MANY-CORE COMPUTING: GPU PROGRAMMING

Ana Lucia Varbanescu, UvA

14-Oct-2013

Original slides: Rob van Nieuwpoort, eScience Center
Schedule

1. Introduction, performance metrics
2. Programming many-cores (10/10)
3. CUDA Programming (14/10)
4. Case study: LOFAR telescope with many-cores (Rob van Nieuwpoort, on 17/10)
Before we start ...

- There were two assignments
  - Difference between AI and OI
  - Vectorization of complex-number division/multiplication

- No answers ... 😞
AI versus OI

- **AI:**
  - FLOPS: arithmetic operations
  - BW: all reads and writes

- **OI:**
  - FLOPS: all operations that involve computational units
  - BW: ONLY operations with DRAM

**Example:**

```latex
B[N-1] = \sqrt{\text{pow}(A[N-1],2) + \text{pow}(A[0],2)}; \\
\text{for (i=0; i<N-1; i++)} \\
\hspace{1cm} B[i] = \sqrt{\text{pow}(A[i],2) + \text{pow}(A[i+1],2)}; \\
\text{AI} = 1\text{ADD}/ (2\text{RD} + 1\text{WR}) = 1/3 \\
\text{OI} = (1\text{ADD} + 2\text{POW} + 1\text{SQRT}) / (1\text{RD} + 1\text{WR})
```
AI versus OI

In reality, we use a mix of the two and call it “arithmetic intensity”:

- Use all operations to compute FLOPs
  - For special functions: search specs *or* microbenchmark
- Use all read and write operations for BW
  - Use all reads and writes assumed from memory
  - Ignore ops from registers
  - Ignore ops from constant/local memory

Consequence: MemOPs are overestimated

- If app is compute-bound => not important
- If app is memory-bound => underestimate peak
  - Peak = memBW * OI = memBW * (FLOPS/MemOPS)
Complex numbers

- Two parts: real and imaginary
- Notation:
  - \( a+bi \) or \( a.re + a.im*i \)
  - \( i = \sqrt{-1} \Rightarrow i^2 = -1 \)
- Multiplication:
  \[
  (a.re+a.im*i) \cdot (b.re+b.im*i) = \\
  (a.re*b.re - a.im*b.im) + (a.re*b.im+a.im*b.re)*i
  \]
- Division:
  \[
  \frac{a.re+a.im*i}{b.re+b.im*i} = \\
  \frac{a.re*b.re + a.im*b.im}{b.re^2+b.im^2} + \\
  \frac{a.im*b.re - a.re*b.im}{b.re^2+b.im^2}*i
  \]
Vectorization [1/2]

- **Addressing by position:**
  
  - \( \text{vec1}.w, \text{vec1}.x, \text{vec1}.y, \text{vec1}.z = \text{vec1}[0],[1],[2],[3]. \)

- **Load/store:**
  
  - \( \text{vecA} = \text{load}(A); \)
  
  - \( \text{store}(\text{vecA}, A); \)

- **Addition, multiplication, subtraction, division:**
  
  - \( \text{vec3} = \text{add/mul/sub/div}(\text{vec1}, \text{vec2}); \)

- **Shuffle:**
  
  - \( \text{Vec3} = \text{shuffle}(\text{vec1},\text{vec2}, (a,b,c,d)); \)
    
    - \( \text{vec3}.w = \text{vec3}[0] = \text{vec1}[a]; \text{vec3}.x = \text{vec3}[1] = \text{vec1}[b]; \)
    
    - \( \text{vec3}.y = \text{vec3}[2] = \text{vec2}[c]; \text{vec3}.x = \text{vec3}[3] = \text{vec2}[d]; \)
for (j=0; j<N; j+=2) {
    vecA = load(A+2*j); vecB = load(B+2*j);
    // b.re*b.re, b.im*b.im
    vecT1=mul(vecB,vecB);
    // a.re*b.re, a.im*b.im
    vecT2=mul(vecA,vecB);
    // rotate A left and B right
    vecT3=shuffle(vecA,vecA,(1,2,3,0);
    vecT4=shuffle(vecB,vecB,(3,0,1,2);
    // b.re*a.im on positions 0 and 2
    vecT5=mul(vecB,vecT3);
    // b.im*a.re on positions 1 and 3
    vecT4=mul(vecA,vecT4);
    C[2*j] = (vecT2.w+vecT2.x)/(vecT1.w+vecT2.x)
    C[2*j+1] = ...
    C[2*j+2] = ...
    C[2*j+3] = ...
}
GPU hardware introduction
Bird-eye view

CPU

many-core

Memory

channel

channel

core core core core
core core core core
core core core core
core core core core
Why is this different from a CPU?

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

- CPU: minimize latency experienced by 1 thread
  - big on-chip caches
  - sophisticated control logic

- GPU: maximize throughput of all threads
  - # threads in flight limited by resources => lots of resources (registers, etc.)
  - multithreading can hide latency => no big caches
  - share control logic across many threads
Chip area CPU vs GPU

CPU

GPU

Control

ALU

ALU

ALU

ALU

Cache
Key architectural ideas

- Data parallel, like a vector machine
  - There, 1 thread issues parallel vector instructions

- SIMT (Single Instruction Multiple Thread) execution
  - Many threads work on a vector, each on a different element
  - They all execute the same instruction
  - HW automatically handles divergence

- Hardware multithreading
  - HW resource allocation & thread scheduling
  - HW relies on threads to hide latency
  - Context switching is (basically) free
GPU Hardware: NVIDIA
Reader:
- NVIDIA’s Next Generation CUDA Compute Architecture: Fermi

Also for the lab:
- NVIDIA’s CUDA Programmer Guide

Recommended further reading:

Note:
CUDA = Compute Unified Device Architecture
Fermi

- **Consumer:** GTX 480, 580
- **GPGPU:** Tesla C2050
  - More memory, ECC
  - 1.0 teraflop single
  - 515 megaflop double
- 16 streaming multiprocessors (SM)
  - GTX 580: 16
  - GTX 480: 15
  - C2050: 14
- SMs are independent
- 768 KB L2 cache
Fermi Streaming Multiprocessor (SM)

- 32 cores per SM
  (512 cores total)
- 64KB configurable
  L1 cache / shared memory
- 32,768 32-bit registers
CUDA Core Architecture

- Decoupled floating point and integer data paths
- Double precision throughput is 50% of single precision
- Integer operations optimized for extended precision
  - 64 bit and wider data element size
- Predication field for all instructions
- Fused-multiply-add
Memory Hierarchy

- Configurable L1 cache per SM
  - 16KB L1 cache / 48KB Shared
  - 48KB L1 cache / 16KB Shared

- Shared 768KB L2 cache
Multiple Memory Scopes

- **Per-thread private memory**
  - Each thread has its own local memory
  - Stacks, other private data
  - Registers

- **Per-SM shared memory**
  - Small memory close to the processor, low latency

- **Device memory**
  - GPU frame buffer
  - Can be accessed by any thread in any SM
Atomic Operations

- Device (global) memory is not coherent!

- Share data between streaming multiprocessors
  - Potential write hazards!

- Atomics are provided to allow consistency for global shared variables!

- Fermi increases atomics performance by 5x to 20x
  - Still, much slower than non-atomic access
ECC (Error-Correcting Code)

- All major internal memories are ECC protected
  - Register file, L1 cache, L2 cache

- DRAM protected by ECC (on Tesla only)

- ECC is a must have for many computing applications
Programming NVIDIA GPUs
NVIDIA GPUs become more generic

- Expand performance sweet spot of the GPU
  - Caching
  - Concurrent kernels
  - Double precision floating point
  - C++

- Full integration in modern software development environment
  - Debugging
  - Profiling

- Bring more users, more applications to the GPU
CUDA

- CUDA: Scalable parallel programming
  - C/C++ extensions
- Provide straightforward mapping onto hardware
  - Good fit to GPU architecture
  - Maps well to multi-core CPUs too
- Scale to 1000s of cores & 100,000s of threads
  - GPU threads are lightweight — create / switch is free
  - GPU needs 1000s of threads for full utilization
Parallel Abstractions in CUDA

- Hierarchy of concurrent threads
- Lightweight synchronization primitives
- Shared memory model for cooperating threads
Hierarchy of concurrent threads

- Parallel kernels composed of many threads
  - All threads execute the same sequential program
  - Called the kernel

- Threads are grouped into thread blocks
  - Threads in the same block can cooperate
  - Threads in different blocks cannot!

- All thread blocks are organized in a Grid

- Threads/blocks have unique IDs
Grids, Thread Blocks and Threads

Grid

Thread Block 0, 0

0,0 0,1 0,2 0,3
1,0 1,1 1,2 2,3
2,0 2,1 2,2 2,3

Thread Block 0, 1

0,0 0,1 0,2 0,3
1,0 1,1 1,2 2,3
2,0 2,1 2,2 2,3

Thread Block 0, 2

0,0 0,1 0,2 0,3
1,0 1,1 1,2 2,3
2,0 2,1 2,2 2,3

Thread Block 1, 0

0,0 0,1 0,2 0,3
1,0 1,1 1,2 2,3
2,0 2,1 2,2 2,3

Thread Block 1, 1

0,0 0,1 0,2 0,3
1,0 1,1 1,2 2,3
2,0 2,1 2,2 2,3

Thread Block 1, 2

0,0 0,1 0,2 0,3
1,0 1,1 1,2 2,3
2,0 2,1 2,2 2,3
CUDA Model of Parallelism

- CUDA virtualizes the physical hardware
  - Devices have
    - Different numbers of SMs
    - Different compute capabilities (Fermi = 2.0)
  - block is a virtualized streaming multiprocessor (threads, shared memory)
  - thread is a virtualized scalar processor (registers, PC, state)

- Scheduled onto physical hardware without pre-emption
  - threads/blocks launch & run to completion
  - blocks should be independent
Memory Spaces in CUDA

- **Host**
- **Grid**
  - **Block (0, 0)**
    - Registers
    - Shared Memory
    - Thread (0, 0)
    - Thread (1, 0)
  - **Block (1, 0)**
    - Registers
    - Shared Memory
    - Thread (0, 0)
    - Thread (1, 0)

- **Device Memory**
- **Constant Memory**
Device Memory

- CPU and GPU have separate memory spaces
  - Data is moved across PCI-e bus
  - Use functions to allocate/set/copy memory on GPU
  - Very similar to corresponding C functions

- Pointers are just addresses
  - Can’t tell from the pointer value whether the address is on CPU or GPU
  - Must exercise care when dereferencing:
    - Dereferencing CPU pointer on GPU will likely crash
    - Same for vice versa
Additional memories

- **Textures**
  - Read-only
  - Data resides in device memory
  - Different read path, includes specialized caches

- **Constant memory**
  - Data resides in device memory
  - Manually managed
  - Small (e.g., 64KB)
  - Use when all threads in a block read the same address
    - Serializes otherwise
Host (CPU) manages device (GPU) memory:

- `cudaMalloc(void **pointer, size_t nbytes)`
- `cudaMemset(void *pointer, int val, size_t count)`
- `cudaFree(void* pointer)`

```c
int n = 1024;
int nbytes = n * sizeof(int);
int* data = 0;
cudaMalloc(&data, nbytes);
cudaMemset(data, 0, nbytes);
cudaFree(data);
```
Data Copies

- `cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);`
  - returns after the copy is complete
  - blocks CPU thread until all bytes have been copied
  - doesn’t start copying until previous CUDA calls complete

- `enum cudaMemcpyKind`
  - `cudaMemcpyHostToDevice`
  - `cudaMemcpyDeviceToHost`
  - `cudaMemcpyDeviceToDevice`

- Non-blocking copies are also available
  - DMA transfers, overlap computation and communication
### CUDA Variable Type Qualifiers

<table>
<thead>
<tr>
<th>Variable declaration</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>int var;</td>
<td>register</td>
<td>thread</td>
<td>thread</td>
</tr>
<tr>
<td>int array_var[10];</td>
<td>local</td>
<td>thread</td>
<td>thread</td>
</tr>
<tr>
<td><strong>shared</strong> int shared_var;</td>
<td>shared</td>
<td>block</td>
<td>block</td>
</tr>
<tr>
<td><strong>device</strong> int global_var;</td>
<td>device</td>
<td>grid</td>
<td>application</td>
</tr>
<tr>
<td><strong>constant</strong> int constant_var;</td>
<td>constant</td>
<td>grid</td>
<td>application</td>
</tr>
</tbody>
</table>
Philosophy: provide minimal set of extensions necessary

Function qualifiers:
```c
__global__ void my_kernel() { }
__device__ float my_device_func() { }
```

Execution configuration:
```c
dim3 gridDim(100, 50);  // 5000 thread blocks
dim3 blockDim(4, 8, 8);  // 256 threads per block (1.3M total)
my_kernel <<< gridDim, blockDim >>> (...);  // Launch kernel
```

Built-in variables and functions valid in device code:
```c
dim3 gridDim;  // Grid dimension
dim3 blockDim;  // Block dimension
dim3 blockIdx;  // Block index
dim3 threadIdx;  // Thread index

void syncthreads();  // Thread synchronization
```
Calculating the global thread index

Grid

- **blockDim.X**

- "global" thread index:

  \[ \text{blockDim.x} \times \text{blockIdx.x} + \text{threadIdx.x}; \]
Calculating the global thread index

“global” thread index:

\[ \text{blockDim} \cdot x \times \text{blockIdx} \cdot x + \text{threadIdx} \cdot x; \]

\[ 4 \times 2 + 1 = 9 \]
void vector_add(int size, float* a, float* b, float* c) {
    for(int i=0; i<size; i++) {
        c[i] = a[i] + b[i];
    }
}
// compute vector sum c = a + b
// each thread performs one pair-wise addition
__global__ void vector_add(float* A, float* B, float* C) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

int main() {
    // initialization code here ...

    // launch N/256 blocks of 256 threads each
    vector_add<<< N/256, 256 >>>(deviceA, deviceB, deviceC);

    // cleanup code here ...
}

// can be in the same file
Vector addition host code

```c
int main(int argc, char** argv) {
    int size = N * sizeof(float);

    // allocate host memory
    hostA = malloc(size);
    hostB = malloc(size);
    hostC = malloc(size);

    // initialize A, B arrays here...

    // allocate device memory
    cudaMalloc(&deviceA, size);
    cudaMalloc(&deviceB, size);
    cudaMalloc(&deviceC, size);
}
```
/transfer the data from the host to the device
cudaMemcpy(deviceA, hostA, size, cudaMemcpyHostToDevice);
cudaMemcpy(deviceB, hostB, size, cudaMemcpyHostToDevice);

// launch N/256 blocks of 256 threads each
vector_add<<<N/256, 256>>>(deviceA, deviceB, deviceC);

// transfer the result back from the GPU to the host
cudaMemcpy(hostC, deviceC, size, cudaMemcpyDeviceToHost);
CUDA: Scheduling, Synchronization and Atomics
Thread Scheduling

- Order in which thread blocks are scheduled is undefined!
  - any possible interleaving of blocks should be valid
  - presumed to run to completion without preemption
  - can run in any order
  - can run concurrently OR sequentially

- Order of threads within a block is also undefined!
Blocks are scheduled on one SM
- They are grouped in warps

SM’s implement zero-overhead warp scheduling
- A warp contains 32 threads that run concurrently on an SM
  - 32 because of the number of physical cores per SM
- All threads in a warp execute the same instruction when selected: **lock-step execution**
- At any time, only one warp is executed by an SM

Instruction:

```
1 2 3 4 5 6 1 2 1 2 3 4 7 8 1 2 1 2 3 4
```

TB = Thread Block, W = Warp
Global synchronization

Q: How do we do global synchronization with these scheduling semantics?

- A1: Not possible!
- A2: Finish a grid, and start a new one:

```c
step1<<<grid1,blk1>>>(...);
// CUDA ensures that all writes from step1 are complete.
step2<<<grid2,blk2>>>(...);
```

- No need to copy the data back and forth between these invocations!
Atomics

- Guarantee that only a single thread has access to a piece of memory during an operation
- No dropped data, but ordering is still arbitrary
- Different types of atomic instructions
  - Atomic Add, Sub, Exch, Min, Max, Inc, Dec, CAS, And, Or, Xor
- Can be done on device memory and shared memory
- Much more expensive than load + operation + store
Example: Histogram

// Determine frequency of colors in a picture.
// Colors have already been converted into integers
// between 0 and 255.
// Each thread looks at one pixel,
// and increments a counter

__global__ void histogram(int* colors, int* buckets)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int c = colors[i];
    buckets[c] += 1;
}
Example: Histogram

// Determine frequency of colors in a picture.
// Colors have already been converted into integers
// between 0 and 255.
// Each thread looks at one pixel,
// and increments a counter

__global__ void histogram(int* colors, int* buckets)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int c = colors[i];
    buckets[c] += 1;
}
Example: Histogram

// Determine frequency of colors in a picture.
// Colors have already been converted into integers
// between 0 and 255.
// Each thread looks at one pixel,
// and increments a counter atomically

__global__ void histogram(int* colors, int* buckets)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int c = colors[i];
    atomicAdd(&buckets[c], 1);
}
CUDA: optimizing your application

Global memory coalescing
Coalescing

traditional multi-core optimal memory access pattern

<table>
<thead>
<tr>
<th>thread 0</th>
<th>thread 1</th>
<th>thread 2</th>
<th>thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="" alt="Diagram" /></td>
<td><img src="" alt="Diagram" /></td>
<td><img src="" alt="Diagram" /></td>
<td><img src="" alt="Diagram" /></td>
</tr>
</tbody>
</table>

many-core GPU optimal memory access pattern

<table>
<thead>
<tr>
<th>thread 0</th>
<th>thread 1</th>
<th>thread 2</th>
<th>thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="" alt="Diagram" /></td>
<td><img src="" alt="Diagram" /></td>
<td><img src="" alt="Diagram" /></td>
<td><img src="" alt="Diagram" /></td>
</tr>
</tbody>
</table>
Consider the stride of your accesses

```c
__global__ void foo(int* input, float3* input2) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    // Stride 1, full bandwidth used!
    int a = input[i];

    // Stride 2, 50% of the bandwidth is wasted
    int b = input[2*i];

    // Stride 3, 67% of the bandwidth is wasted
    float c = input2[i].x;
}
```
Example: Array of Structures (AoS)

Struct record {
    int key;
    int value;
    int flag;
};

record *d_AoS_data;
cudaMalloc((void**) &d_AoS_data, ...);

kernel {
    threadID = blockDim.x * blockIdx.x + threadIdx.x;
    ...
    d_AoS_data[threadID].value += i; // wastes bandwidth!
    ...
}
Example: Structure of Arrays (SoA)

```c
Struct SoA {
    int* keys;
    int* values;
    int* flags;
};

SoA d_SoA_data;
cudaMalloc((void**) &d_SoA_data.keys, ...);
cudaMalloc((void**) &d_SoA_data.values, ...);
cudaMalloc((void**) &d_SoA_data.flags, ...);

kernel {
    threadID = blockDim.x * blockIdx.x + threadIdx.x;
    ...
    d_SoA_data.values[threadID] += i; // full bandwidth!
    ...
}
```
Memory Coalescing

- A coalesced memory transaction is one in which all of the threads in a half-warp access global memory at the same time. This is oversimple, but the correct way to do it is just have consecutive threads access consecutive memory addresses.

- Stride 1 access patterns are preferred!
  - Other patterns can still get benefits

- Unpredictable/irregular access patterns
  - Case-by-case performance impact

- Structure of arrays is often better than array of structures

- No coalescing => performance loss 10 – 30x!
CUDA: optimizing your application

Shared Memory
Using shared memory

- Equivalent with providing software caching
  - Explicit: Load data to be re-used in shared memory
  - Use it for computation
  - Explicit: Store results back to global memory

- All threads in a block share memory
  - Load/Store: using all threads
  - Barrier: `__syncthreads`
    - Guard against using uninitialized data – not all threads have finished loading data to shared memory
    - Guard against corrupting live data – not all threads have finished computing
A Common Programming Strategy

- Partition data into subsets that fit into shared memory
- Handle each data subset with one thread block
Load the subset from device memory to shared memory, using multiple threads to exploit memory-level parallelism.
A Common Programming Strategy

- Perform the computation on the subset from shared memory
A Common Programming Strategy

- Copy the result from shared memory back to device memory
Matrix multiplication example

- \( C = A \times B \)
- Each element \( C_{i,j} \)
  
  \[ C_{i,j} = \text{dot}(\text{row}(A,i),\text{col}(B,j)) \]

- Parallelization strategy
  - Each thread computes element in \( C \)
  - 2D kernel
Matrix multiplication implementation

```c
__global__ void mat_mul(float *a, float *b,
                        float *c, int width)
{
    // calc row & column index of output element
    int row = blockIdx.y*blockDim.y + threadIdx.y;
    int col = blockIdx.x*blockDim.x + threadIdx.x;

    float result = 0;

    // do dot product between row of a and column of b
    for(int k = 0; k < width; k++) {
        result += a[row*width+k] * b[k*width+col];
    }
    c[row*width+col] = result;
}
```
Matrix multiplication performance

<table>
<thead>
<tr>
<th>Loads per dot product term</th>
<th>2 (a and b) = 8 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLOPS</td>
<td>2 (multiply and add)</td>
</tr>
<tr>
<td>AI</td>
<td>2 / 8 = 0.25</td>
</tr>
<tr>
<td>Performance GTX 580</td>
<td>1581 GFLOPs</td>
</tr>
<tr>
<td>Memory bandwidth GTX 580</td>
<td>192 GB/s</td>
</tr>
<tr>
<td>Attainable performance</td>
<td>192 * 0.25 = 48 GFLOPS</td>
</tr>
<tr>
<td>Maximum efficiency</td>
<td>3.0 % of theoretical peak</td>
</tr>
</tbody>
</table>
- Each input element in A and B is read WIDTH times.
- Load elements into shared memory.
- Have several threads use local version to improve memory bandwidth.
Partition kernel loop into phases

In each thread block, load a tile of both matrices into shared memory each phase

Each phase, each thread computes a partial result
Matrix multiply with shared memory

```c
__global__ void mat_mul(float *a, float *b, float *c, int width) {

    // shorthand
    int tx = threadIdx.x, ty = threadIdx.y;
    int bx = blockIdx.x, by = blockIdx.y;

    // allocate tiles in shared memory
    __shared__ float s_a[TILE_WIDTH][TILE_WIDTH];
    __shared__ float s_b[TILE_WIDTH][TILE_WIDTH];

    // calculate the row & column index from A,B
    int row = by*blockDim.y + ty;
    int col = bx*blockDim.x + tx;

    float result = 0;
```
Matrix multiply with shared memory

// loop over input tiles in phases, p = crt. phase
for(int p = 0; p < width/TILE_WIDTH; p++) {
    // collaboratively load tiles into shared memory
    s_a[ty][tx] = a[row*width + (p*TILE_WIDTH + tx)];
    s_b[ty][tx] = b[(p*TILE_WIDTH + ty)*width + col];
    // barrier: ALL writes to shared memory are finished
    __syncthreads();

    // dot product between row of s_a and col of s_b
    for(int k = 0; k < TILE_WIDTH; k++) {
        result += s_a[ty][k] * s_b[k][tx];
    }
    // barrier: ALL writes to shared memory are finished
    __syncthreads();
}

c[row*width+col] = result;
Use of Barriers in mat_mul

- Two barriers per phase:
  - `__syncthreads` after all data is loaded into shared memory
  - `__syncthreads` after all data is read from shared memory
    - Second `__syncthreads` in phase p guards the load in phase p+1

- Formally, `__syncthreads` is a barrier for shared memory for a block of threads:
  
  "void __syncthreads();

  waits until all threads in the thread block have reached this point and all global and shared memory accesses made by these threads prior to `__syncthreads()` are visible to all threads in the block."
# Matrix multiplication performance

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>shared memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global loads</td>
<td>$2N^3 \times 4\text{ bytes}$</td>
<td>$(2N^3 / \text{TILE}_\text{WIDTH}) \times 4\text{ bytes}$</td>
</tr>
<tr>
<td>Total ops</td>
<td>$2N^3$</td>
<td>$2N^3$</td>
</tr>
<tr>
<td>AI</td>
<td>0.25</td>
<td>$0.25 \times \text{TILE}_\text{WIDTH}$</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Performance GTX 580</strong></td>
<td>1581 GFLOPs</td>
</tr>
<tr>
<td><strong>Memory bandwidth GTX 580</strong></td>
<td>192 GB/s</td>
</tr>
<tr>
<td><strong>AI needed for peak</strong></td>
<td>$1581 / 192 = 8.23$</td>
</tr>
<tr>
<td><strong>TILE_WIDTH required to achieve peak</strong></td>
<td>$0.25 \times \text{TILE_WIDTH} = 8.23$, \text{TILE_WIDTH} = 32.9</td>
</tr>
</tbody>
</table>
Summary and Conclusions
Higher performance cannot be reached by increasing clock frequencies anymore

Solution: introduction of large-scale parallelism

Multiple cores on a chip

Today:
- Up to 48 CPU cores in a node
- Up to 3200 cores on a single GPU
- Host system can contain multiple GPUs: 10,000+ cores
- We can build clusters of these nodes!

Future: 100,000s – millions of cores?
Many different types of many-core hardware

Very different properties
- Performance
- Programmability
- Portability

It's all about the memory

Choose the right platform for your application
- Arithmetic intensity [/ Operational intensity ]
- Roofline model
Summary and conclusions

- Many different many-core programming models
- Most models are hardware-induced, low-level
  - Vectorization
  - Coalescing
  - Explicit software caching (shared memory on GPU)
- Future
  - Cuda? OpenCL?
  - high-level models on top of OpenCL?
- Many-cores are here to stay
Backup slides
CPU vs GPU Chip

AMD Magny-Cours (6 cores)  
- 2 billion transistors  
- 346 mm²

ATI 4870 (800 cores)  
- 1 billion transistors  
- 256 mm²
ATI GPUs
Latest generation ATI

- Southern Islands
  - 1 chip: HD 7970
    - 2048 cores
    - 264 GB/sec memory bandwidth
    - 3.8 tflops single, 947 gflops double precision
    - Maximum power: 250 Watts
    - 399 euros!
  - 2 chips: HD 7990
    - 4096 cores, 7.6 tflops
- Comparison: entire 72-node DAS-4 VU cluster has 4.4 tflops
ATI 5870 architecture overview
Each of the 20 SIMD engines has:
- 16 thread processors x 5 stream cores = 80 scalar stream processing units
- 20 * 16 * 5 = 1600 cores total
- 32KB Local Data Share
- its own control logic and runs from a shared set of threads
- a dedicated fetch unit with 8KB L1 cache
- a 64KB global data share to communicate with other SIMD engines
Each thread processor includes:
- 4 stream cores + 1 special function stream core
- general purpose registers
- FMA in a single clock
EDC (Error Detection Code)

- CRC Checks on Data Transfers for Improved Reliability at High Clock Speeds

Bandwidths

- Up to 1 TB/sec L1 texture fetch bandwidth
- Up to 435 GB/sec between L1 & L2
- 153.6 GB/s to device memory
- PCI-e 2.0, 16x: 8GB/s to main memory
Unified Load/Store Addressing

Non-unified Address Space

- Local
  - \*p\_local
- Shared
  - \*p\_shared
- Device
  - 32-bit

Unified Address Space

- Local
- Shared
- Device
  - 40-bit
Latest generation ATI

- Southern Islands
  - 1 chip: HD 7970
    - 2048 cores
    - 264 GB/sec memory bandwidth
    - 3.8 tflops single, 947 gflops double precision
    - Maximum power: 250 Watts
  - 2 chips: HD 7990
    - 4096 cores, 7.6 tflops
- Comparison: entire 72-node DAS-4 VU cluster has 4.4 tflops
ATI programming models

- Low-level: CAL (assembly)
- High-level: Brook+
  - Originally developed at Stanford University
  - Streaming language
  - Performance is not great
- Now: OpenCL
CUDA: optimizing your application

Optimizing Occupancy
SM implements zero-overhead warp scheduling

A warp is a group of 32 threads that runs concurrently on an SM

At any time, only one of the warps is executed by an SM

Warps whose next instruction has its inputs ready for consumption are eligible for execution

Eligible Warps are selected for execution on a prioritized scheduling policy

All threads in a warp execute the same instruction when selected

TB = Thread Block, W = Warp
Stalling warps

- What happens if all warps are stalled?
  - No instruction issued → performance lost

- Most common reason for stalling?
  - Waiting on global memory

- If your code reads global memory every couple of instructions
  - You should try to maximize occupancy
Occupancy

- What determines occupancy?
- Limited resources!
  - Register usage per thread
  - Shared memory per thread block
Pool of registers and shared memory per SM

- Each thread block grabs registers & shared memory
- If one or the other is fully utilized → no more thread blocks
Resource Limits (2)

- Can only have 8 thread blocks per SM
  - If they’re too small, can’t fill up the SM
  - Need 128 threads / block on gt200 (4 cycles/instruction)
  - Need 192 threads / block on Fermi (6 cycles/instruction)

- Higher occupancy has diminishing returns for hiding latency
Hiding Latency with more threads

Throughput, 32-bit words

GB/s

Threads Per Multiprocessor

0 128 256 384 512 640 768 896 1024
How do you know what you’re using?

- Use "\texttt{nvcc -Xptxas -v}" to get register and shared memory usage.

- Plug those numbers into CUDA Occupancy Calculator.
1) Select Compute Capability (click):

2) Enter your resource usage:
- Threads Per Block
- Registers Per Thread
- Shared Memory Per Block (bytes)

3) GPU Occupancy Data is displayed here and in the graphs:
- Active Threads per Multiprocessor
- Active Wafps per Multiprocessor
- Active Thread Blocks per Multiprocessor
- Occupancy of each Multiprocessor

Physical Limits for GPU Compute Capability:
- Threads per Warp
- Wafps per Multiprocessor
- Threads per Multiprocessor
- Thread Blocks per Multiprocessor
- Total # of 32-bit registers per Multiprocessor
- Register Allocation Unit size
- Register Allocation granularity
- Shared Memory per Multiprocessor (bytes)
- Shared Memory Allocation unit size
- Warp allocation granularity (for register allocation)

Allocation Per Thread Block:
- Warps
- Registers
- Shared Memory

Maximum Thread Blocks Per Multiprocessor:
- Limited by Max Warps / Blocks per Multiprocessor
- Limited by Registers per Multiprocessor
- Limited by Shared Memory per Multiprocessor

Thread Block Limit Per Multiprocessor highlighted

CUDA Occupancy Calculator
Version: 2.0

The other data points represent the range of possible block sizes, register counts, and shared memory allocation.
CUDA: optimizing your application

Shared memory bank conflicts
Shared Memory Banks

- Shared memory is banked
  - Only matters for threads within a warp
  - Full performance with some restrictions
    - Threads can each access different banks
    - Or can all access the same value

- Consecutive words are in different banks

- If two or more threads access the same bank but different value, we get bank conflicts
Bank Addressing Examples: OK

- No Bank Conflicts

- No Bank Conflicts
Bank Addressing Examples: BAD

- **2-way Bank Conflicts**
  - Thread 0
  - Thread 1
  - Thread 2
  - Thread 3
  - Thread 4
  - Thread 8
  - Thread 9
  - Thread 10
  - Thread 11
  - Bank 0
  - Bank 1
  - Bank 2
  - Bank 3
  - Bank 4
  - Bank 5
  - Bank 6
  - Bank 7

- **8-way Bank Conflicts**
  - Thread 0
  - Thread 1
  - Thread 2
  - Thread 3
  - Thread 4
  - Thread 5
  - Thread 6
  - Thread 7
  - Thread 15
  - Bank 0
  - Bank 1
  - Bank 2
  - Bank 3
  - Bank 4
  - Bank 5
  - Bank 6
  - Bank 7
  - Bank 8
  - Bank 9
  - Bank 15

- 8-way conflict diagram with additional banks under the 8-way conflicts section.
Trick to Assess Performance Impact

- Change all shared memory reads to the same value
- All broadcasts = no conflicts
- Will show how much performance could be improved by eliminating bank conflicts

- The same doesn’t work for shared memory writes
  - So, replace shared memory array indices with `threadIdx.x`
  - (Could also be done for the reads)
Programming in CUDA
// For algorithms where the amount of work per item
// is highly non-uniform, it often makes sense to
// continuously grab work from a queue.

__global__
void workq(int* work_q, int* q_counter,
            int queue_max, int* output)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int q_index = atomicInc(q_counter, queue_max);
    int result = do_work(work_q[q_index]);
    output[q_index] = result;
}
// Adjacent Difference application:
// compute result[i] = input[i] - input[i-1]

__global__ void adj_diff_naive(int *result, int *input) {
    // compute this thread's global index
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

    if(i > 0) {
        // each thread loads two elements from device memory
        int x_i = input[i];
        int x_i_minus_one = input[i-1];

        result[i] = x_i - x_i_minus_one;
    }
}
// Adjacent Difference application:
// compute result[i] = input[i] – input[i-1]

__global__ void adj_diff_naive(int *result, int *input) {
    // compute this thread’s global index
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

    if(i > 0) {
        // each thread loads two elements from device memory
        int x_i = input[i];
        int x_i_minus_one = input[i-1];

        result[i] = x_i - x_i_minus_one;
    }
}

How do we use device memory bandwidth?
// Adjacent Difference application:
// compute result[i] = input[i] - input[i-1]

__global__ void adj_diff_naive(int *result, int *input) {
    // compute this thread's global index
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

    if(i > 0) {
        // each thread loads two elements from device memory
        int x_i = input[i];
        int x_i_minus_one = input[i-1];

        result[i] = x_i - x_i_minus_one;
    }
}

The next thread also reads input[i]
__global__ void adj_diff(int *result, int *input) {
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

    __shared__ int s_data[BLOCK_SIZE];  // shared, 1 elt / thread
    // each thread reads 1 device memory elt, stores it in s_data
    s_data[threadIdx.x] = input[i];

    // avoid race condition: ensure all loads are complete
    __syncthreads();

    if(threadIdx.x > 0) {
        result[i] = s_data[threadIdx.x] - s_data[threadIdx.x-1];
    } else if(i > 0) {
        // I am thread 0 in this block: handle thread block boundary
        result[i] = s_data[threadIdx.x] - input[i-1];
    }
}
Using shared memory: coalescing

```c
__global__ void adj_diff(int *result, int *input) {
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

    __shared__ int s_data[BLOCK_SIZE]; // shared, 1 elt / thread
    // each thread reads 1 device memory elt, stores it in s_data
    s_data[threadIdx.x] = input[i];    // COALESCED ACCESS!

    // avoid race condition: ensure all loads are complete
    __syncthreads();

    if(threadIdx.x > 0) {
        result[i] = s_data[threadIdx.x] - s_data[threadIdx.x-1];
    } else if(i > 0) {
        // I am thread 0 in this block: handle thread block boundary
        result[i] = s_data[threadIdx.x] - input[i-1];
    }
}
```
In search of portability

OpenCL
Portability

- Inter-family vs inter-vendor
  - NVIDIA Cuda runs on all NVIDIA GPU families
  - OpenCL runs on all GPUs, Cell, CPUs

- Parallelism portability
  - Different architecture requires different granularity
  - Task vs data parallel

- Performance portability
  - Can we express platform-specific optimizations?
The Khronos group

Over 100 companies creating visual computing standards

Board of Promoters
OpenCL: Open Compute Language

- Architecture independent
- Explicit support for many-cores
- Low-level host API
  - Uses C library, no language extensions
- Separate high-level kernel language
  - Explicit support for vectorization
- Run-time compilation
- Architecture-dependent optimizations
  - Still needed
  - Possible
## Cuda vs OpenCL Terminology

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
<td>Work item</td>
</tr>
<tr>
<td>Thread block</td>
<td>Work group</td>
</tr>
<tr>
<td>Device memory</td>
<td>Global memory</td>
</tr>
<tr>
<td>Constant memory</td>
<td>Constant memory</td>
</tr>
<tr>
<td>Shared memory</td>
<td>Local memory</td>
</tr>
<tr>
<td>Local memory</td>
<td>Private memory</td>
</tr>
</tbody>
</table>
## Cuda vs OpenCL Qualifiers

### Functions

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>global</strong></td>
<td>__kernel</td>
</tr>
<tr>
<td><strong>device</strong></td>
<td>(no qualifier needed)</td>
</tr>
</tbody>
</table>

### Variables

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>constant</strong></td>
<td><strong>constant</strong></td>
</tr>
<tr>
<td><strong>device</strong></td>
<td><strong>global</strong></td>
</tr>
<tr>
<td><strong>shared</strong></td>
<td><strong>local</strong></td>
</tr>
</tbody>
</table>
## Cuda vs OpenCL Indexing

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>gridDim</td>
<td>get_num_groups()</td>
</tr>
<tr>
<td>blockDim</td>
<td>get_local_size()</td>
</tr>
<tr>
<td>blockIdx</td>
<td>get_group_id()</td>
</tr>
<tr>
<td>threadIdx</td>
<td>get_local_id()</td>
</tr>
<tr>
<td>Calculate manually</td>
<td>get_global_id()</td>
</tr>
<tr>
<td>Calculate manually</td>
<td>get_global_size()</td>
</tr>
</tbody>
</table>

`__syncthreads()` → `barrier()`
Vector add: Cuda vs OpenCL kernel

**CUDA**

```c
__global__ void vectorAdd(float* a, float* b, float* c) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    c[index] = a[index] + b[index];
}
```

**OpenCL**

```c
__kernel void vectorAdd(__global float* a, __global float* b,
                        __global float* c) {
    int index = get_global_id(0);
    c[index] = a[index] + b[index];
}
```
const size_t workGroupSize = 256;
const size_t nrWorkGroups = 3;
const size_t totalSize = nrWorkGroups * workGroupSize;

cl_platform_id platform;
clGetPlatformIDs(1, &platform, NULL);

// create properties list of key/values, 0-terminated.
cl_context_properties props[] = {
    CL_CONTEXT_PLATFORM, (cl_context_properties)platform,
    0
};

cl_context context = clCreateContextFromType(props,
    CL_DEVICE_TYPE_GPU, 0, 0, 0);
cl_device_id device;
clGetDeviceIDs(platform, CL_DEVICE_TYPE_DEFAULT, 1,
   &device, NULL);

// create command queue on 1st device the context reported
cl_command_queue commandQueue =
    clCreateCommandQueue(context, device, 0, 0);

// create & compile program
cl_program program = clCreateProgramWithSource(context, 1,
    &programSource, 0, 0);
clBuildProgram(program, 0, 0, 0, 0, 0, 0, 0);

// create kernel
cl_kernel kernel = clCreateKernel(program, "vectorAdd", 0);
float* A, B, C = new float[totalSize]; // alloc host vecs
// initialize host memory here...

// allocate device memory
cl_mem deviceA = clCreateBuffer(context,
   CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
   totalSize * sizeof(cl_float), A, 0);

cl_mem deviceB = clCreateBuffer(context,
   CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
   totalSize * sizeof(cl_float), B, 0);

cl_mem deviceC = clCreateBuffer(context,
   CL_MEM_WRITE_ONLY, totalSize * sizeof(cl_float), 0, 0);
// setup parameter values

clSetKernelArg(kernel, 0, sizeof(cl_mem), &deviceA);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &deviceB);
clSetKernelArg(kernel, 2, sizeof(cl_mem), &deviceC);

clEnqueueNDRangeKernel(commandQueue, kernel, 1, 0,
                        &totalSize, &workGroupSize, 0, 0, 0); // execute kernel

// copy results from device back to host, blocking

clEnqueueReadBuffer(commandQueue, deviceC, CL_TRUE, 0,
                     totalSize * sizeof(cl_float), C, 0, 0, 0);

delete[] A, B, C; // cleanup

clReleaseMemObject(deviceA);  clReleaseMemObject(deviceB);
clReleaseMemObject(deviceC);