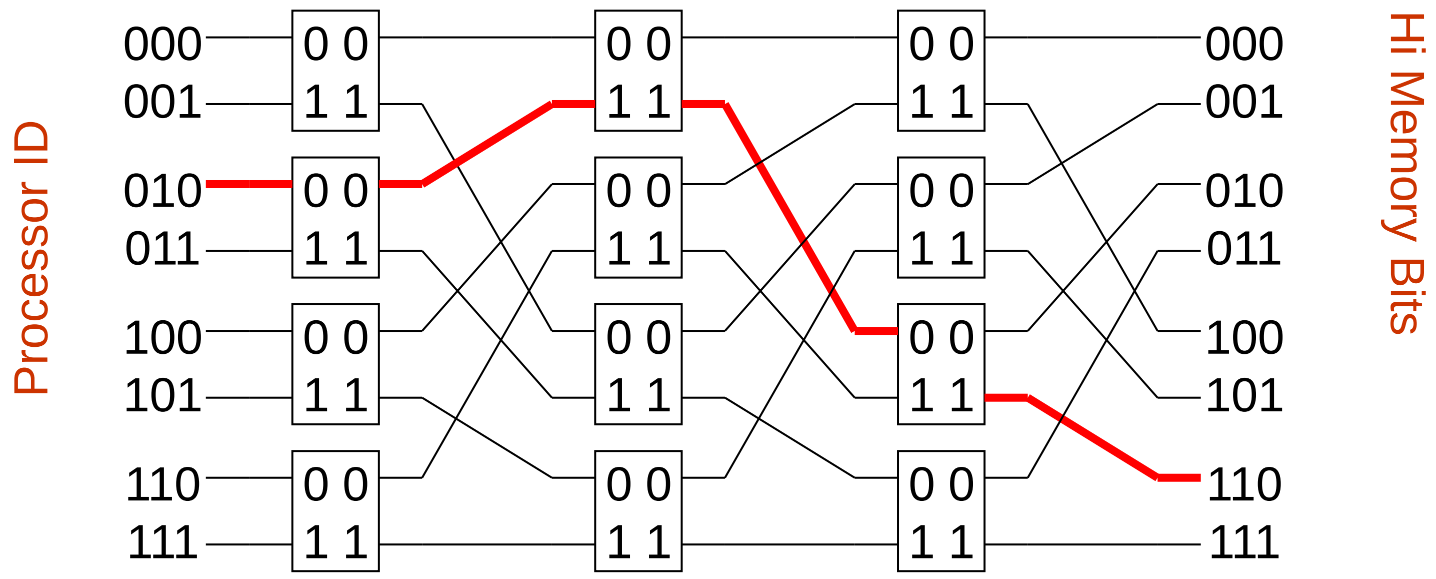
Implementing Fetch&Add

- Fetch&Add assumes a flat shared memory as implemented by a "dance hall architecture"



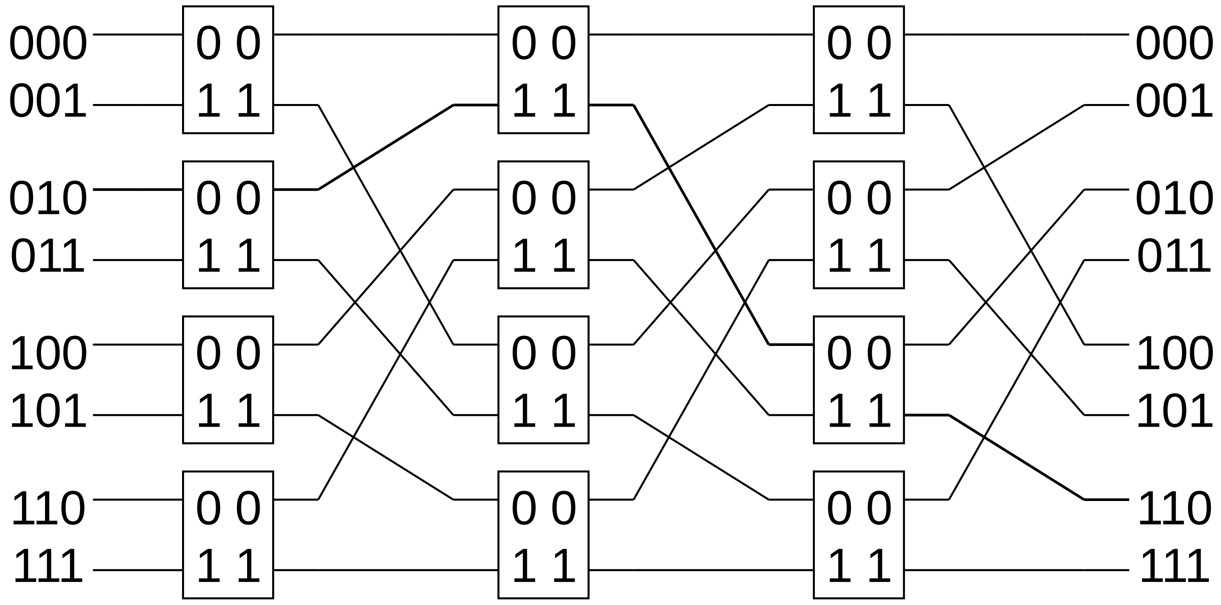
Omega Network

- The interconnection network is an Ω-network
  - Connection between 2 and 6 … follow bits to destination lsb to msb



Notice Details

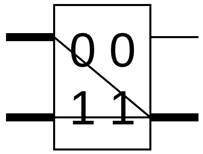
- The Ω-Network requires O(P log P) routers
- The given network uses 2x2 but 2^bx2^b work
- Wiring is consistent at each stage



Long Wires Are Necessary

Routing In Ω-Network

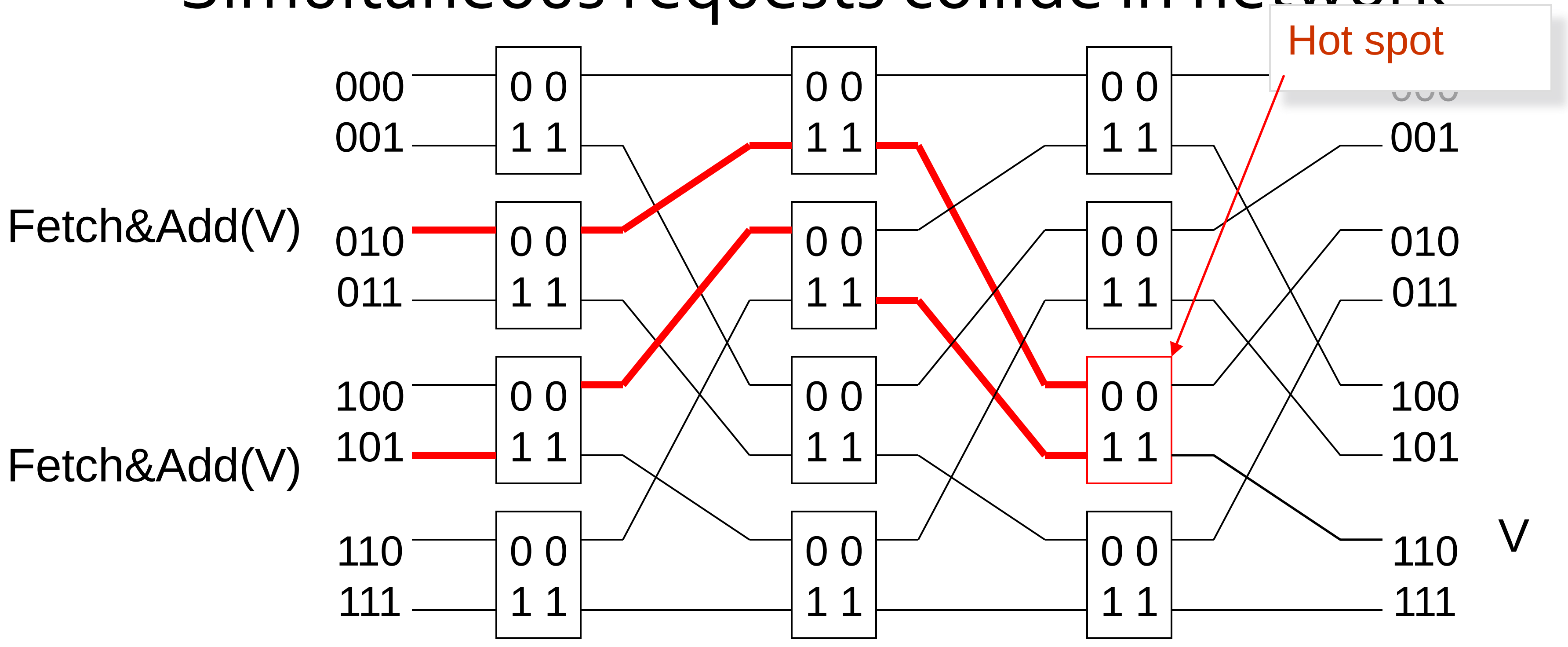
- The network is pipelined
- There is a unique path between any processor and memory port pair
- Conflicts are possible because there exist permutations in which packets collide
- What happens when two packets collide at a router?
  - Packet is delayed, leaving its "file"
  - Pipelining is affected, here comes more



The separate packets must be serialized

Two Processors Do Fetch&Add(V,1)

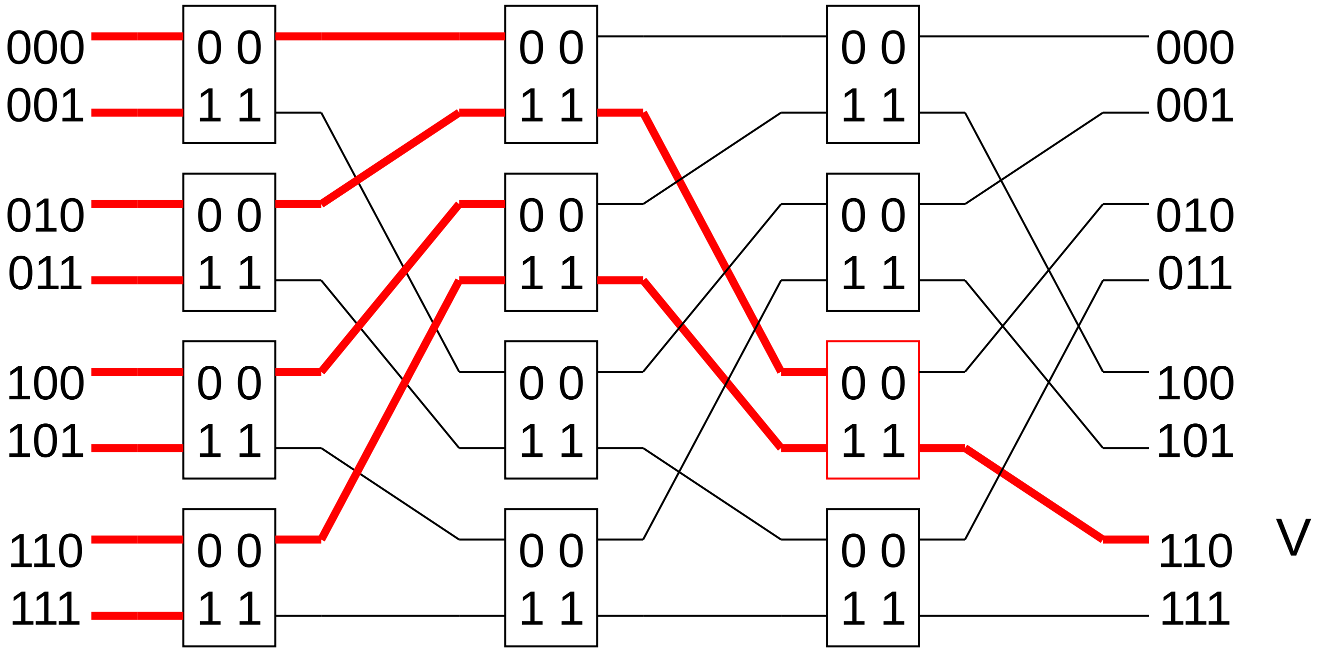
- Simultaneous requests collide in network



Fetch&Add increases potential for collisions

The Bright Idea: Combine Requests

Idea: Combine requests for same dest. In limit all nodes could reference same loc.

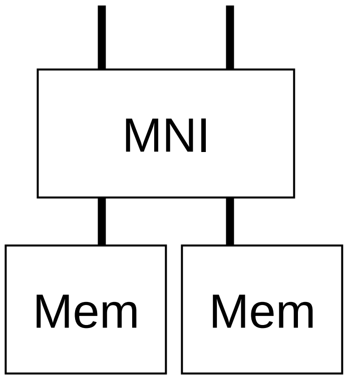


Loads & Stores

- At a switch combine loads and stores to a common location as follows
  - Load/Load -- forward one of the loads towards the memory, and when the value is returned, satisfy both
  - Load/Store -- forward the store, and when the ACK arrives back at the switch, return value to satisfy load
  - Store/Store -- forward one of the stores, and when the ACK arrives back at the switch, return it for both
- Processors are restricted to having only one outstanding request at a time to a given location

Implementing Fetch&Add

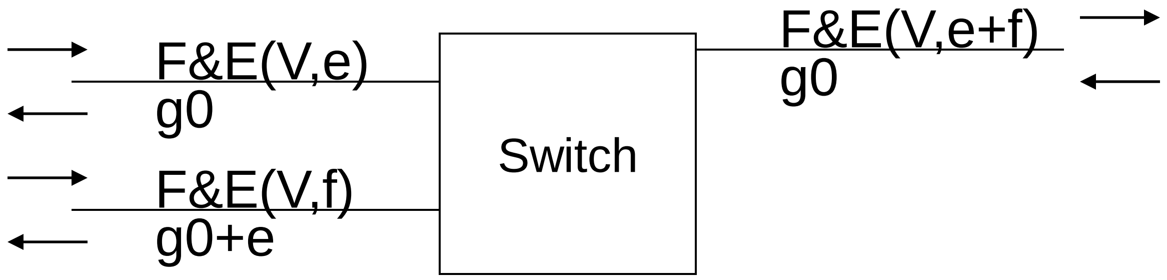
- Include an adder with the Memory Network Interface chips
- For Fetch&Add(V,e)
  - Fetch the value of V, say eo
  - Return eo to processor requesting
  - Add eo+e
  - Store eo+e back into V
- It is probably necessary to do these concurrently



Combining Fetch&Adds at Switch

Suppose Fetch&Add(V,e), Fetch&Add(V,f) arrive at a switch together …

- Form the sum f+e
- Send Fetch&Add(V,f+e) on to the memory
- Store e locally
- When go is returned by the memory
  - Return go as response to Fetch&Add(V,e)
  - Return go+e as response to Fetch&Add(V,f)



Combine F&A w/ other requests

- Combining can apply to all memory traffic to a location V
- Consider the following cases
  - Fetch&Add/Fetch&Add -- as just described
  - Fetch&Add/Load -- Treat Load as Fetch&Add(V,0)
  - Fetch&Add/Store -- If Fetch&Add(V,e) meets Store(V,f) send Store(V,e+f) to memory; when ACK is received, return f as value of F&A
- Conclusion -- it is possible to combine all requests to the same memory location

Just Thinking About, Will It Work?

- Potential Problems …
  - Network routing is driven entirely by performance, so a complicated switch is usually a problem
  - Routers typically forward non-blocked packets in <= 3 tix
  - Matching to recognize that two requests collide is an "add" operation
  - Combining is an "add" operation *after* the previous add
  - Combining relies on the requests getting to the switch simultaneously, or at worst, before the forwarded packet leaves … this is improbable
  - Most traffic is non-combinable -- headed for different places

  - A combining router was created by Susan Dickey

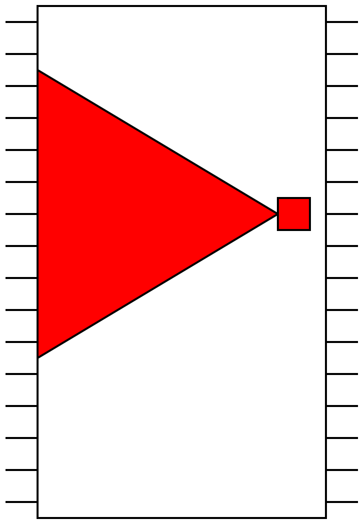
A Backup Strategy

- If the network switch is too slow then …
  - Do not combine at every stage … so that some stages can be fast
  - Use two networks, one fast and one that does combining -- it can handle the sharing requests
  - Combine only like requests, e.g. loads/loads
  - Limit combining at a node to two requests

  - As it happened
    - Only like requests have ever been implemented in switch
    - IBM used the two network solution in the RP3

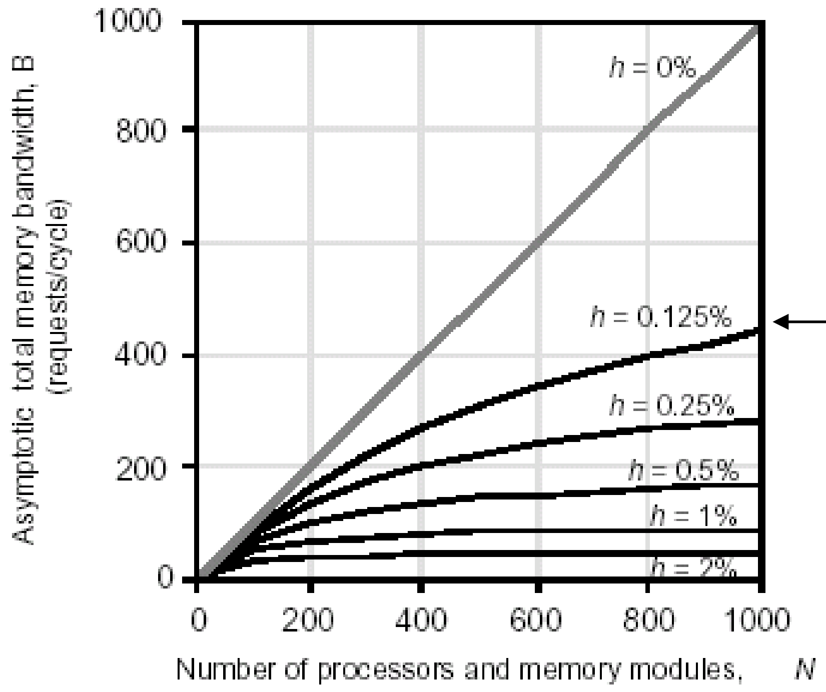
More Globally

- Norton and Pfister discovered in simulation for the RP3 computer that the Ω-Network develops hot-spots
- It was thought that combining would remove the hot-spots … it seemed to for 64-way network
- The problem is that once a node becomes hot, a "back-up" tree forms "behind" the node



More Globally

- Lee, Kruskal, Kuck studied by simulation, analysis
  - LKK discovered and named the "back-up tree"
  - Showed in simulation that the 64-way network is lucky
  - Combining doesn't help because it's the other traffic



0.125% traffic directed at a hot spot

Assessment

- It was a good idea but it didn’t work
  - Good
    - Fetch&Add is clever -- a primitive with good properties
    - Shifting from protecting data to allocating work is better
    - Computation at memory is powerful, worth doing
  - Bad
    - Pipelined multistage networks probably just don’t work
    - Complexity in a switch is wrong -- speed is essential
    - Failed to exploit locality -- caching basically impossible

Wrap Up

Goal: Summarize what we’ve covered and what you might have learned

High-Order Bit

- To write successful parallel programs, we need a model of how a || computer works
  - CTA is a generalization of || hardware
  - CTA favors ...
    - Locality because of high λ
    - Minimizing inter-thread dependences because of high λ plus wait time and sometimes contention
  - CTA algorithms were successful and usually they were the only ones that worked well

Lambda is always relevant, usually very significant

The Take Home Message

In parallel algorithm design and parallel programming, be guided by the CTA

Next Most Significant Bits

- We acquired parallel programming skills using small exercises
  - Thread programming
  - Peril-L
  - MPI
  - Chapel
- Experiences --
  - Much of it is low-level and grotty
  - Not always easy to get performance

Other Things To Attend To …

- Approach problems "top down", maximizing the highest level of parallelism first
  - Corollary: Don't try to be too smart at low level
- Granularity, the coarser the better
- Communication load, less is better
- Bandwidth, use wisely
- Minimize "programming to the machine"

Wrap-Up on Parallel Computing

- What in your view was the high-order bit?

My List …

- Using parallel computers is tough
- Parallel computers generally behave like the CTA, so program to it … it won’t disappoint
- Parallel algorithms often require fresh thinking -- sequential case may not be a good place to begin
- CMPs are sweet-spot now, but for how long?
- Reduce/Scan are basic building blocks

My List (continued) …

- Programming tools are all over the place
  - OpenMP is **very** simple, but not too expressive
  - Pthreads, a standard library, but very low level
  - MPI is universal, low level and abstraction-free
  - ZPL leaves all parallelism to compiler, but gives WYSIWYG as guide to writing good programs
  - Chapel, Transactional Memory have promise, but NRFPT
- Hardware design is very volatile at moment

Course Evaluations

- Need a volunteer at each site
- Fill out both forms
  - Bubble in the white form
  - Give me suggestions on the yellow form

  - Volunteer
    - On campus, drop the completed forms in box across st
    - In Redmond, Fedex the completed forms to campus