CIS 631
Parallel Processing

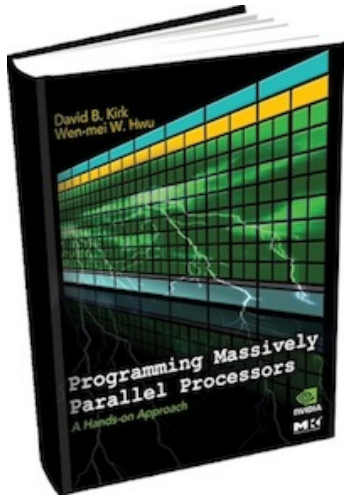
Lecture 13: GPUs and Heterogeneous Computing

Allen D. Malony
malony@cs.uoregon.edu

Department of Computer and Information Science
University of Oregon

Acknowledgements

- Portions of the lectures slides were adopted from:
  - Programming Massively Parallel Processors: A Hands-on Approach
    - David B. Kirk, NVIDIA, and Wen-mei Hwu, University of Illinois, Urbana-Champaign
    - Morgan Kaufmann, 2010

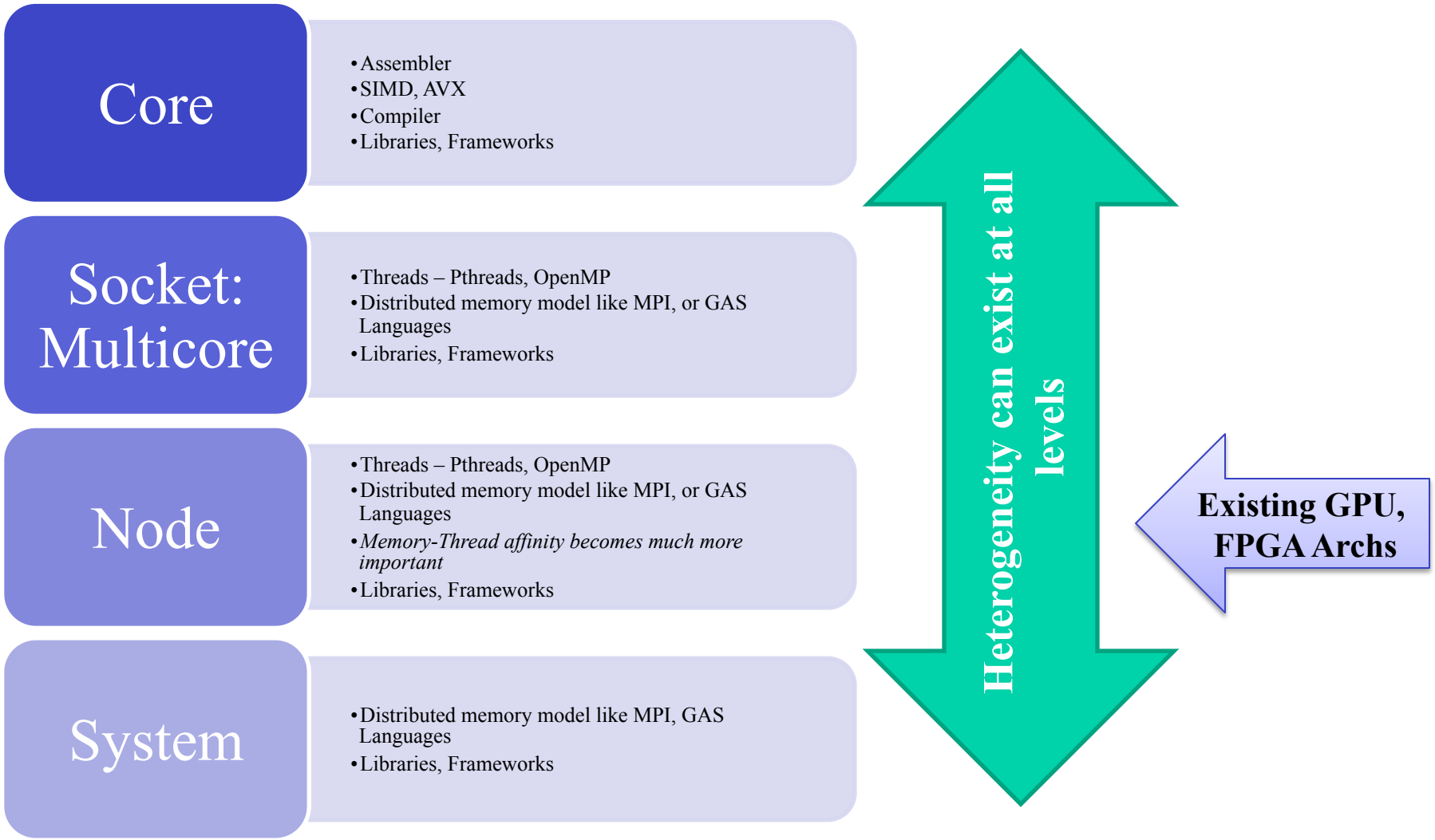


  - Rick Vuduc, CSE 6230, Fall 2011, Georgia Institute of Technology
  - Bryan Cantanzaro, Introduction to CUDA/OpenCL and Manycore Graphics Processors, NVIDIA
  - Jeff Vetter, Heterogeneous Computing with GPUs
  - NVIDIA

Thinking about Parallelism

Core	• Assembler • SIMD, AVX • Compiler • Libraries, Frameworks
Socket: Multicore	• Threads – Pthreads, OpenMP • Distributed memory model like MPI, or GAS Languages • Libraries, Frameworks
Node	• Threads – Pthreads, OpenMP • Distributed memory model like MPI, or GAS Languages • *Memory-Thread affinity becomes much more important* • Libraries, Frameworks
System	• Distributed memory model like MPI, GAS Languages • Libraries, Frameworks

Thinking about Parallelism (2)



#2: Tianhe-1A uses 7,000 NVIDIA GPUs (2011)

- Tianhe-1A uses
  - 7,168 NVIDIA Tesla M2050 GPUs
  - 14,336 Intel Westmeres
- Performance
  - 4.7 PF peak
  - 2.5 PF sustained on HPL
- 4.04 MW
  - If Tesla GPU's were not used in the system, the whole machine could have needed 12 megawatts of energy to run with the same performance, which is equivalent to 5000 homes
- Custom fat-tree interconnect
  - 2x bandwidth of Infiniband QDR

EDITION: INTERNATIONAL | U.S. | MÉXICO | ARABIC
Set edition preference

CNN

Home | Video | World | U.S. | Africa | Asia | Europe | Latin America | Middle East | Business

World's fastest supercomputer belongs to China

Mashable By **Stan Schroeder,** Mashable
October 28, 2010 -- Updated 1406 GMT (2206 HKT) | Filed under: Innovation

The supercomputer was unveiled yesterday at the Annual Meeting of National High Performance Computing.

STORY HIGHLIGHTS

- Tianhe-1A unveiled Wednesday at HPC China 2010 in Beijing
- Supercomputer has a performance record of 2.507 petaflops
- Tianhe-1A designed by the National University of Defense Technology
- System cost $88 million and its 103 cabinets weigh 155 tons

RELATED TOPICS

China
Computer Technology

(Mashable) -- The United States no longer owns the world's fastest supercomputer.

A computer called Tianhe-1A, unveiled on Wednesday at a conference in Beijing, China, can run calculations faster than the previous speed leader, a computer at a U.S. lab in Tennessee.

The new computer set a performance record by crunching 2.507 petaflops of data at once. The previous leader, a computer called Cray XT5 Jaguar and located at the Oak Ridge National Laboratory, completed 1.75 petaflop calculations.

Analysts say the new record underscores China's place as a global tech leader.

ORNL "Titan" System

- Upgrade of existing Jaguar Cray XT5
- Cray Linux Environment operating system
- Gemini interconnect
  - 3-D Torus
  - Globally addressable memory
  - Advanced synchronization features
- AMD Opteron 6200 processor (Interlagos)
- New accelerated node design using NVIDIA multi-core accelerators
  - 2011: 960 NVIDIA M2090 "Fermi" GPUs
  - 2012: 10-20 PF NVIDIA "Kepler" GPUs
- 10-20 PFlops peak performance
  - Performance based on available funds
- 600 TB DDR3 memory (2x that of Jaguar)

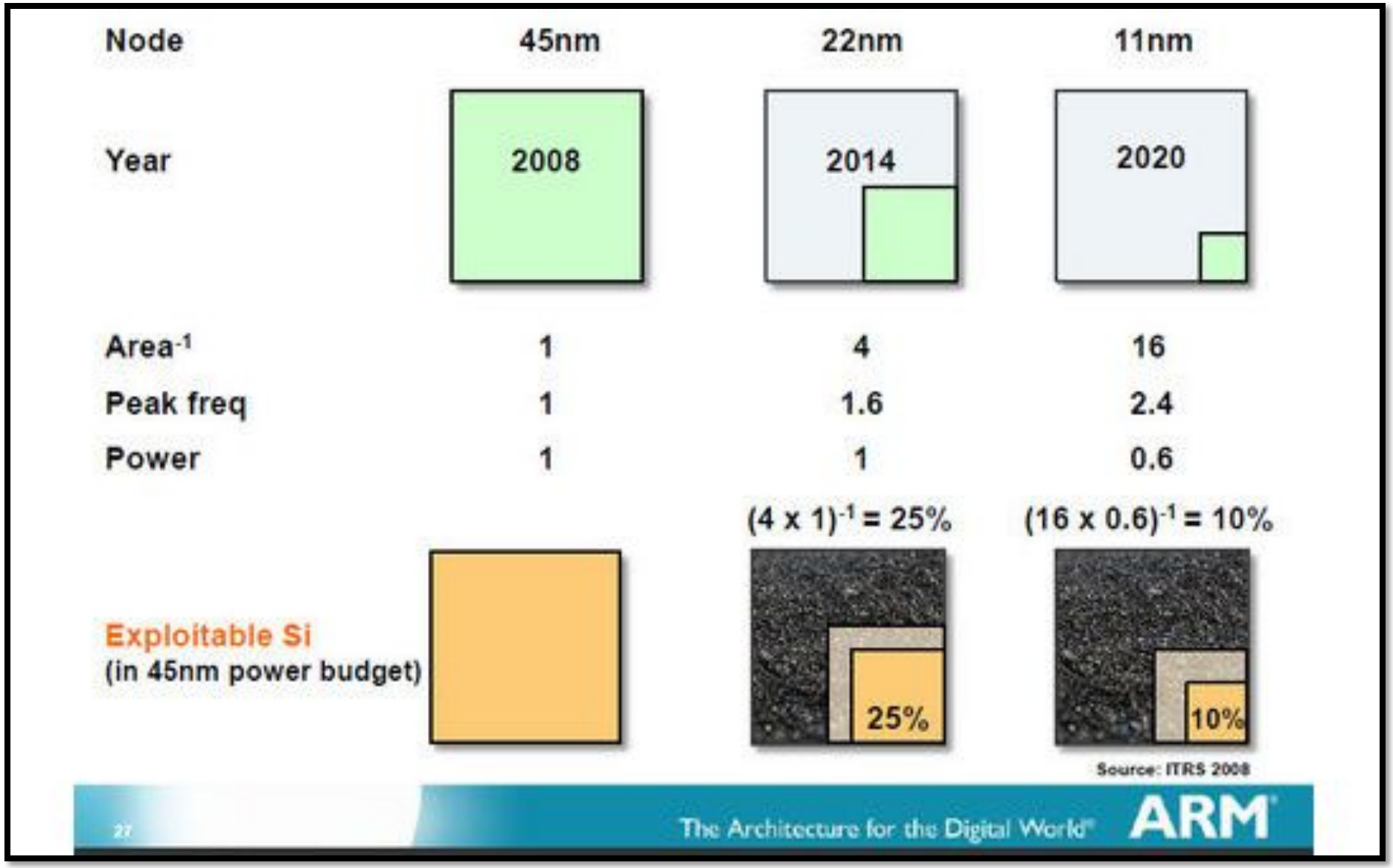
Titan Specs	
Compute Nodes	18,688
Login & I/O Nodes	512
Memory per node	32 GB + 6 GB
NVIDIA "Fermi" (2011)	665 GFlops
# of Fermi chips	960
NVIDIA "Kepler" (2012)	>1 TFlops
Opteron	2.2 GHz
Opteron performance	141 GFlops
Total Opteron Flops	2.6 PFlops
Disk Bandwidth	~ 1 TB/s

Contemporary Architectures

Date	System	Location	Comp	Comm	Peak (PF)	Power (MW)
2009	Jaguar; Cray XT5	ORNL	AMD 6c	Seastar2	2.3	7.0
2010	Tianhe-1A	NSC Tianjin	Intel + NVIDIA	Proprietary	4.7	4.0
2010	Nebulae	NSCS Shenzhen	Intel + NVIDIA	IB	2.9	2.6
2010	Tsubame 2	TiTech	Intel + NVIDIA	IB	2.4	1.4
2011	K Computer	RIKEN/Kobe	SPARC64 VIIIfx	Tofu	10.5	12.7
2012	Titan; Cray XK6	ORNL	AMD + NVIDIA	Gemini	10-20	7?
2012	Mira; BlueGeneQ	ANL	SoC	Proprietary	10	?
2012	Sequoia; BlueGeneQ	LLNL	SoC	Proprietary	20	?
2012	Blue Waters; Cray	NCSA/UIUC	AMD + (partial) NVIDIA	Gemini	?	?
2013	Stampede	TACC	Intel + MIC	IB	?	10

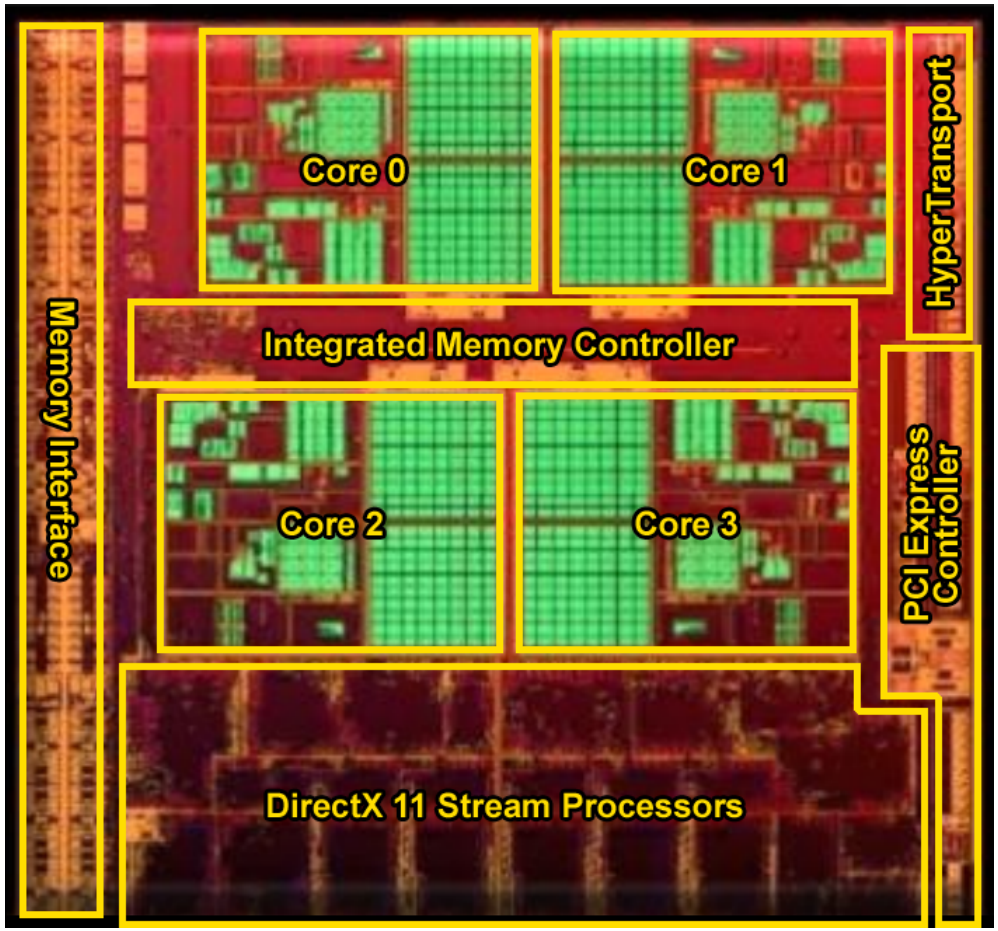
Trend #1: Facilities and Power

Trend #2: Dark Silicon (Heterogeneity, Specialization)



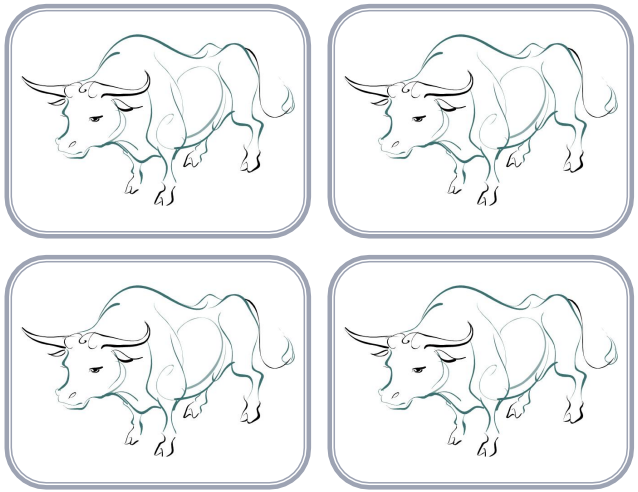
AMD's Llano: A-Series APU

- Combines
  - 4 x86 cores
  - Array of Radeon cores
  - Multimedia accelerators
  - Dual channel DDR3
- 32nm
- Up to 29 GB/s memory bandwidth
- Up to 500 Gflops SP
- 45W TDP



Source: AMD

Multicore and Manycore

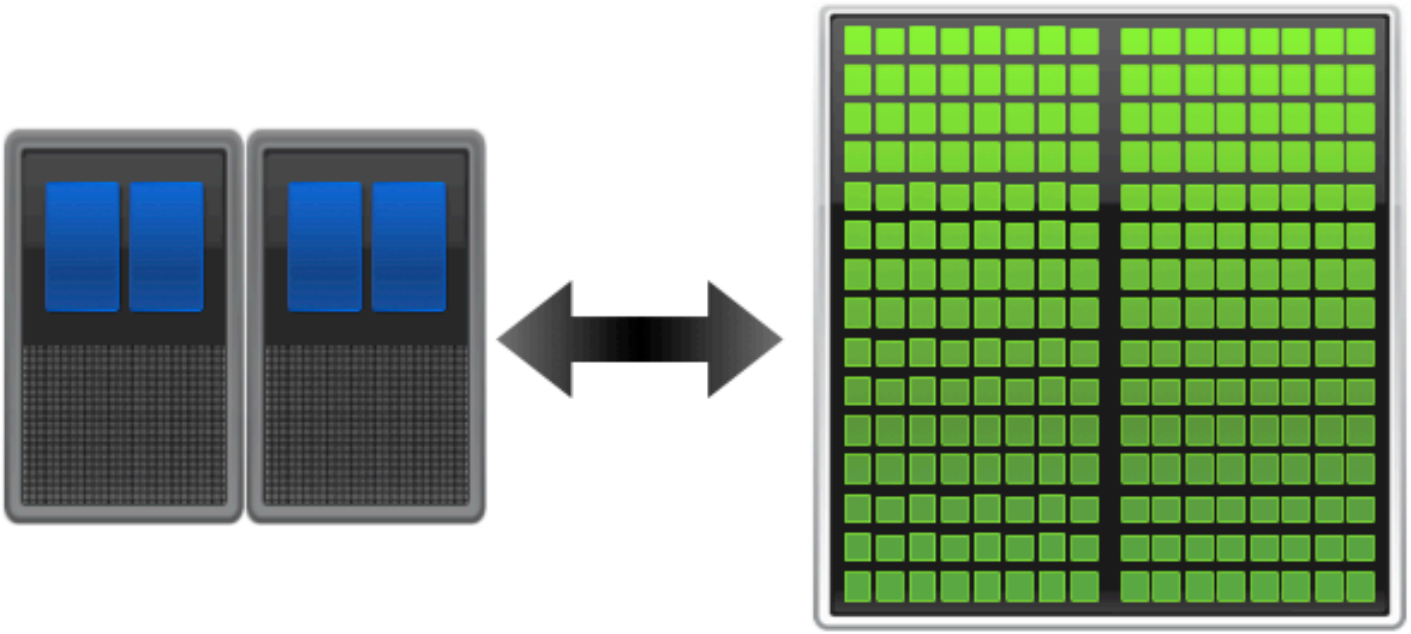


Multicore

Manycore

- Multicore: yoke of oxen
  - Each core optimized for executing a single thread
- Manycore: flock of chickens
  - Cores optimized for aggregate throughput, deemphasizing individual performance

Heterogeneous Parallel Computing



Multicore CPU

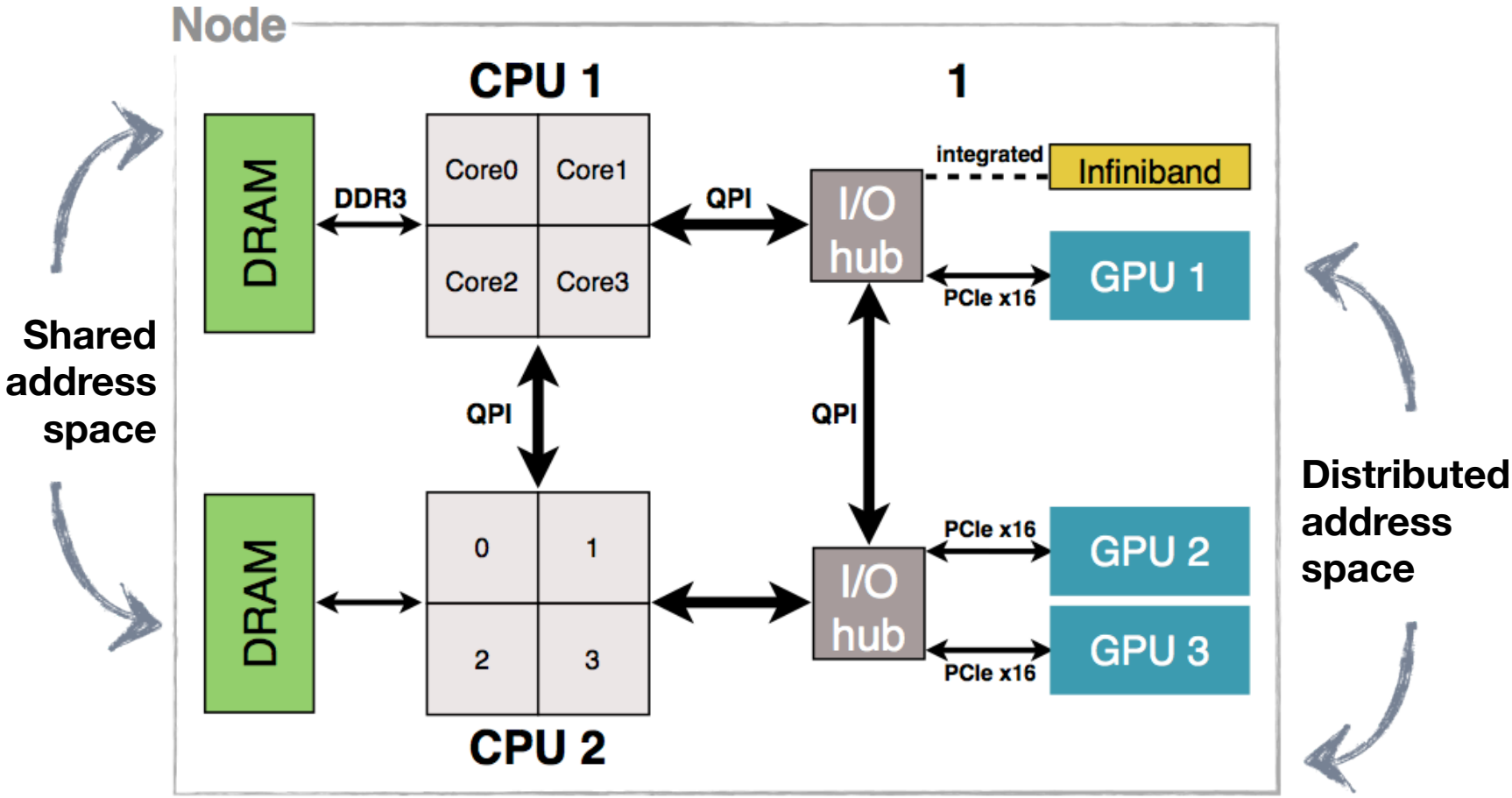
Fast Serial Processing

Manycore GPU

Scalable Parallel Processing

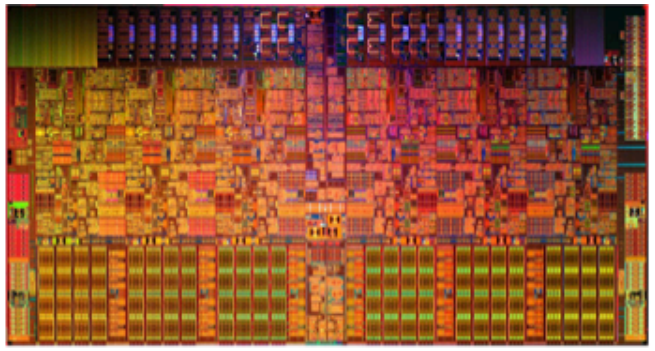
Multicore CPUs Connected to Manycore GPUs

- Currently, GPU devices are connected to CPUs over I/O

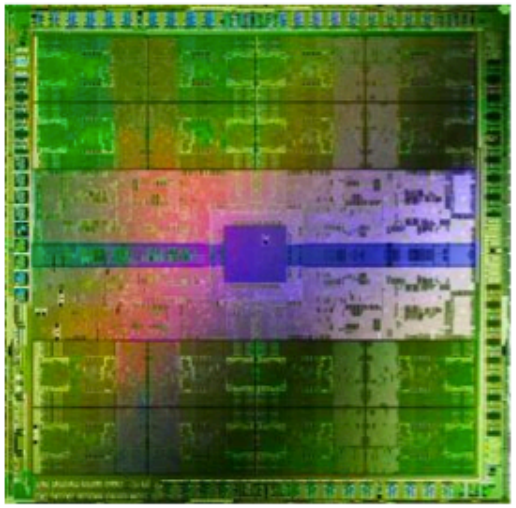


Multicore and Manycore – Real Chips

Specifications	Westmere-EP	Fermi (Tesla C2050)
Processing Elements	6 cores, 2 issue, 4 way SIMD @3.46 GHz	14 SMs, 2 issue, 16 way SIMD @1.15 GHz
Resident Strands/ Threads (max)	6 cores, 2 threads, 4 way SIMD: 48 strands	14 SMs, 48 SIMD vectors, 32 way SIMD: 21504 threads
SP GFLOP/s	166	1030
Memory Bandwidth	32 GB/s	144 GB/s
Register File	6 kB (?)	1.75 MB
Local Store/L1 Cache	192 kB	896 kB
L2 Cache	1536 kB	0.75 MB
L3 Cache	12 MB	-
# transistors & area:	**1.2 B, 240 mm^2**	**3 B, 520 mm^2**
thermal design power:	**130 Watts**	**160+ Watts? (240 W/card)**



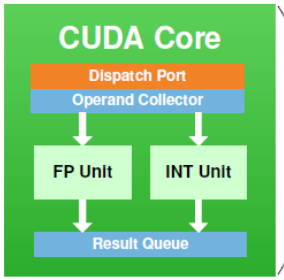
Westmere-EP (32nm)

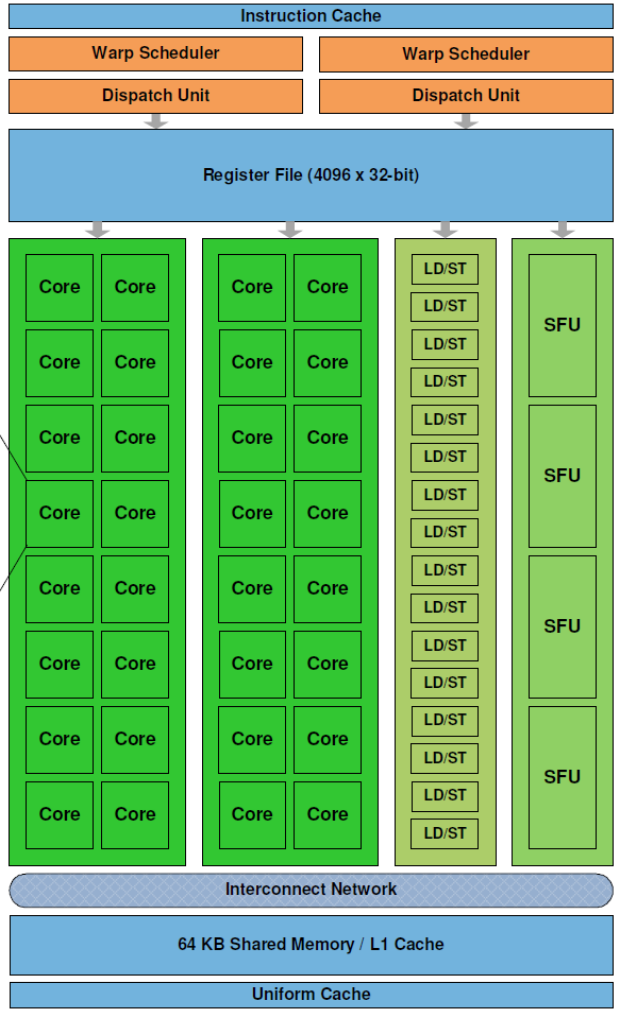


Fermi (40nm)

NVIDIA Fermi

- 3B transistors in 40nm
- 512 CUDA Cores
  - New IEEE 754-2008 floating-point standard
    - FMA
    - 8× the peak double precision arithmetic performance over NVIDIA's last generation GPU
  - 32 cores per SM, 21k threads per chip
- 384b GDDR5, 6 GB capacity
  - 178 GB/s memory BW
- C/M2090
  - 665 GigaFLOPS DP, 6GB
  - ECC Register files, L1/L2 caches, shared memory and DRAM





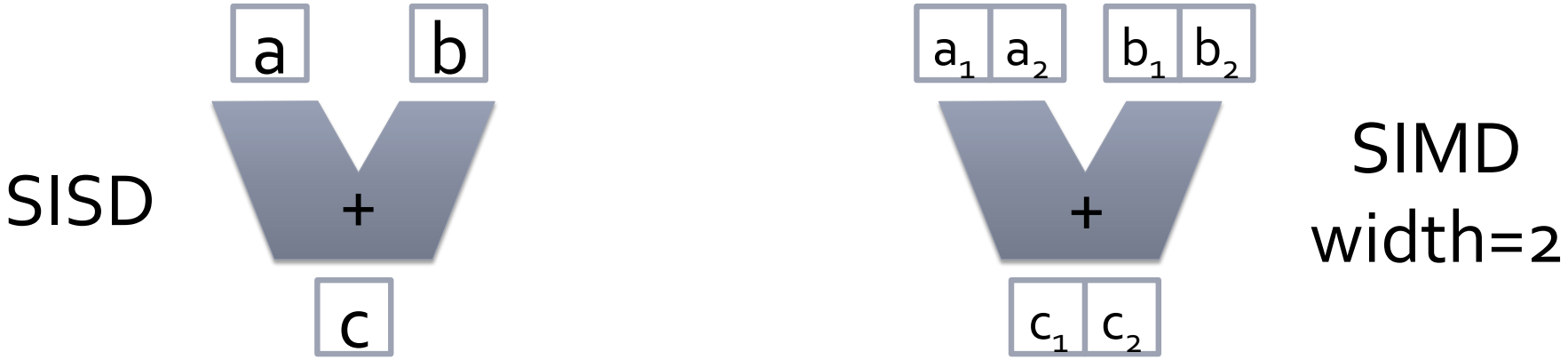
Why Heterogeneity?

- Different goals produce different designs
  - Manycore assumes work load is highly parallel
  - Multicore must be good at everything, parallel or not

- Multicore: minimize latency experienced by 1 thread
  - lots of big on-chip caches
  - extremely sophisticated control

- Manycore: maximize throughput of all threads
  - lots of big ALUs
  - multithreading can hide latency … so skip the big caches
  - simpler control, cost amortized over ALUs via SIMD

SIMD



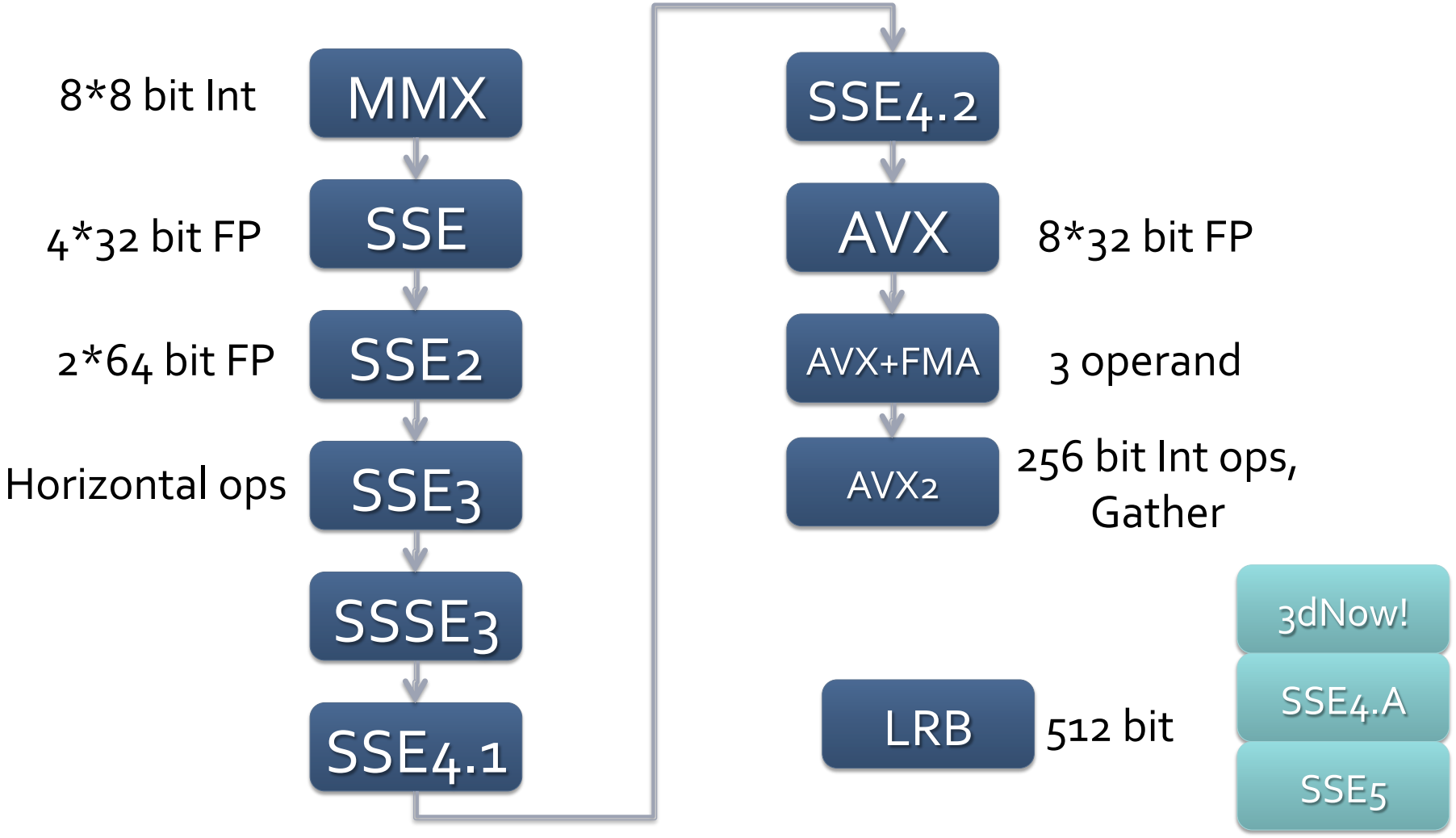
- Single Instruction Multiple Data architectures make use of data parallelism
- We care about SIMD because of area and power efficiency concerns
  - Amortize control overhead over SIMD width
- Parallelism exposed to programmer & compiler

SIMD – Neglected Parallelism

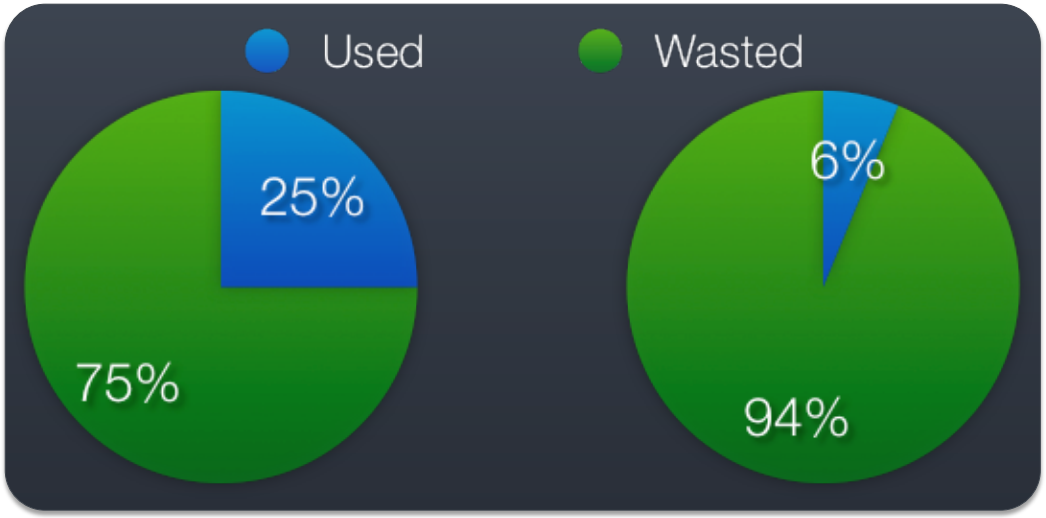
- It is difficult for a compiler to exploit SIMD
- How do you deal with sparse data & branches?
  - Many languages (like C) are difficult to vectorize
  - Fortran is somewhat better
- Most common solution:
  - Either forget about SIMD
    - Pray the autovectorizer likes you
  - Or instantiate intrinsics (assembly language)
  - Requires a new code version for every SIMD extension

Arguably much better, with recent extensions

A Brief History of x86 SIMD Extensions



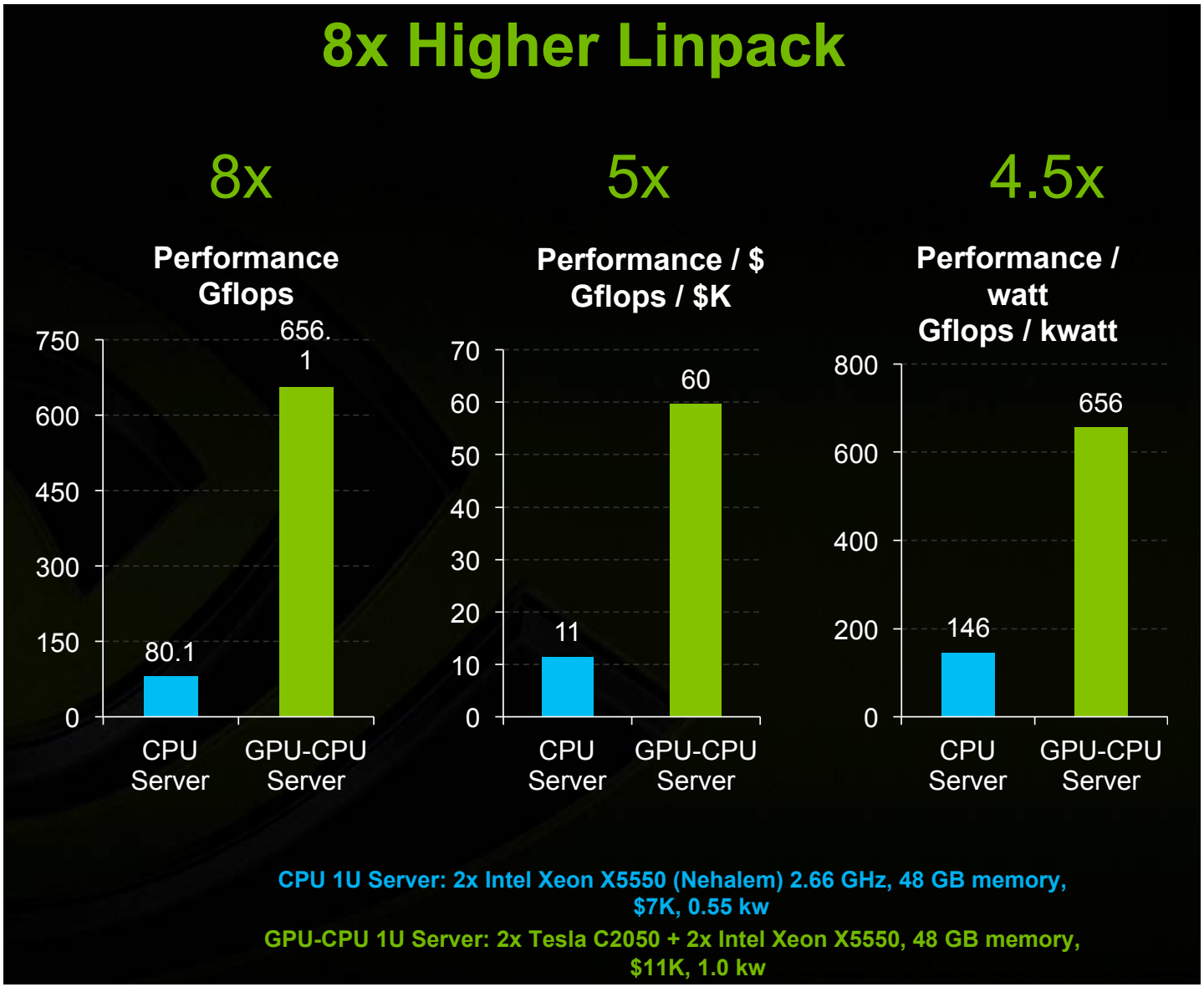
What to do with SIMD?



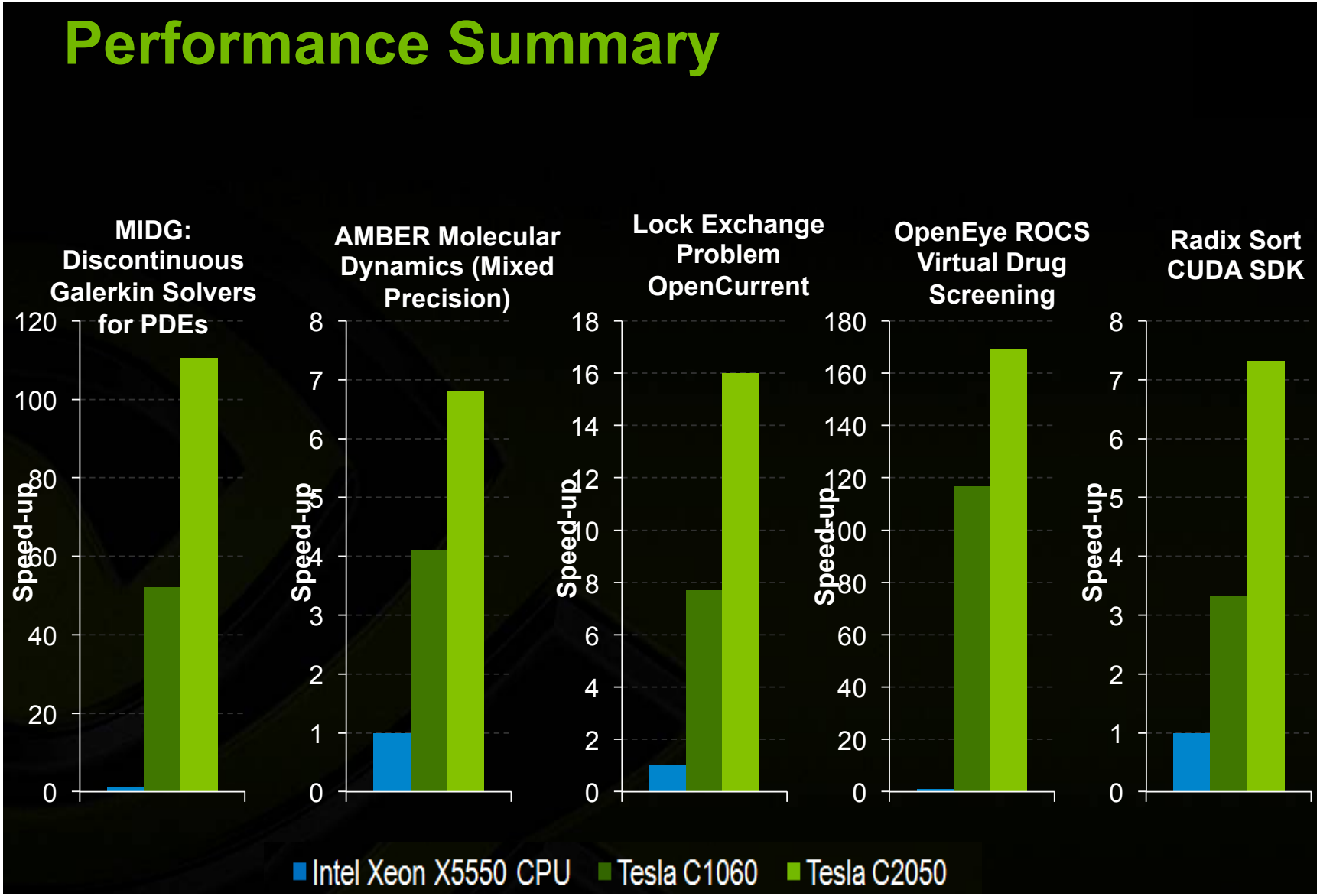
4 way SIMD (SSE) 16 way SIMD (LRB)

- Neglecting SIMD is becoming more expensive
  - AVX: 8 way SIMD, Larrabee: 16 way SIMD, Nvidia: 32 way SIMD, ATI: 64 way SIMD
- This problem composes with thread level parallelism
- We need a programming model which addresses both problems

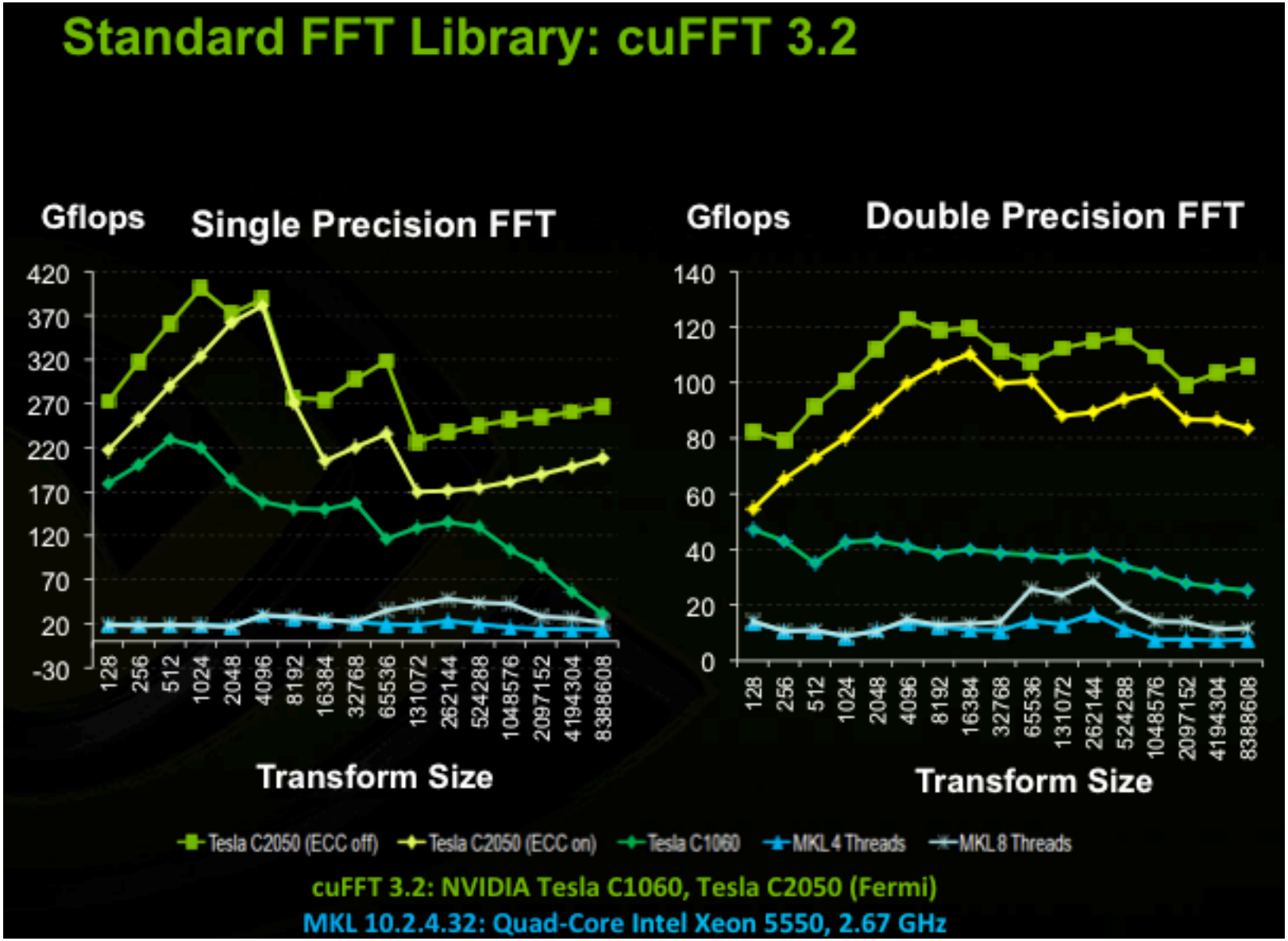
Manycore GPU Performance



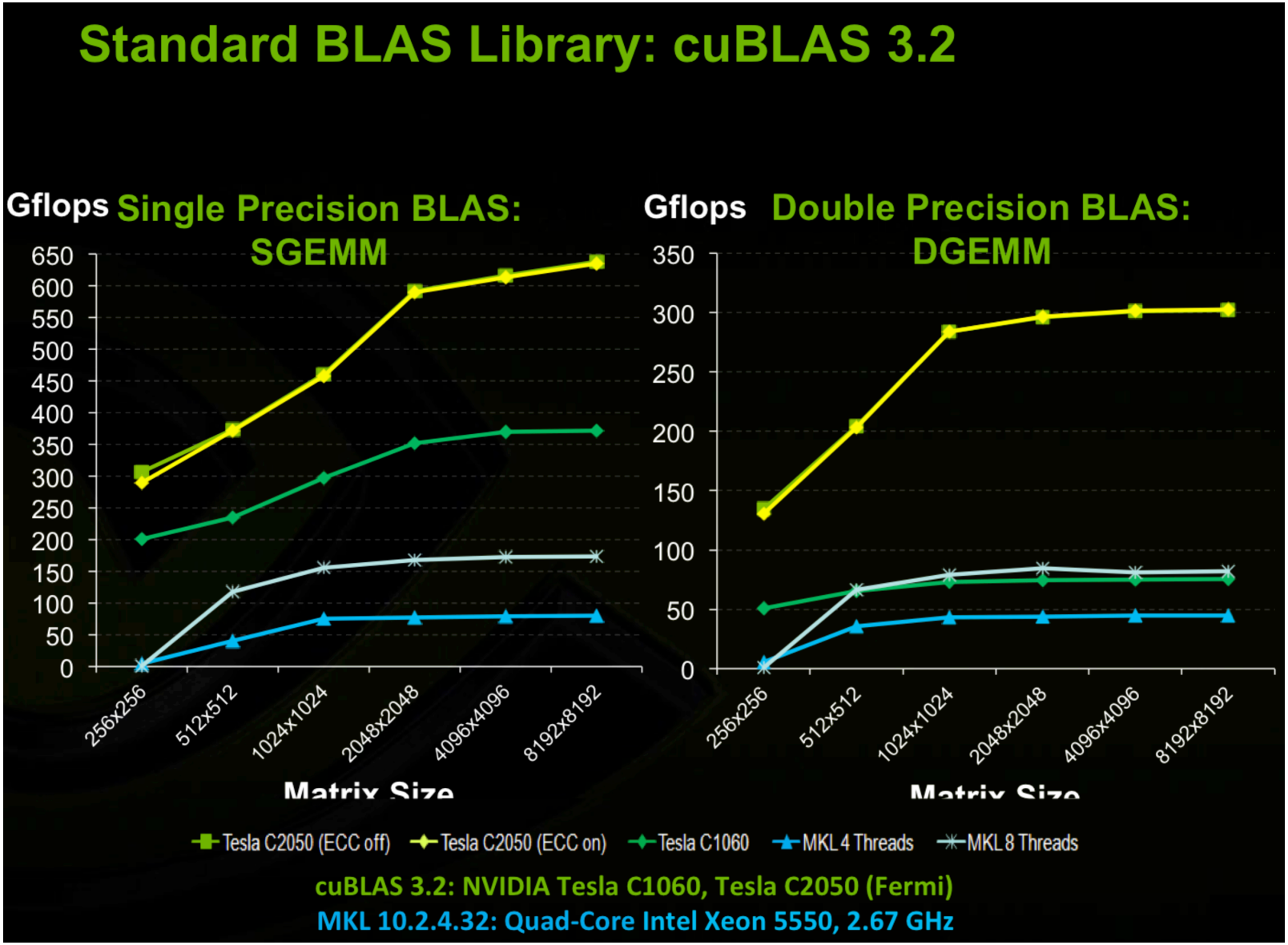
Manycore GPU Performance (2)



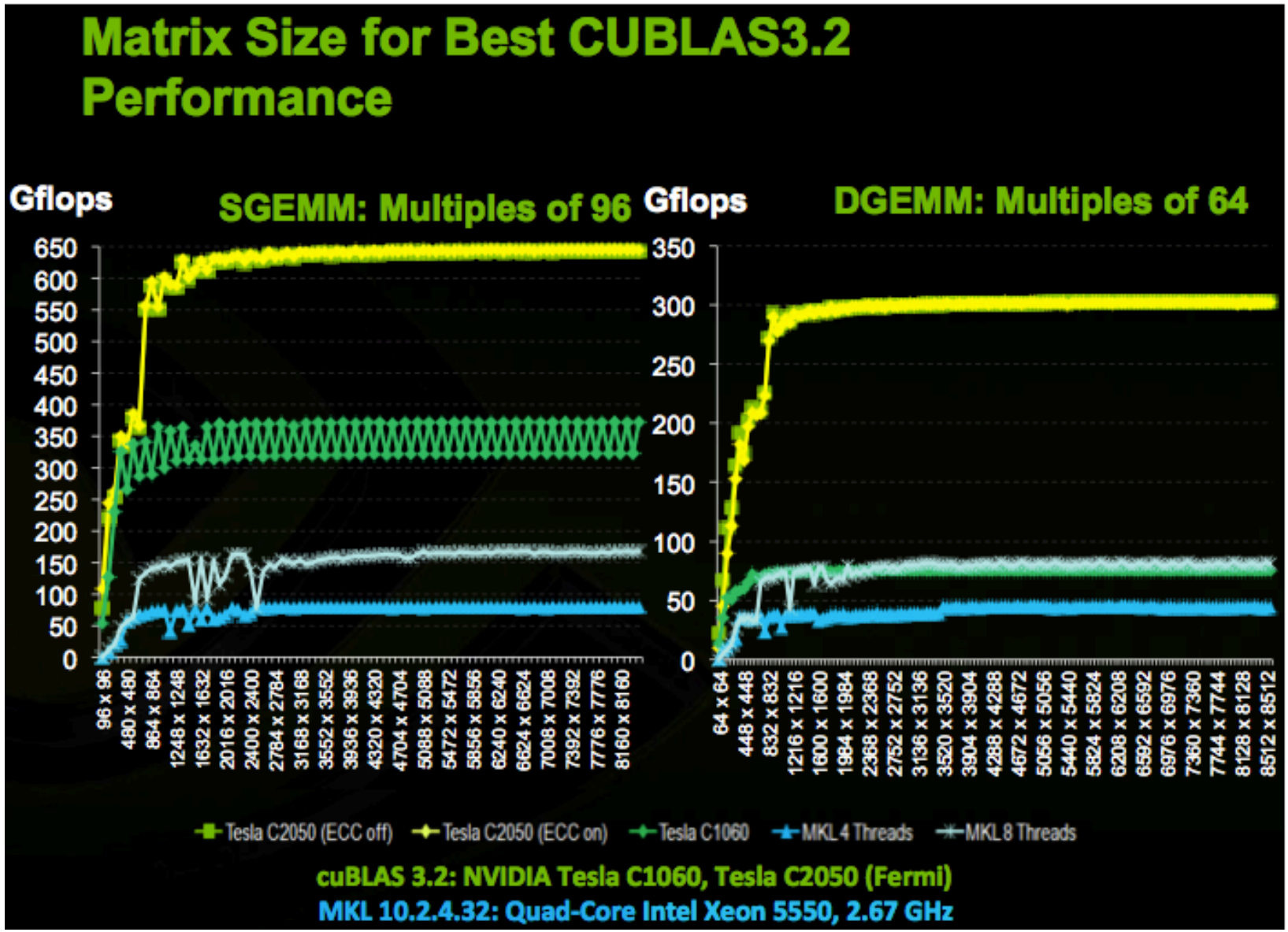
Manycore GPU Performance (3)



Manycore GPU Performance (4)



Manycore GPU Performance (5)



The CUDA Programming Model

- CUDA is a recent programming model, designed for
  - Manycore architectures
  - Wide SIMD parallelism
  - Scalability

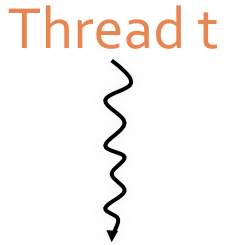
- CUDA provides:
  - A thread abstraction to deal with SIMD
  - Synchronization & data sharing between small groups of threads

- CUDA programs are written in C++ with minimal extensions

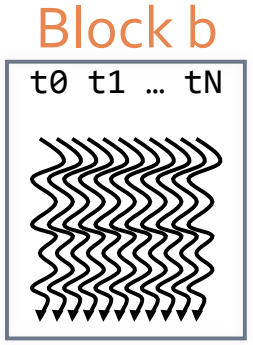
- OpenCL is inspired by CUDA, but HW & SW vendor neutral
  - Similar programming model, C only for device code

Hierarchy of Concurrent Threads

- Parallel kernels composed of many threads
  - all threads execute the same sequential program



- Threads are grouped into thread blocks
  - threads in the same block can cooperate



- Threads/blocks have unique IDs

What is a CUDA Thread?

- Independent thread of execution
  - has its own PC, variables (registers), processor state, etc.
  - no implication about how threads are scheduled

- CUDA threads might be physical threads
  - as mapped onto NVIDIA GPUs

- CUDA threads might be virtual threads
  - might pick 1 block = 1 physical thread on multicore CPU

What is a CUDA Thread Block?

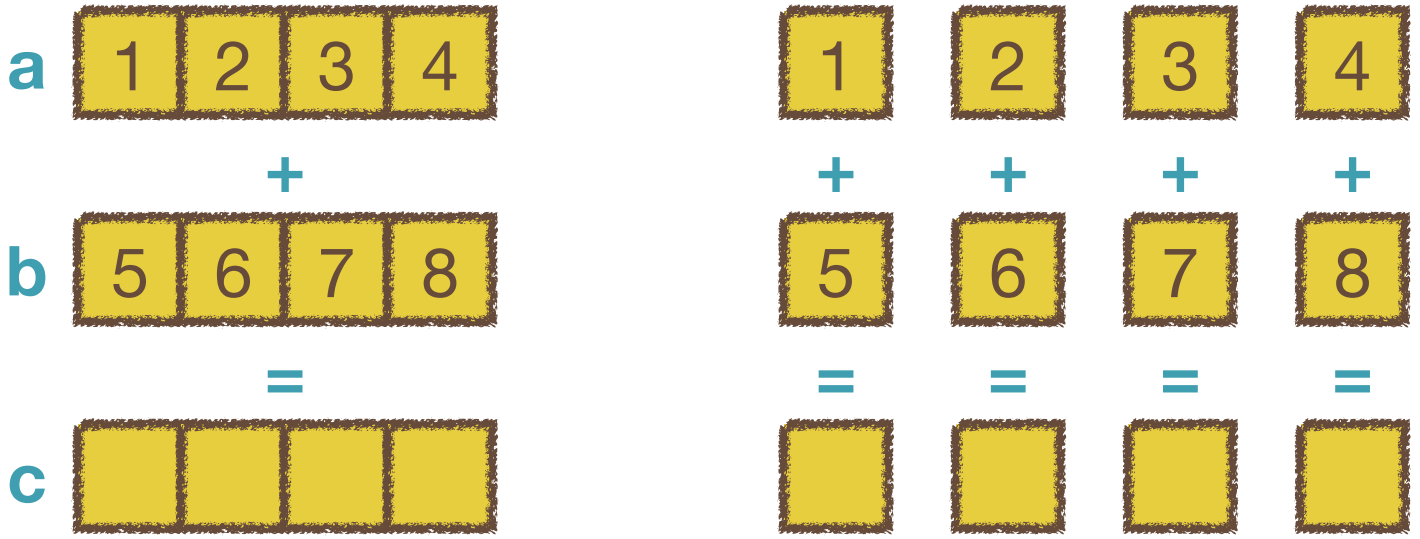
- Thread block = a (data) parallel task
  - all blocks in kernel have the same entry point
  - but may execute any code they want

- Thread blocks of kernel must be independent tasks
  - program valid for ***any interleaving*** of block executions

What CUDA Supports

- Thread parallelism
  - each thread is an independent thread of execution
- Data parallelism
  - across threads in a block
  - across blocks in a kernel
- Task parallelism
  - different blocks are independent
  - independent kernels executing in separate streams

SIMD (SSE) View versus SIMT (CUDA) View




```
__m128 a = _mm_set_ps (4, 3, 2, 1);
__m128 b = _mm_set_ps (8, 7, 6, 5);
__m128 c = _mm_add_ps (a, b);
```

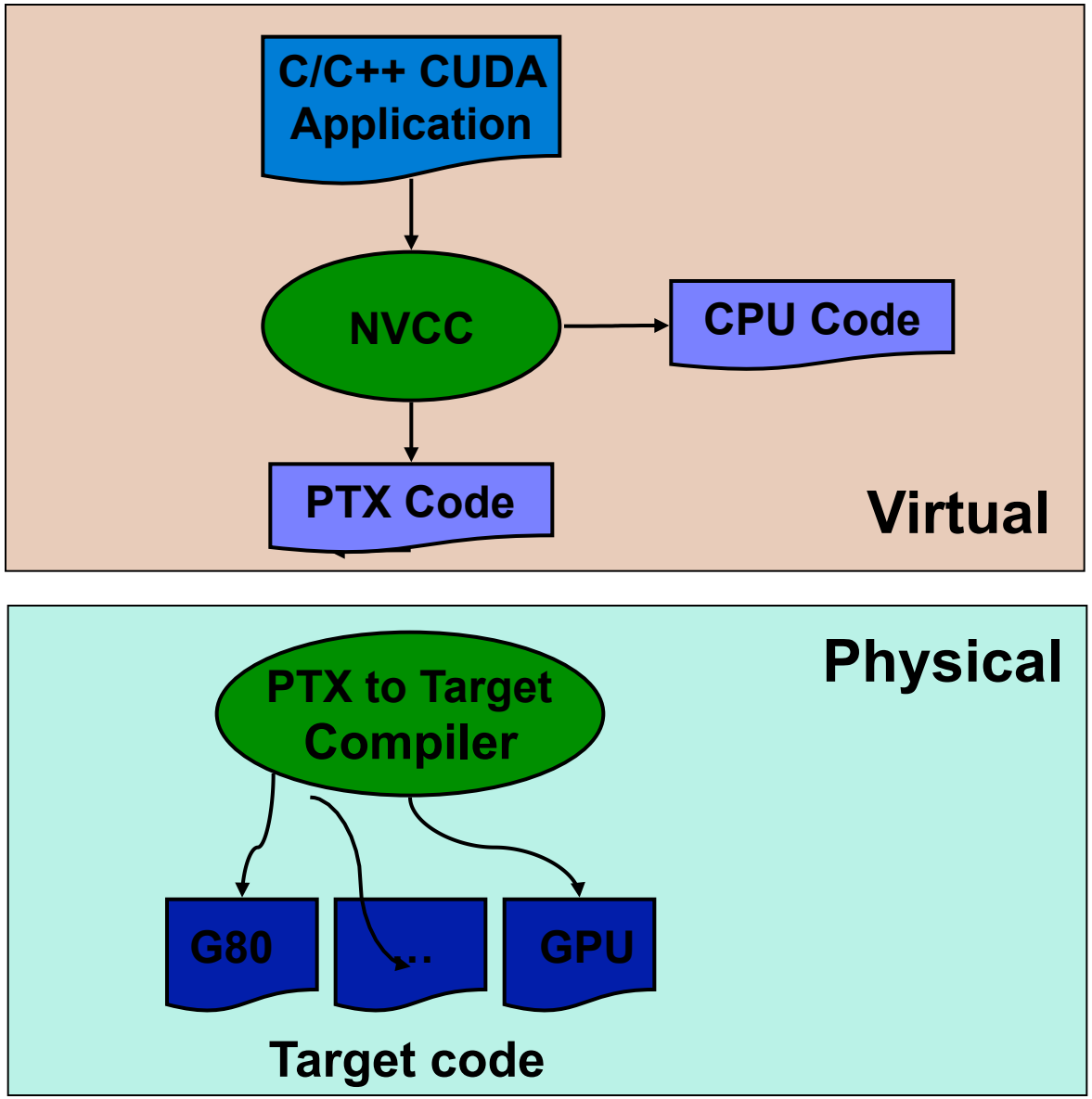
```
float a[4] = {1, 2, 3, 4},
      b[4] = {5, 6, 7, 8}, c[4];

// …
// Define a compute kernel, which
// a fine-grained thread executes.
{
  int id = … ;  // my thread ID
  c[id] = a[id] + b[id];
}
```

CUDA is Extended C

- **Declspecs**
  - **global, device, shared, local, constant**
- **Keywords**
  - **threadIdx, blockIdx**
- **Intrinsics**
  - **__syncthreads**
- **Runtime API**
  - **Memory, symbol, execution management**
- **Function launch**

```
__device__ float filter[N];

__global__ void convolve (float *image)  {

  __shared__ float region[M];
  ...

  region[threadIdx] = image[i];

  __syncthreads()
  ...

  image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)

// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```

CUDA Compilaton



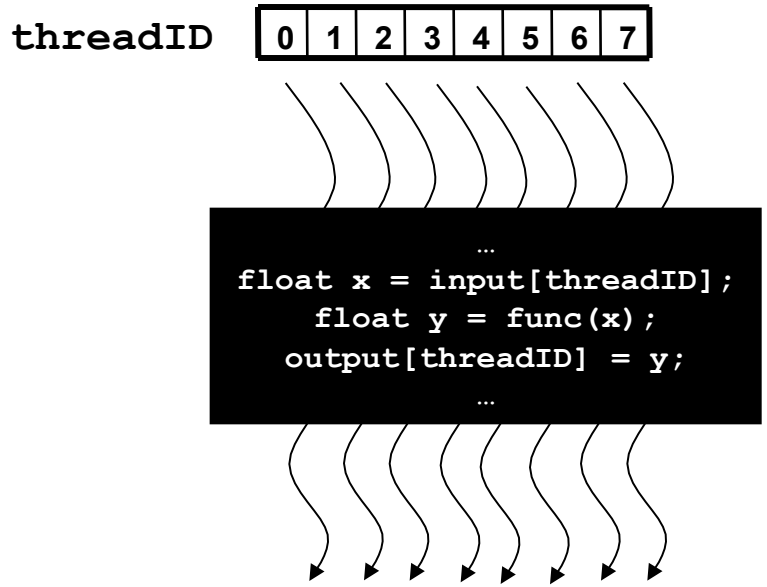
- Parallel Thread eXecution (PTX)
  - Virtual Machine and ISA
  - Programming model
  - Execution resources and state

CUDA Compilation (2)

- Any source file containing CUDA language extensions must be compiled with NVCC
- NVCC is a compiler driver
  - Works by invoking all the necessary tools and compilers like cudacc, g++, cl, ...
- NVCC outputs:
  - C code (host CPU Code)
    - Must then be compiled with the rest of the application using another tool
  - PTX
    - Object code directly
    - Or, PTX source, interpreted at runtime

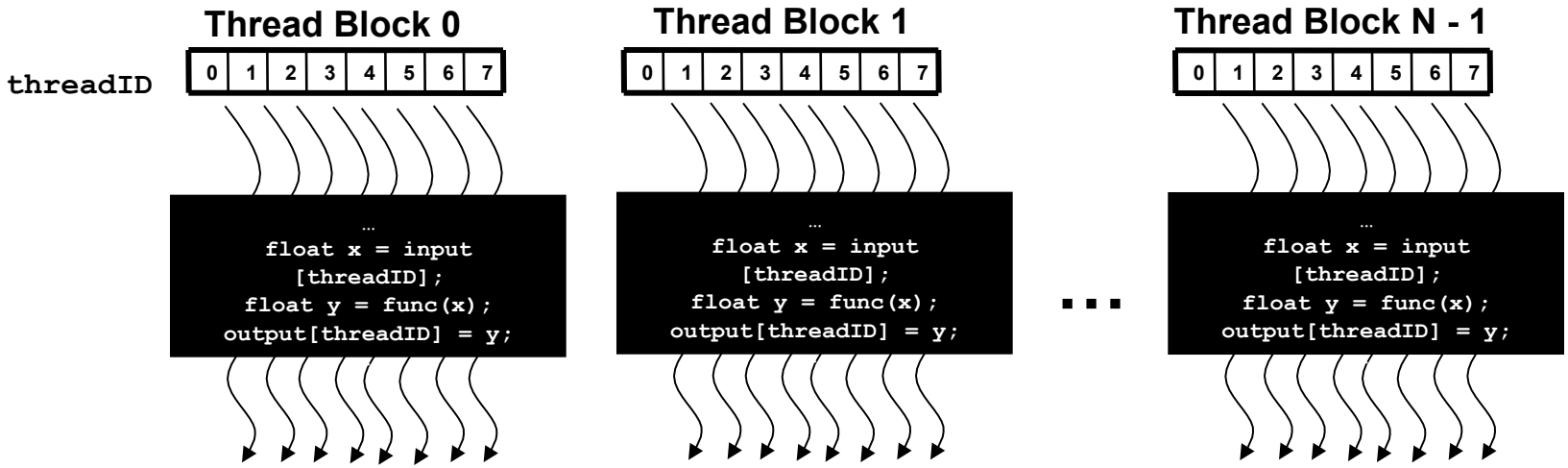
Array of Parallel Threads

- A CUDA kernel is executed by an array of threads
  - All threads run the same code (SPMD)
  - Each thread has an ID that it uses to compute memory addresses and make control decisions



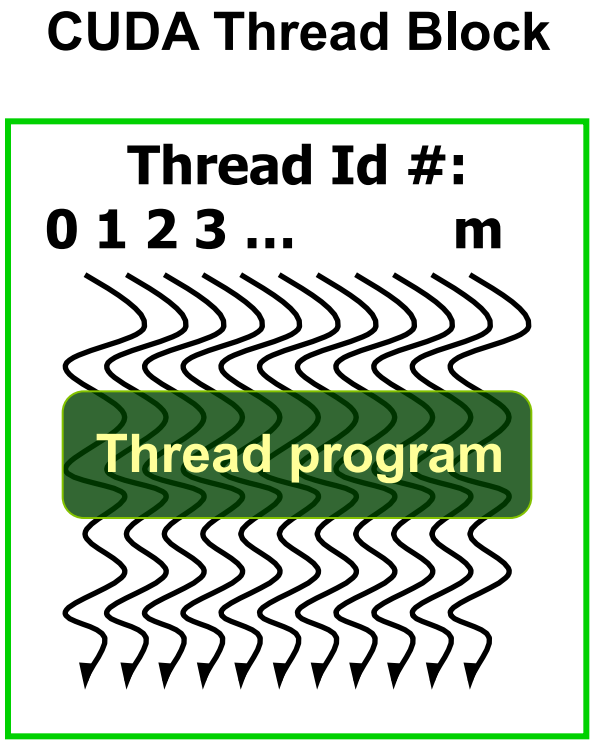
Thread Blocks – Scalable Cooperation

- Divide monolithic thread array into multiple blocks
  - Threads within a block cooperate via shared memory, atomic operations and barrier synchronization
  - Threads in different blocks cannot cooperate



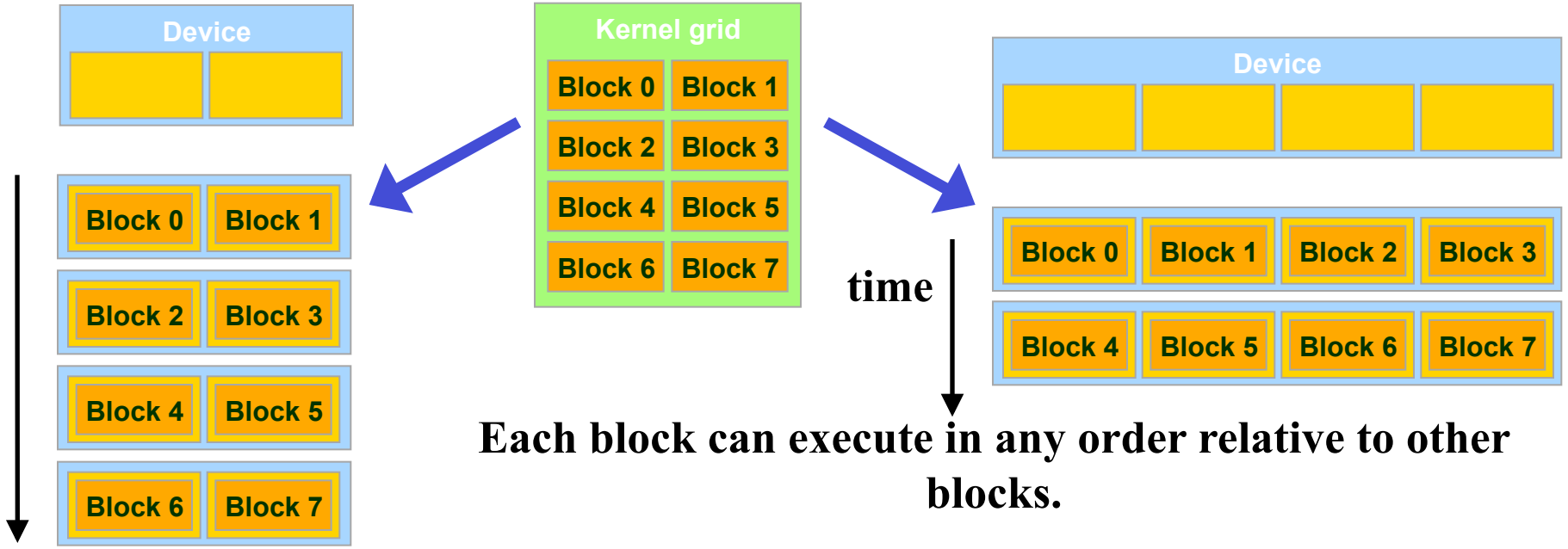
CUDA Thread Block

- All threads in a block execute the same kernel program (SPMD)
- Programmer declares block:
  - Block size 1 to 512 concurrent threads
  - Block shape 1D, 2D, or 3D
  - Block dimensions in threads
- Threads have thread id numbers within block
  - Thread program uses thread id to select work and address shared data
- Threads in the same block share data and synchronize while doing their share of the work
- Threads in different blocks cannot cooperate
  - Each block can execute in any order relative to other blocs!



Transparent Scalability

- Hardware is free to assigns blocks to any processor at any time
  - A kernel scales across any number of parallel processors



Each block can execute in any order relative to other blocks.

Hello World – Vector Addition

```
//Compute vector sum C=A+B
//Each thread performs one pairwise addition
__global__ void vecAdd(float* a, float* b, float* c) {
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   c[i] = a[i] + b[i];
}

int main() {
   //Run N/256 blocks of 256 threads each
   vecAdd<<<N/256, 256>>>(d_a, d_b, d_c);
}
```

Synchronization

- Threads within a block may synchronize with barriers

```
… Step 1 …
__syncthreads();
… Step 2 …
```

- Blocks coordinate via atomic memory operations
  - e.g., increment shared queue pointer with `atomicInc()`

- Implicit barrier between dependent kernels

```
vec_minus<<<nblocks, blksize>>>(a, b, c);

vec_dot<<<nblocks, blksize>>>(c, c);
```

Blocks Must Be Independent

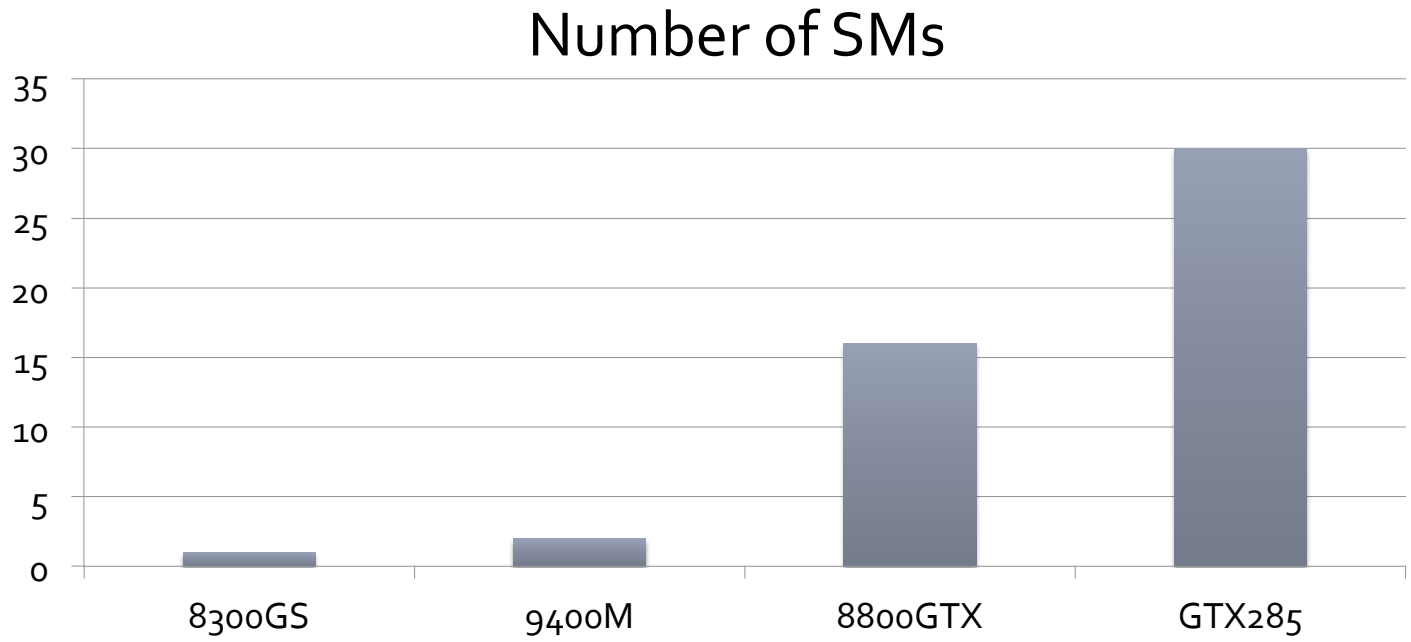
- Any possible interleaving of blocks should be valid
  - presumed to run to completion without pre-emption
  - can run in any order
  - can run concurrently OR sequentially

- Blocks may coordinate but not synchronize
  - shared queue pointer: OK
  - shared lock: BAD … can easily deadlock

- Independence requirement gives scalability

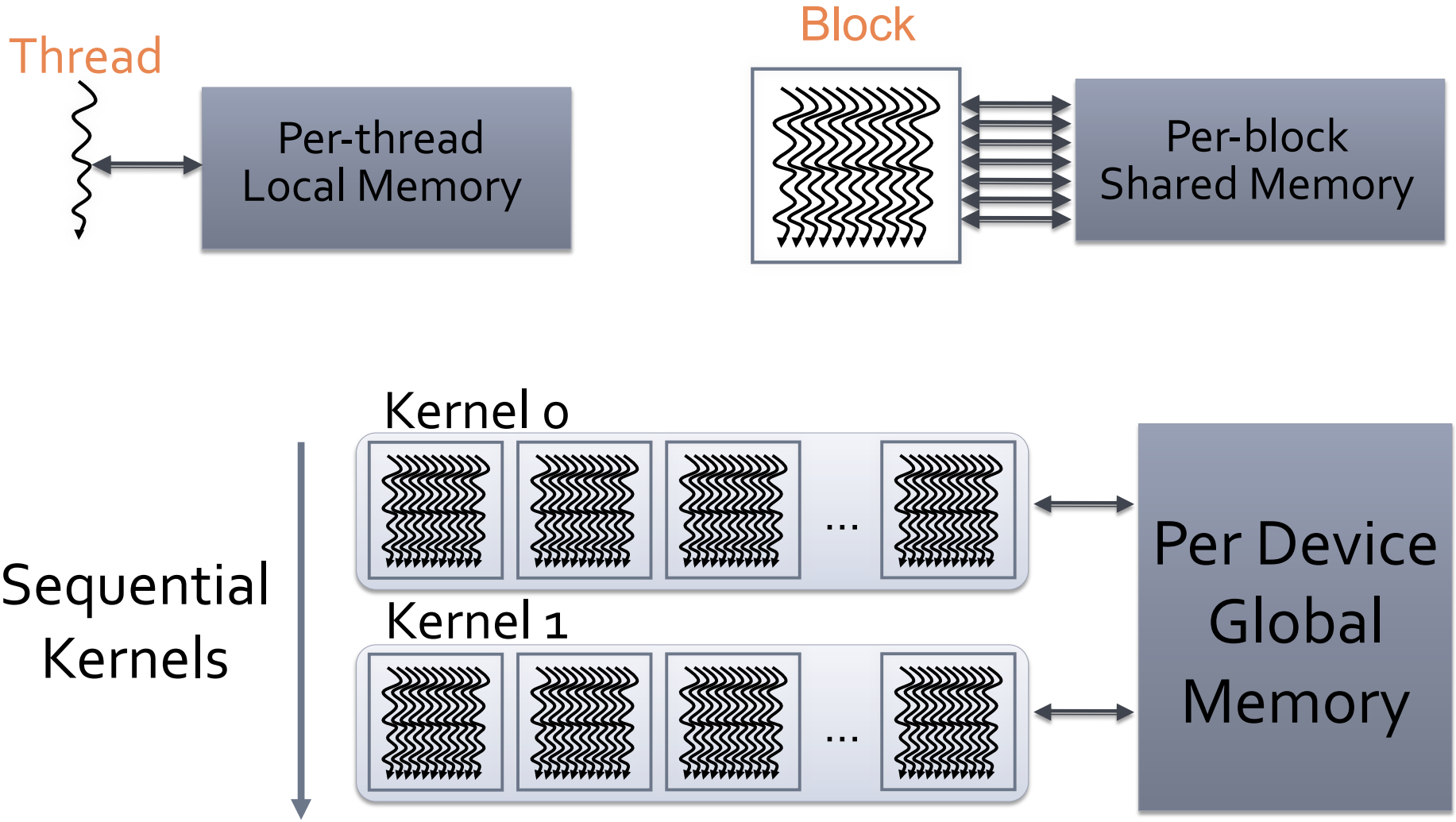
Scalability

- Manycore chips exist in a diverse set of configurations

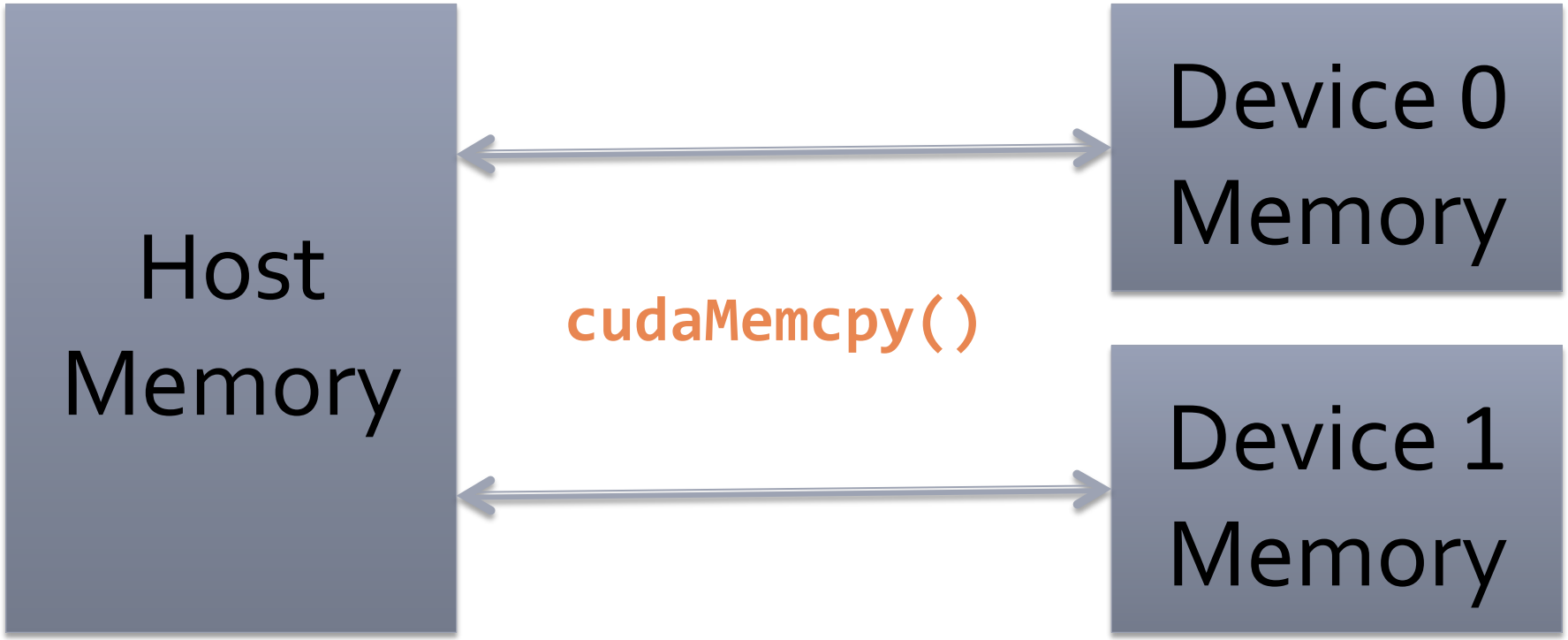


- CUDA allows one binary to target all these chips
- Thread blocks bring scalability!

Memory Model



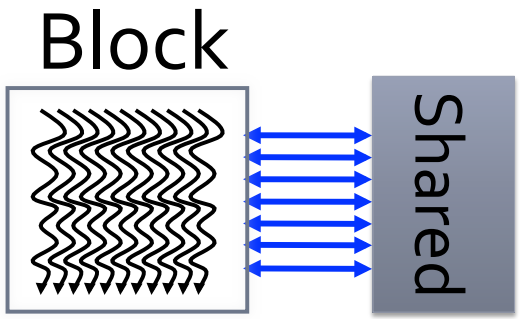
Memory Model – CPU to/from GPU Device



Hello World – Managing Data

```
int main() {
   int N = 256 * 1024;
   float* h_a = malloc(sizeof(float) * N);
   //Similarly for h_b, h_c. Initialize h_a, h_b

   float *d_a, *d_b, *d_c;
   cudaMalloc(&d_a, sizeof(float) * N);
   //Similarly for d_b, d_c

   cudaMemcpy(d_a, h_a, sizeof(float) * N, cudaMemcpyHostToDevice);
   //Similarly for d_b

   //Run N/256 blocks of 256 threads each
   vecAdd<<<N/256, 256>>>(d_a, d_b, d_c);

   cudaMemcpy(h_c, d_c, sizeof(float) * N, cudaMemcpyDeviceToHost);
}
```

Using per-Block Shared Memory



- Variables shared across block

```
__shared__ int *begin, *end;
```

- Scratchpad memory

```
__shared__ int scratch[BLOCKSIZE];
scratch[threadIdx.x] = begin[threadIdx.x];
// … compute on scratch values …
begin[threadIdx.x] = scratch[threadIdx.x];
```

- Communicating values between threads

```
scratch[threadIdx.x] = begin[threadIdx.x];
__syncthreads();
int left = scratch[threadIdx.x - 1];
```

- Per-block shared memory is faster than L1 cache, slower than register file
- It is relatively small: register file is 2-4x larger

CUDA – Minimal Extensions to C/C++

- Declaration specifiers to indicate where things live

```
__global__ void KernelFunc(...);   // kernel callable from host
__device__ void DeviceFunc(...);   // function callable on device
__device__ int  GlobalVar;         // variable in device memory
__shared__ int  SharedVar;         // in per-block shared memory
```

- Extend function invocation syntax for parallel kernel launch

```
KernelFunc<<<500, 128>>>(...);     // 500 blocks, 128 threads each
```

- Special variables for thread identification in kernels

```
dim3 threadIdx;  dim3 blockIdx;  dim3 blockDim;
```

- Intrinsics that expose specific operations in kernel code

```
__syncthreads();                   // barrier synchronization
```

CUDA – Features Available on GPU

- Double and single precision (IEEE compliant)
- Standard mathematical functions
  - `sinf`, `powf`, `atanf`, `ceil`, `min`, `sqrtf`, etc.
- Atomic memory operations
  - `atomicAdd`, `atomicMin`, `atomicAnd`, `atomicCAS`, etc.
- These work on both global and shared memory

CUDA – Runtime Support

- Explicit memory allocation returns pointers to GPU memory
  - `cudaMalloc()`, `cudaFree()`
- Explicit memory copy for host ↔ device, device ↔ device
  - `cudaMemcpy()`, `cudaMemcpy2D()`, …
- Texture management
  - `cudaBindTexture()`, `cudaBindTextureToArray()`, …
- OpenGL & DirectX interoperability
  - `cudaGLMapBufferObject()`, `cudaD3D9MapVertexBuffer()`, …

Imperatives for Efficient CUDA Code

- Expose abundant fine-grained parallelism
  - need 1000's of threads for full utilization
- Maximize on-chip work
  - on-chip memory orders of magnitude faster
- Minimize execution divergence
  - SIMT execution of threads in 32-thread warps
- Minimize memory divergence
  - warp loads and consumes complete 128-byte cache line

Mapping CUDA to NVIDIA GPUs

- CUDA is designed to be functionally forgiving
  - First priority: make things work. Second: get performance.
- However, to get good performance, one must understand how CUDA is mapped to Nvidia GPUs
- Threads: each thread is a SIMD vector lane
- Warps: A SIMD instruction acts on a "warp"
  - Warp width is 32 elements: ***LOGICAL*** SIMD width
- Thread blocks: Each thread block is scheduled onto an SM
  - Peak efficiency requires multiple thread blocks per SM

Mapping CUDA to NVIDIA GPUs (2)

- The GPU is very deeply pipelined to maximize throughput
- This means that performance depends on the number of thread blocks which can be allocated on a processor
- Therefore, resource usage costs performance:
  - More registers => Fewer thread blocks
  - More shared memory usage => Fewer thread blocks
- It is often worth trying to reduce register count in order to get more thread blocks to fit on the chip
  - For Fermi, target 20 registers or less per thread for full occupancy

Occupancy (Constants for Fermi)

- The Runtime tries to fit as many thread blocks simultaneously as possible on to an SM
  - The number of simultaneous thread blocks (B) is ≤ 8
- The number of warps per thread block (T) ≤ 32
- B * T ≤ 48 (Each SM has scheduler space for 48 warps)
- The number of threads per warp (V) is 32
- B * T * V * Registers per thread ≤ 32768
- B * Shared memory (bytes) per block ≤ 49152/16384
  - Depending on Shared memory/L1 cache configuration

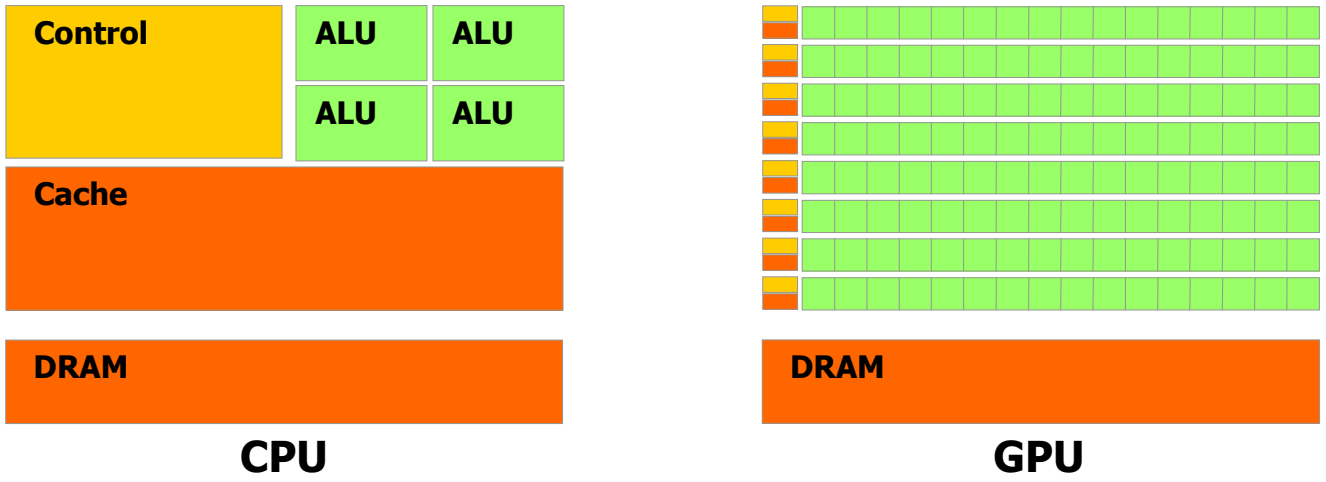
- Occupancy is reported as B * T / 48

SIMD and Control Flow

- Nvidia GPU hardware handles control flow divergence and reconvergence
  - Write scalar SIMD code, the hardware schedules the SIMD execution
  - One caveat: __syncthreads() can't appear in a divergent path
    - This will cause programs to hang
  - Good performing code will try to keep the execution convergent within a warp
    - Warp divergence only costs because of a finite instruction cache

Memory, Memory, Memory

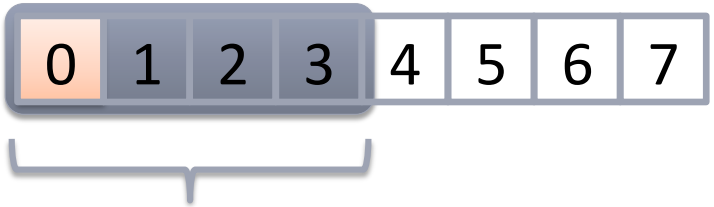
- A many core processor ≡ A device for turning a compute bound problem into a memory bound problem



- Lots of processors, only one socket
- Memory concerns dominate performance tuning

Memory is SIMD Too!

- Virtually all processors have SIMD memory subsystems



- This has two effects:
  - Sparse access wastes bandwidth

2 words used, 8 words loaded:
¼ effective bandwidth

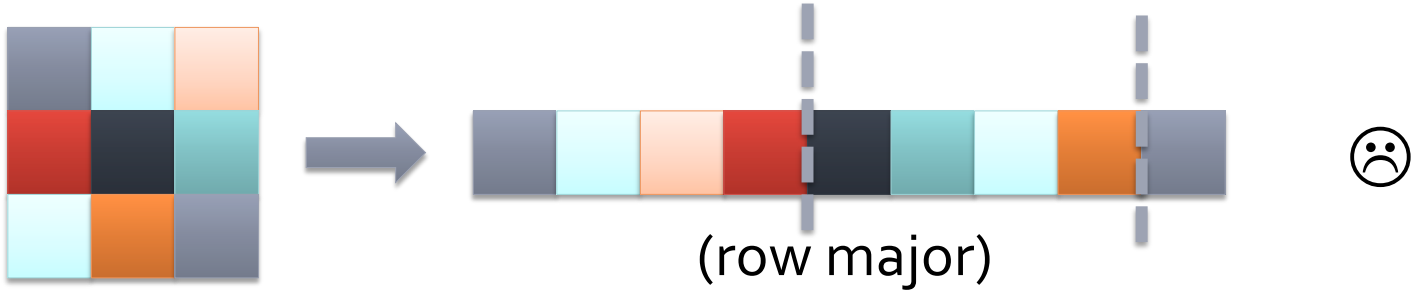
  - Unaligned access wastes bandwidth

4 words used, 8 words loaded:
½ effective bandwidth

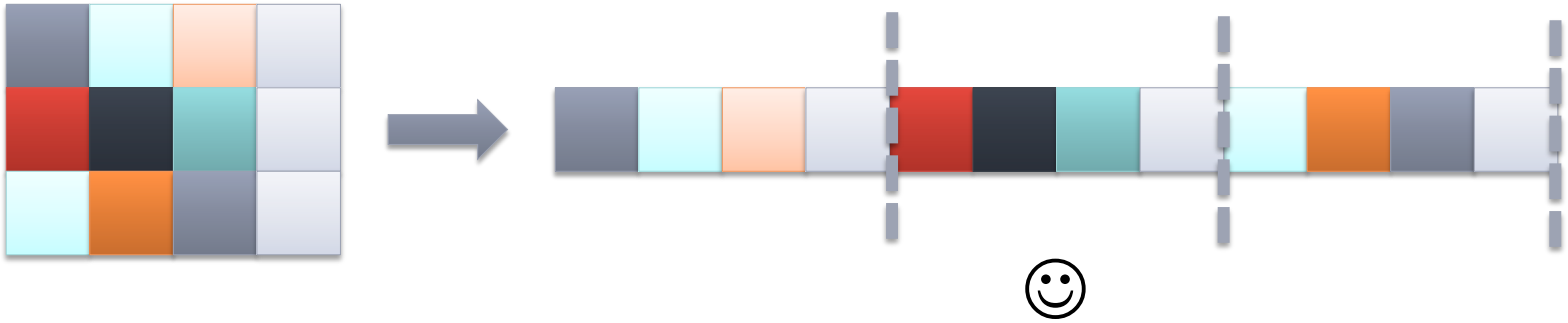
Coalescing

- GPUs and CPUs both perform memory transactions at a larger granularity than the program requests ("cache line")
- GPUs have a "coalescer", which examines memory requests dynamically and coalesces them
- To use bandwidth effectively, when threads load, they should:
  - Present a set of unit strided loads (dense accesses)
  - Keep sets of loads aligned to vector boundaries

Data Structure Padding

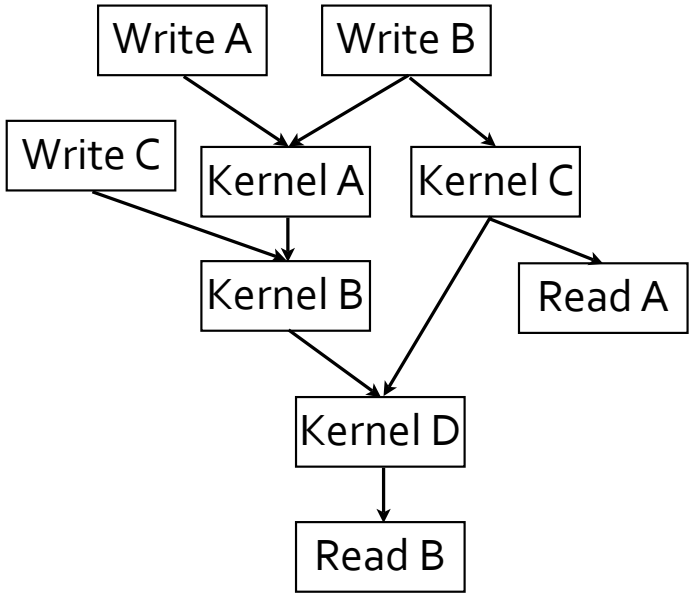


- Multidimensional arrays are usually stored as monolithic vectors in memory
- Care should be taken to assure aligned memory accesses for the necessary access pattern



OpenCL

- OpenCL is supported by AMD {CPUs, GPUs} and Nvidia
  - Intel, Imagination Technologies (purveyor of GPUs for iPhone/OMAP/etc.) are also on board
- OpenCL's data parallel execution model mirrors CUDA, but with different terminology
- OpenCL has rich task parallelism model
- Runtime walks a dataflow DAG of kernels/memory transfers



Thrust

- There exist many tools and libraries for GPU programming

- Thrust is now part of the CUDA SDK
- C++ libraries for CUDA programming, inspired by STL
- Many important algorithms:
  - reduce, sort, reduce_by_key, scan, …
- Dramatically reduces overhead of managing heterogeneous memory spaces
- Includes OpenMP backend for multicore programming

Thrust Hello World

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>
#include <cstdlib>

int main(void)
{
    // generate 32M random numbers on the host
    thrust::host_vector<int> h_vec(32 << 20);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);

    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;

    // sort data on the device (846M keys per sec on GeForce GTX 480)
    thrust::sort(d_vec.begin(), d_vec.end());

    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

    return 0;
}
```

Thrust saxpy

```
// C++ functor replaces __global__ function
struct saxpy
{
    float a;

    saxpy(float _a) : a(_a) {}

    __host__ __device__
    float operator()(float x, float y)
    {
        return a * x + y;
    }
};

transform(x.begin(), x.end(), y.begin(), y.begin(), saxpy(a));
```

Summary

- Manycore processors provide useful parallelism
- Programming models like CUDA and OpenCL enable productive parallel programming
- They abstract SIMD, making it easy to use wide SIMD vectors
- CUDA and OpenCL encourages SIMD friendly, highly scalable algorithm design and implementation
- Thrust is a productive C++ library for CUDA development

Next Class

- Preparation for midterm