

Lecture 3

Programming with GPUs

Announcements

- GPU access
 - lilliput: Tesla C1060 (4 devices)
 - cseclass0{1,2}: Fermi GTX 570 (1 device each)
- MPI
 - Trestles @ SDSC
 - Kraken @ NICS

Today's lecture

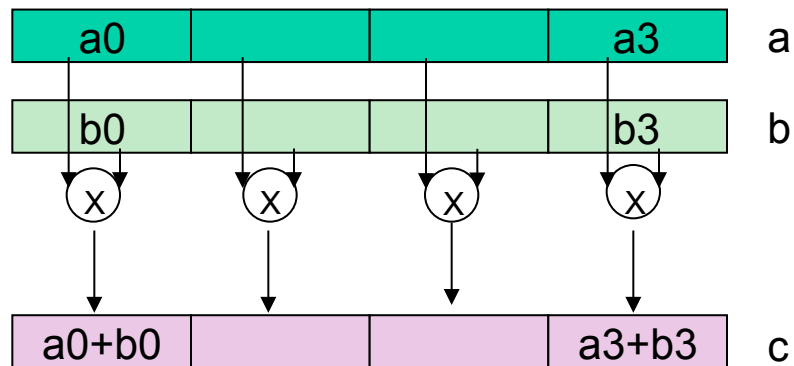
- Vector processing (SIMD)
- Introduction to Stream Processing
- Graphical Processing Units

Vector processing

Streaming SIMD Extensions

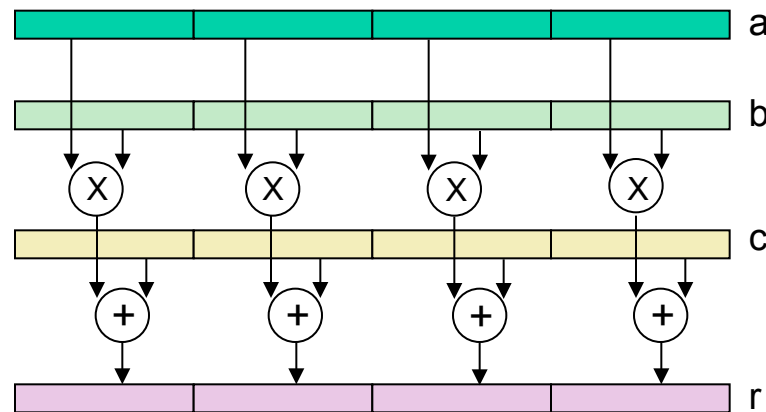
- en.wikipedia.org/wiki/Streaming_SIMD_Extensions
- SSE (SSE4 on Intel Nehalem), AltiVec
- Short vectors: up to 256 bits

for $i = 0:3$ { $c[i] = a[i] * b[i];$ }



More about streaming extensions

- Fused multiply-add
- Memory accesses must be contiguous and aligned
- How do we sum the values in a vector?



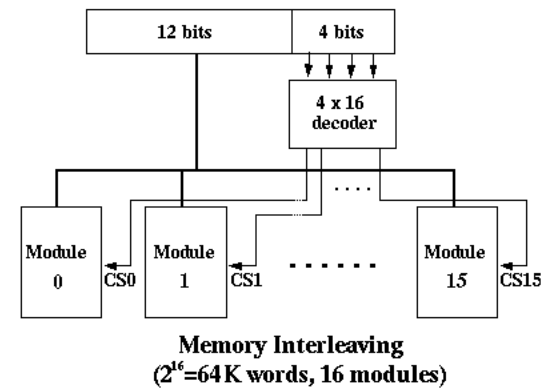
$$\mathbf{r}[0:3] = \mathbf{c}[0:3] + \mathbf{a}[0:3] * \mathbf{b}[0:3]$$

Courtesy of Mercury Computer Systems, Inc.

Memory interleaving

- Compensates for slow memory access times
- Assume we are accessing memory consecutively
- What happens if the stride = number of banks?

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19



Ruye Wang,
fourier.eng.hmc.edu/e85/lectures/memory

How do we use the SSE instructions?

- Low level: assembly language or libraries
- Higher level: a vectorizing compiler

```
/opt/intel/bin/icpc -O2 -vec-report3 src.c
```

```
float a[n], b[n], c[n] = ...;
```

```
for (int i=0; i<n; i++)
```

```
    a[i] = b[i] + c[i];
```

```
t2a.c(19): (col. 3) remark: LOOP WAS VECTORIZED.
```

- Intel Xeon E5504 “Gainestown” @ 2.00GHz (lilliput)
- Double precision
 - With vectorization: 1.48 sec. [0.180 Gflops/s]
 - Without vectorization: 2.96 sec. [0.090 Gflops/s]
- Single precision
 - With vectorization: 0.574 Gflops/s
 - Without vectorization: 0.142 Gflops/s

How does the vectorizer work?

- Original code

```
for (int i=0; i<n; i++)  
    a[i] = b[i] + c[i];
```

- Transformed code

```
for (i = 0; i < 1024; i+=4)  
    a[i:i+3] = b[i:i+3] + c[i:i+3];
```

- Vector instructions

```
for (i = 0; i < 1024; i+=4){  
    vB = vec_ld( &b[i] );  
    vC = vec_ld( &c[i] );  
    vA = vec_add( vB, vC );  
    vec_st( vA, &a[i] );  
}
```

Data dependencies prevent vectorization

- Data dependencies

```
for (int i = 1; i < N; i++)
```

```
    b[i] = b[i-1] + 2;
```

```
b[1] = b[0] + 2;
```

```
b[2] = b[1] + 2;
```

```
b[3] = b[2] + 2;
```

flow.c(6): warning #592: variable "b" is used before its value is set

```
b[i] = b[i-1] + 2; /* data dependence cycle */
```

^

flow.c(5): (col. 1) remark: loop was not vectorized: existence of vector dependence.

flow.c(6): (col. 2) remark: vector dependence: assumed FLOW dependence between b line 6 and b line 6.

But note different output from C++ compiler:

~~flow.c(6): warning #592: variable "b" is used before its value is set~~
~~b[i] = b[i-1] + 2;~~

What prevents vectorization

- Interrupted flow out of the loop

```
for (i=0; i<n; i++) {  
    a[i] = b[i] + c[i];  
    maxval = (a[i] > maxval ? a[i] : maxval);  
    if (maxval > 1000.0) break;  
}
```

t2mx.c(13): (col. 5) remark: loop was not vectorized:
nonstandard loop is not a vectorization candidate.

- This loop will vectorize

```
for (i=0; i<n; i++) {  
    a[i] = b[i] + c[i];  
    maxval = (a[i] > maxval ? a[i] : maxval);  
}
```

Compiler makes conservative decisions

- Interrupted flow out of the loop

```
void copy(char *p, char *q, int n) { // $PUB/Examples/Vectorization
    int i;
    for (i = 0; i < n; i++) p[i] = q[i];
}
```

copy.c(3): (col. 3) remark: loop was not vectorized: not inner loop.

copy.c(3): (col. 3) remark: loop was not vectorized: existence of vector dependence.

copy.c(3): (col. 27) remark: vector dependence: assumed ANTI dependence between q line 3 and p line 3.

copy.c(3): (col. 27) remark: vector dependence: assumed FLOW dependence between p line 3 and q line 3.

copy.c(3): (col. 27) remark: vector dependence: assumed FLOW dependence between p line 3 and q line 3.

copy.c(3): (col. 27) remark: vector dependence: assumed ANTI dependence between q line 3 and p line 3.

Run-time data dependence testing

- Restrict keyword needed to ensure correct semantics
<http://www.devx.com/tips/Tip/13825>
“During the scope of the pointer declaration, all data accessed through it will be accessed .. through any other pointer... [thus] a given object cannot be changed through another pointer.”

`/opt/intel/bin/icpc -O2 -vec-report3 -restrict t2a.c`

```
void copy_conserve(char *restrict p, char *restrict q, int n) {  
    int i;  
    if (p+n < q || q+n < p)  
        #pragma ivdep  
        for (i = 0; i < n; i++) p[i] = q[i];      /* vector loop */  
    else  
        for (i = 0; i < n; i++) p[i] = q[i];      /* serial loop */  
}
```

`copy.c(11): (col. 3) remark: LOOP WAS VECTORIZED.`

Restrictions on vectorization

- Inner loops only

```
for(int j=0; j< reps; j++)  
    for (int i=0; i<N; i++)  
        a[i] = b[i] + c[i];
```

t2av.cpp(95): (col. 7) remark: loop was not vectorized: not inner loop.

Alignment

- Unaligned data movement is expensive
- Accesses aligned on 16 byte boundaries go faster
- Intel compiler can handle some alignments

<http://drdobbs.com/cpp/184401611>

```
foo(double *a, double *b, int N);  
for (int i = 1; i < N-1; i++)  
    a[i+1] = b[i] * 3;
```

```
a[2] = b[1] - 1  
for (int i = 2; i < N-1; i++)  
    a[i+1] = b[i] * 3
```

cannot be vectorized:

```
void fill (char *x){  
    for (int i = 0; i < 1024; i++)  
        x[i] = 1;  
}
```

a[0]	a[1]	a[2]	a[3]
------	------	------	------

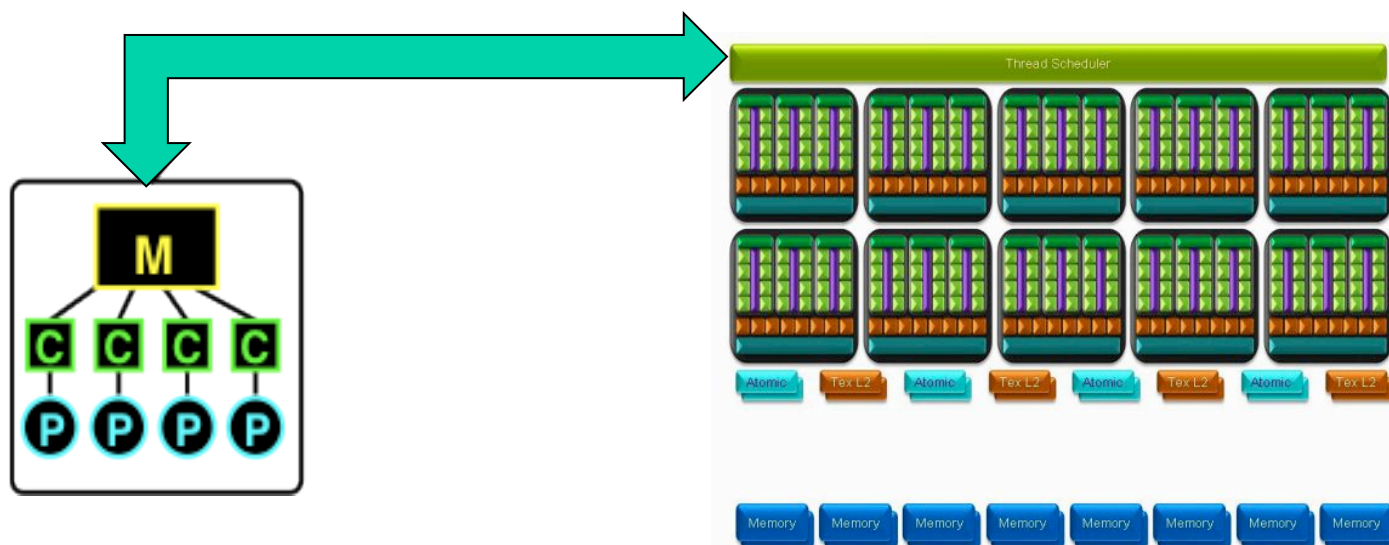
b[0]	b[1]	b[2]	b[3]
------	------	------	------

```
peel = x & 0x0f;  
if (peel != 0) {  
    peel = 16 - peel;  
    for (i = 0; i < peel; i++) x[i] = 1;  
}  
/* aligned access */  
for (i = peel; i < 1024; i++) x[i] = 1;
```

Computing with Graphical Processing Units (GPUs)

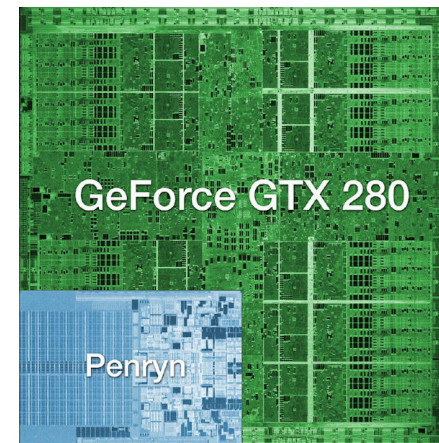
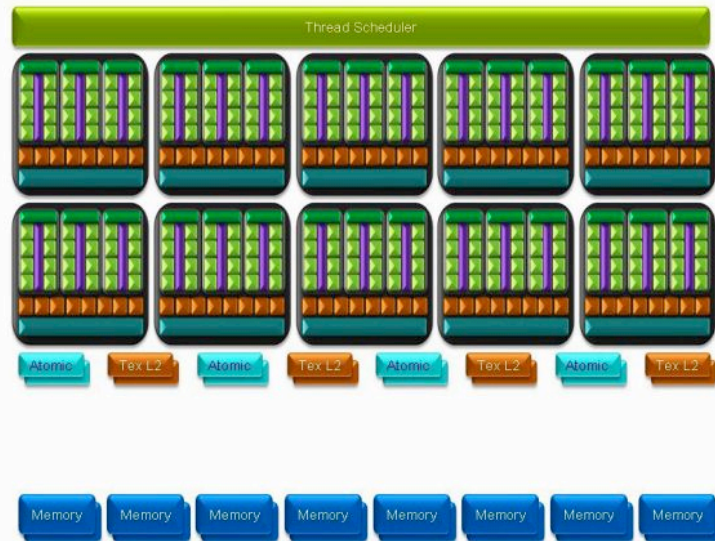
Heterogeneous processing

- Two types of processors: general purpose + accelerator
- Accelerator can perform certain tasks more quickly subject to various overhead costs
- Accelerator amplifies relative cost of communication



NVIDIA GeForce GTX 280

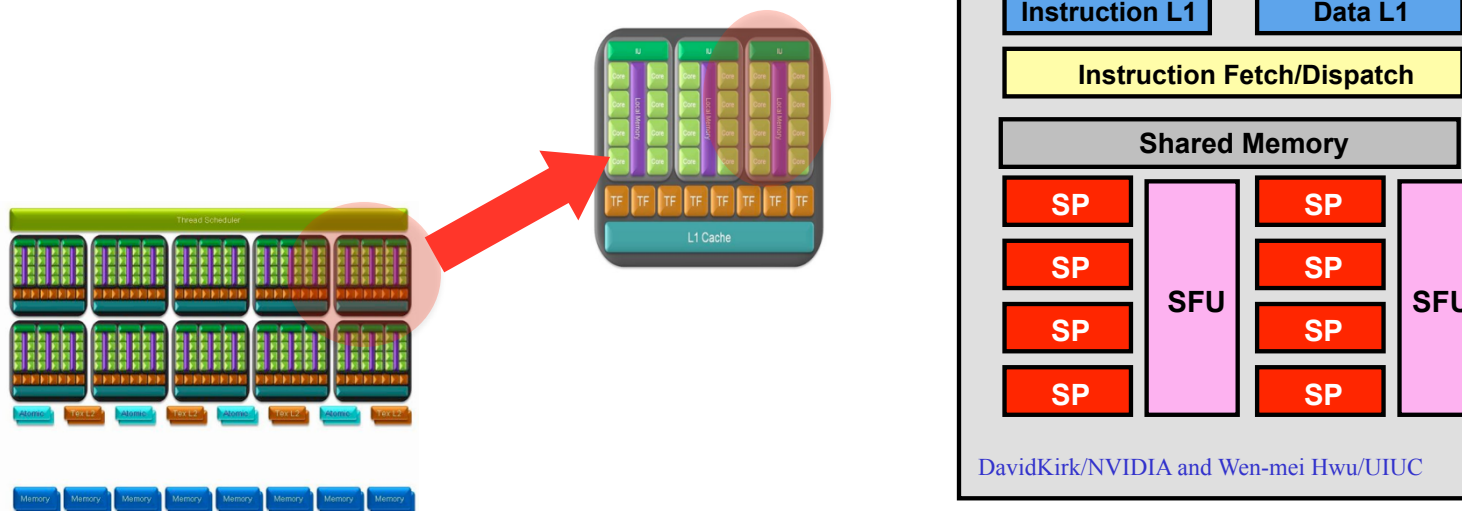
- Hierarchically organized clusters of streaming multiprocessors
 - 240 cores @ 1.296 GHz
 - Peak performance 933.12 Gflops/s
- SIMT parallelism
- 1 GB “device” memory (frame buffer)
- 512 bit memory interface @ 132 GB/s



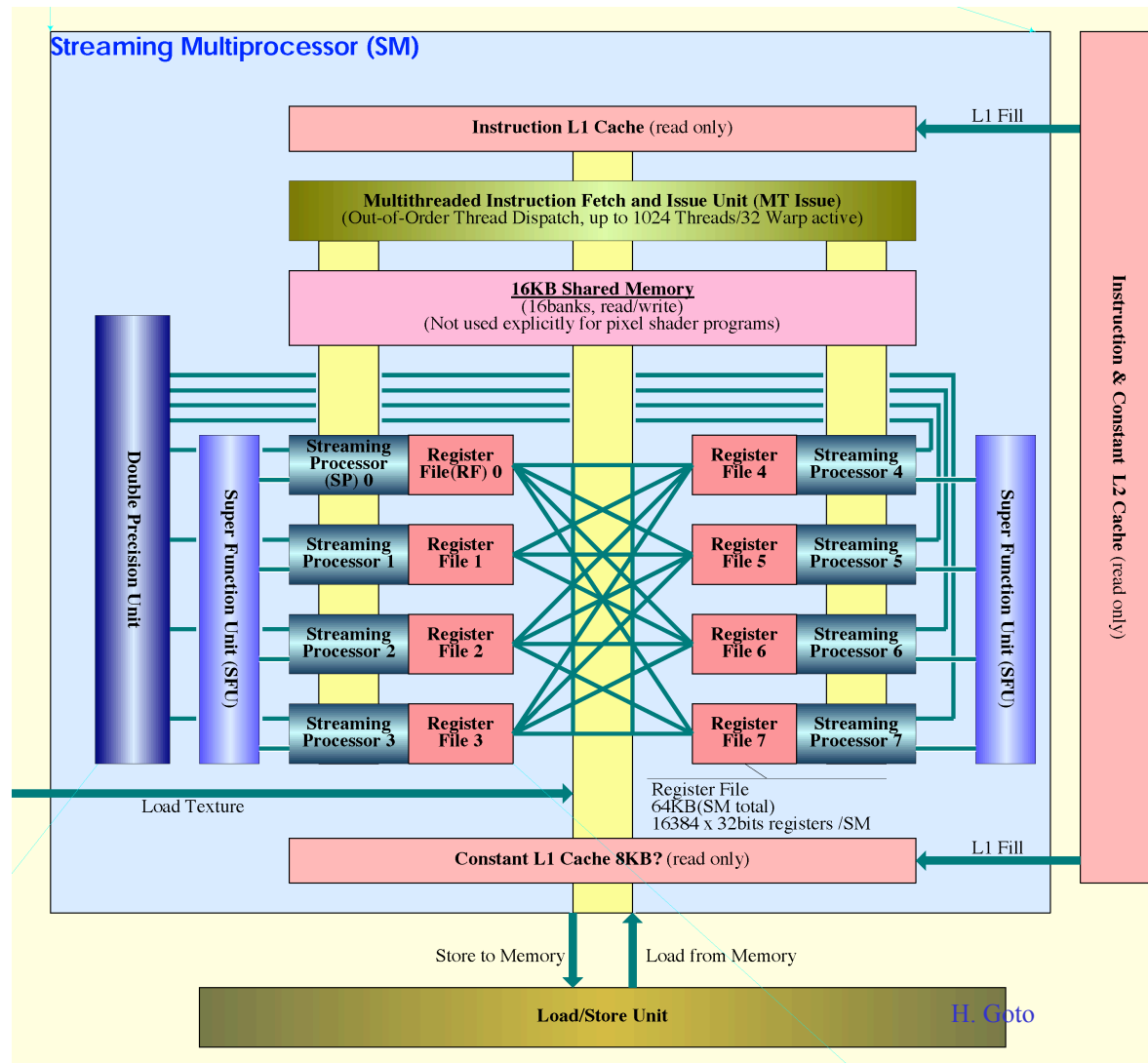
GTX 280: 1.4B transistors
Intel Penryn: 410M
(dual core)

Streaming processor cluster

- GTX-280 GPU
 - 10 clusters @ 3 streaming multiprocessors or *vector cores*
- Each vector core
 - 8 scalar cores: fused multiply adder + multiplier (32 bits), truncate intermediate result
 - Shared memory (16KB) and registers (64 KB)
 - 64-bit fused multiply-adder + 2 super function units (2 fused multiply-adders)
 - 1 FMA + 1 multiply per cycle = 3 flops / cycle / core * 240 cores = 720 flops/cycle
 - @ 1.296 Ghz: 933 GFLOPS

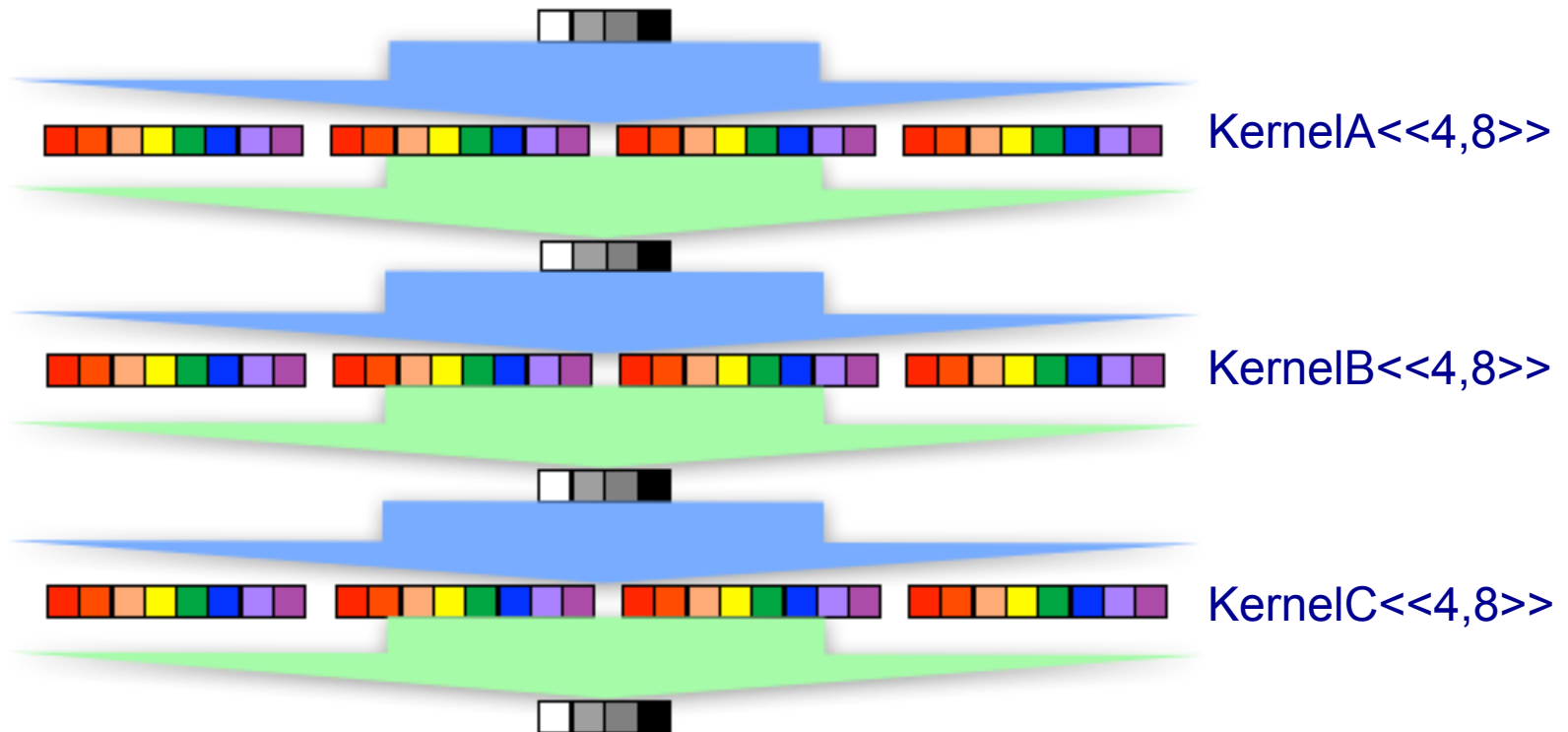


Streaming Multiprocessor



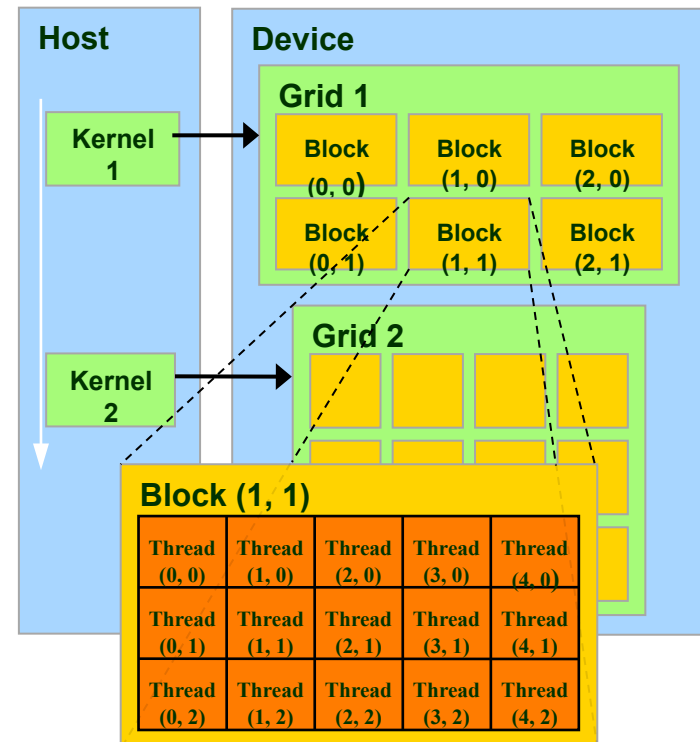
CUDA Acceleration model

- Under control of the *host*, invoke sequences of multithreaded kernels on the *device* (GPU)
- Many lightweight threads
- CUDA: programming environment + C extensions



Thread execution model

- Kernel call spawns virtualized, hierarchically organized threads
 - Grid $\supset$ Block $\supset$ Thread
 - Specify number and geometry of threads in a block and similarly for blocks
- Thread Blocks
 - Subdivide a global index domain
 - Cooperate, synchronize, with access fast on-chip shared memory
 - Threads in different blocks communicate only through slow global memory
- Threads are lightweight
 - Assigned to SM in units of blocks
 - Compiler re-arranges loads to hide latencies
- Global synchronization: kernel invocation

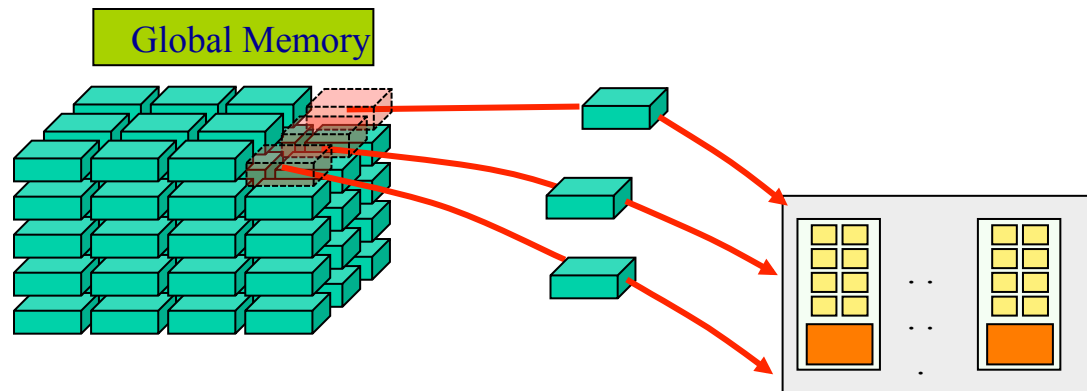


KernelA<<<2,3>,<3,5>>>

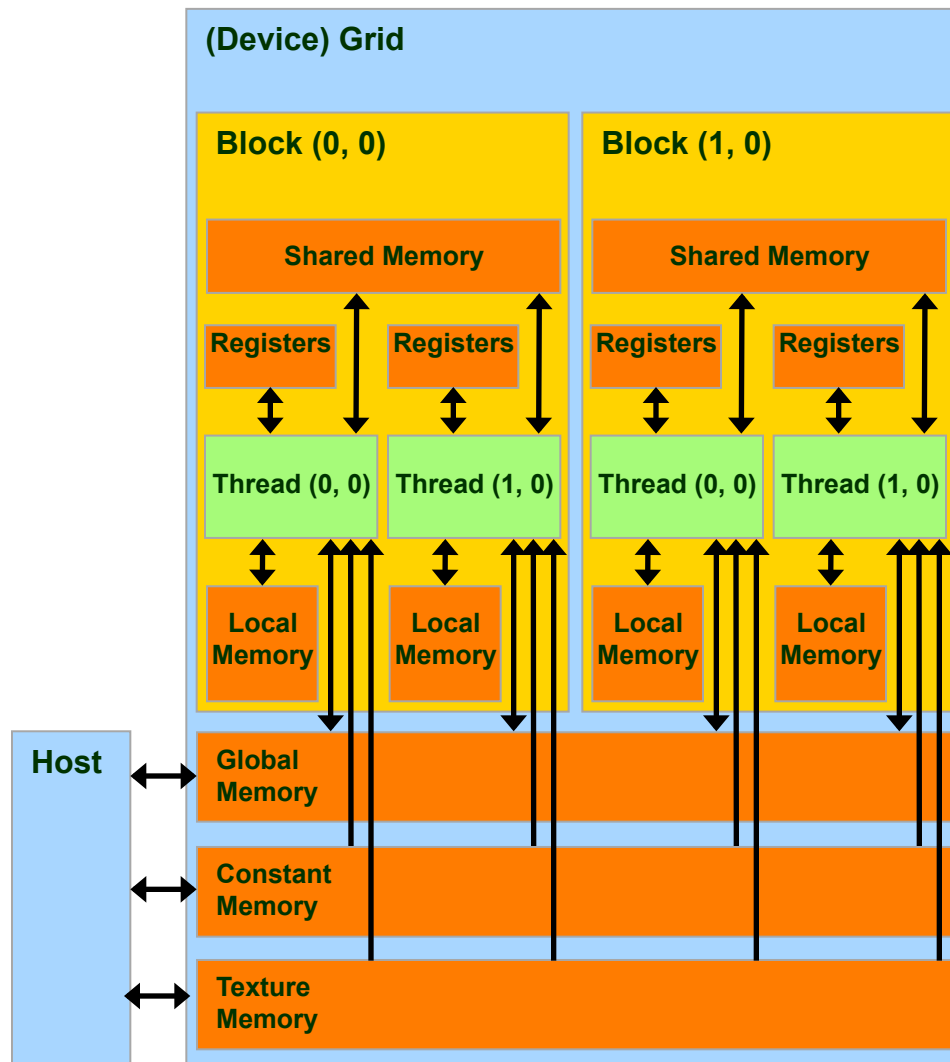
DavidKirk/NVIDIA & Wen-mei Hwu/UIUC

Threads, blocks, grids and data

- All threads execute same instruction (SIMT)
- A block may have a different number of dimensions (1d, 2d or 3d) than a grid (1d/2d)
- Each thread uniquely specified by block & thread ID
- Programmer determines the mapping of virtual thread IDs to global memory locations
 - $\Pi: Z^n \rightarrow Z^2 \times Z^3$
 - $\Theta(\Pi_i), \forall \Pi_i \in \Pi$



Memory Hierarchy



Courtesy DavidKirk/NVIDIA and Wen-mei Hwu/UIUC

Name	Latency (cycles)	Cached
Global	DRAM – 100s	No
Local	DRAM – 100s	No
Constant	1s – 10s – 100s	Yes
Texture	1s – 10s – 100s	Yes
Shared	1	--
Register	1	--

Run time

- `void __syncthreads()`
- Synchronizes **all** threads in a block
- Threads wait until all have arrived
- Avoids memory hazards when accessing shared or global memory
- Beware of conditionals
`if (condition(x)) __syncthreads();`

Coding example – Increment Array

Serial Code

```
void incrementArrayOnHost(float *a, int N){  
    int i;  
    for (i=0; i < N; i++) a[i] = a[i]+1.f;  
}
```

```
#include <cuda.h>  
__global__ void incrementOnDevice(float *a, int N)  
{  
    int idx = blockIdx.x*blockDim.x + threadIdx.x;  
    if (idx<N) a[idx] = a[idx]+1.f;  
}  
  
incrementOnDevice <<< nBlocks, blockSize >>> (a_d, N);
```

Rob Farber, Dr Dobb's Journal

Managing memory

```
float *a_h, *b_h;           // pointers to host memory
float *a_d;                 // pointer to device memory

cudaMalloc((void **) &a_d, size);

for (i=0; i<N; i++) a_h[i] = (float)i; // init host data

cudaMemcpy(a_d, a_h, sizeof(float)*N,
           cudaMemcpyHostToDevice);
```

Computing and returning result

```
int bSize = 4;
int nBlocks = N/bSize + (N%bSize == 0?0:1);
incrementOnDevice <<< nBlocks, bSize >>> (a_d, N);

// Retrieve result from device and store in b_h
cudaMemcpy(b_h, a_d, sizeof(float)*N,
           cudaMemcpyDeviceToHost);

// check results
for (i=0; i<N; i++) assert(a_h[i] == b_h[i]);

// cleanup
free(a_h); free(b_h);
cudaFree(a_d);
```

CUDA language extensions

- Type qualifiers to declare device kernel functions
`__global__ void matrixMul( ...)`
- Kernel launch syntax
`matrixMul<<< grid, threads >>>(...)`
- Keywords
`blockIdx, threadIdx, blockDim, gridDim`
- Runtime, e.g. storage allocation
`cudaMalloc, cudaFree, cudaMemcpy`

Configuration variables

- Types to manage thread geometries
- `dim3 gridDim, blockDim`
 - Dimensions of the grid in blocks
(`gridDim.z` not used)
 - Dimensions of a thread block in threads
- `dim3 blockIdx, threadIdx;`
 - Block index within the grid
 - Thread index within the block

```
__global__ void KernelFunc(...);  
dim3  DimGrid(40, 30);  // 1200 thread blocks  
dim3  DimBlock(4, 8, 16); // 512threads per block  
Kernel<<< DimGrid, DimBlock, >>>(...);
```

For next time

- Run the provided increment array benchmark
 - Report performance on Lilliput for various values of N , # timesteps, and thread block sizes
 - What settings minimized the *running time per point*?
 - Compare performance with the host
- Next, change the increment function to the $\sin()$ function, scaling the input array to cover the range $[0:2\pi)$
 - Why are the results different on the host and device?
 - Comment out the verification code and report performance
 - Add a loop to repeat the computation several times, account for what you observe
- Be prepared to discuss your results in class next time
- Code is found in `lilliput:$PUB/Examples/CUDA/incrArr`
- `PUB = /class/public/cse262-sp11`

Assignment

- Starting from a basic (unoptimized) CUDA implementation
- Optimize matrix multiplication for the GPU
 - Fermi and 200 series
- Next, implement matrix multiplication in MPI
 - Lilliput → Trestles, if time Kraken
- You'll be given an efficient (blocked for cache) serial implementation
- Goals
 - Understand differences between Fermi and 200 series GPUs
 - Comparative performance across platforms: how many MPI cores needed to deliver comparable performance to the GPU?