

Lecture 6

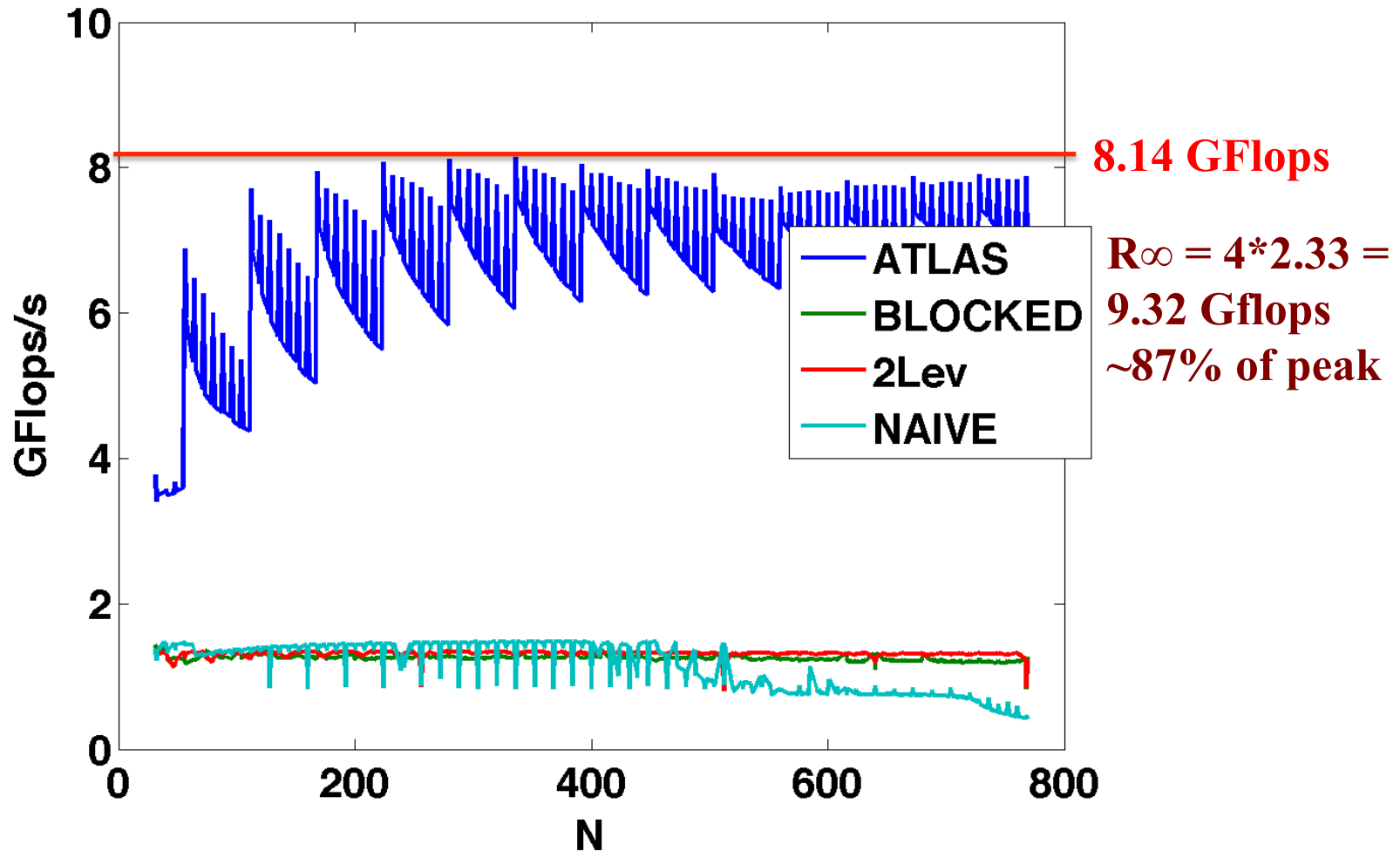
CUDA Programming

Announcements

- A2: GPU programming matrix multiplication
- Get ready for CUDA
 - ♦ Run the incrArray code example on a CSEClass machine and on Dirac
- Projects (A3) discussed next week

Assignment #1

- Blocking for cache will boost performance but a lot more is needed to approach ATLAS' performance

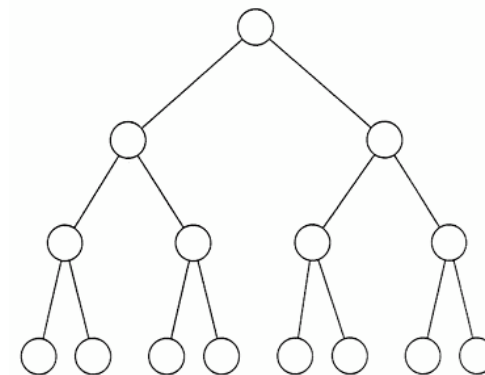
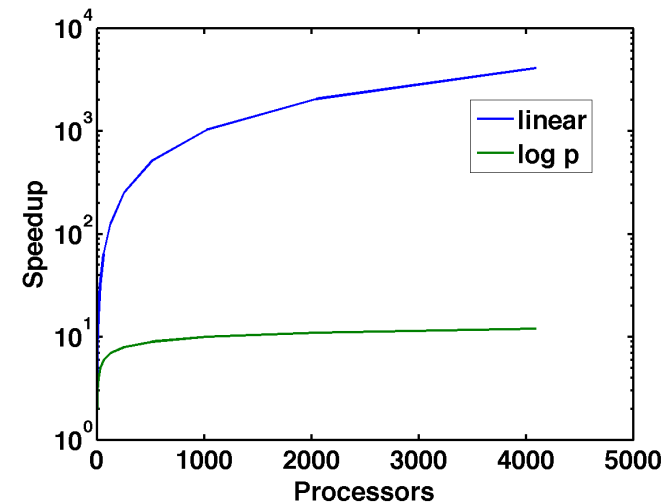


Today's Lecture

- CUDA Programming
- Applications

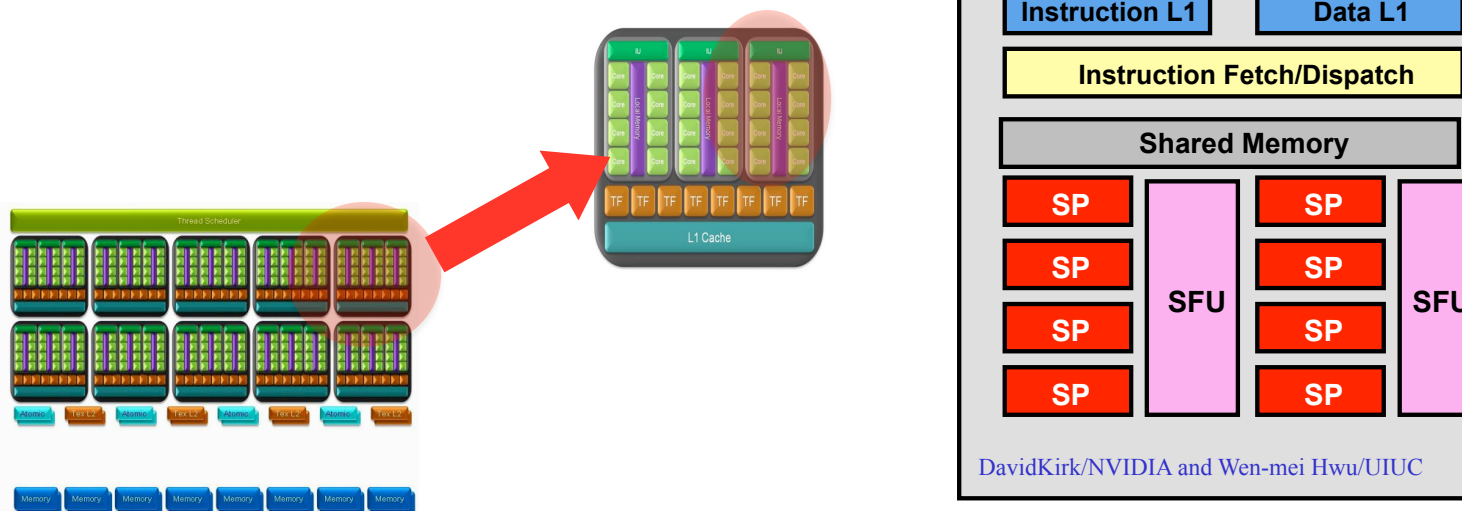
Recapping: Maximize Performance on a GPU

- Avoid algorithms that present intrinsic barriers to utilizing the hardware
 - Avoid costly branches, or render harmless
 - Minimize serial sections
- **Cut data motion costs**
 - ◆ Hide latency of host $\leftrightarrow$ device memory transfers
 - ◆ Reduce global memory accesses $\rightarrow$ fast on-chip accesses
 - ◆ **Coalesced memory transfers**



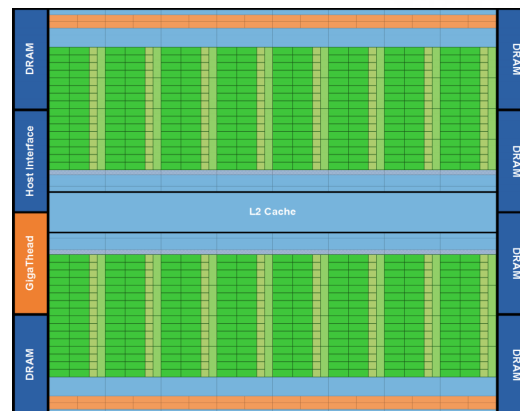
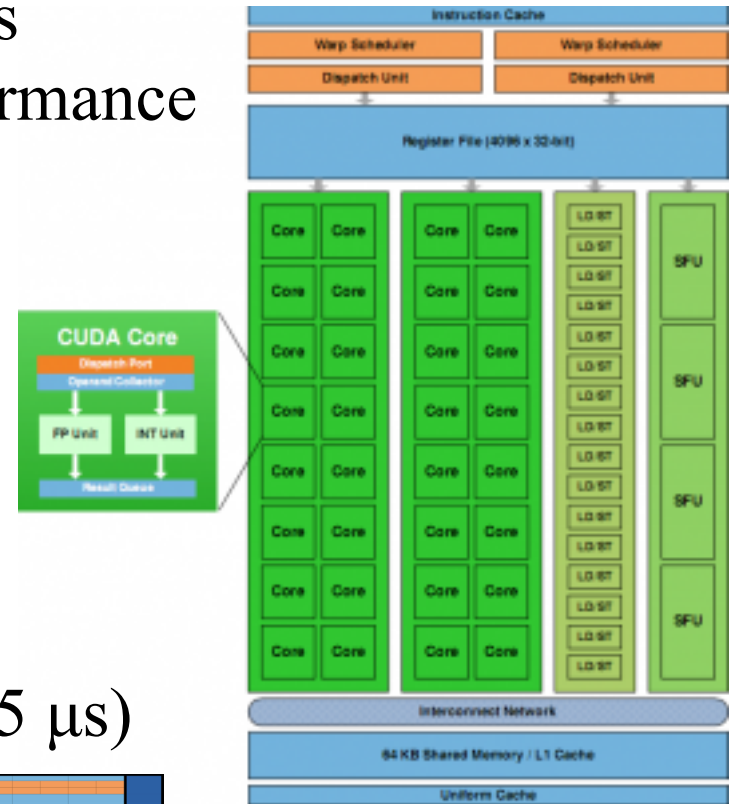
Streaming processor cluster

- 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 (16K × 32 bits = 64KB)
 - ♦ 1 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



Fermi

- Larger vector units, more total cores
- Higher peak double precision performance
- L1 Cache, configurable as $\frac{3}{4}$ L1 or SM, $\frac{1}{4}$ SM or L1 64 KB/vector unit
- Shared L2 Cache (768 KB)
- Dual thread schedulers
- Concurrent kernel execution (some models)
- Reduced kernel launch overhead (25 μ s)
- Improved predication



Vector units

- Each vector unit
 - 32 CUDA cores for integer and floating-point arithmetic
 - 4 special function units for Single Precision transcendentals
 - FMA without truncation (32 or 64 bits)
- For devices of compute capability 2.1
 - 48 CUDA cores for arithmetic operations
 - 8 special function units for single-precision
- *CUDA C Programming Guide*, §G.4

Fermi platforms in the class

CSEClass 01, 02: GeForce GTX 580 [2.0, GF100]

15 Vector units @ 32 cores/unit (480 cores), 4 SFUs

1.25 GB device memory [01 not working currently]

CSEClass 03-07: GeForce GTX 460 [2.1, GF104]

7 Vector units @ 48 cores (384 total cores), 8 SFUs

1.0 GB device memory

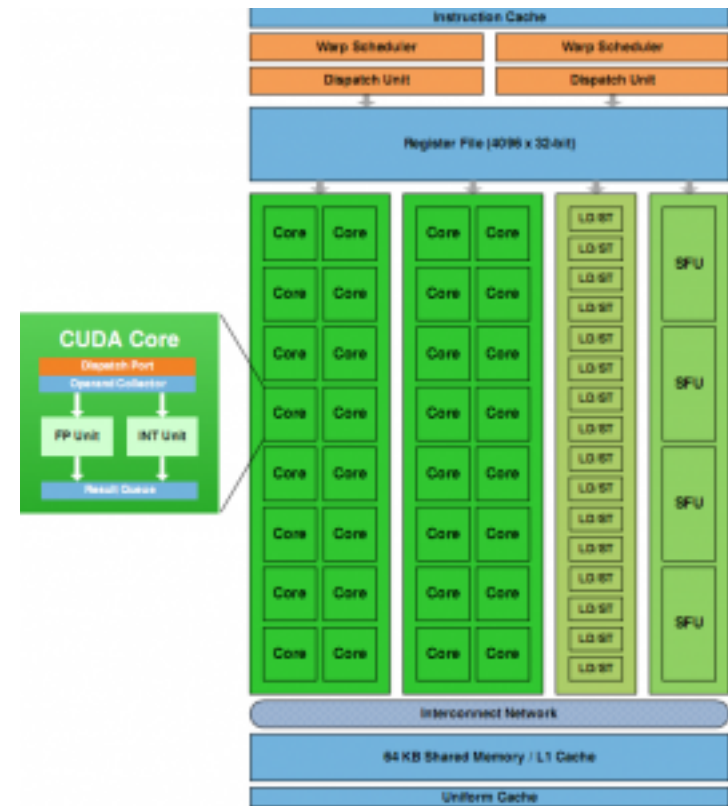
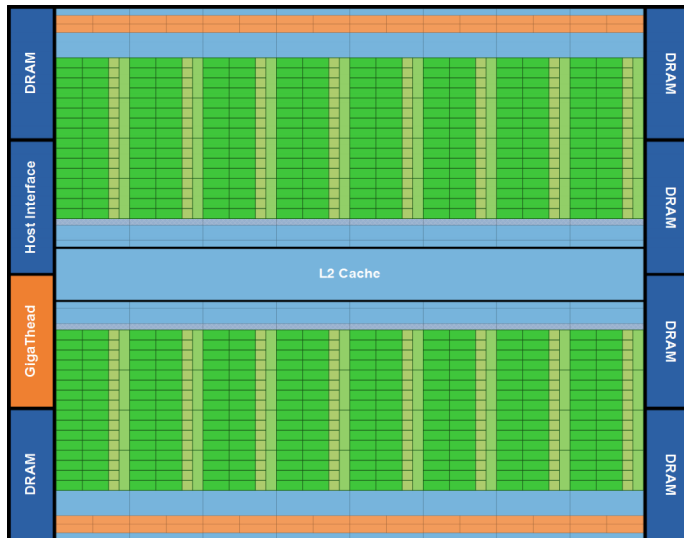
Dirac: Tesla C2050 [2.0, GF100]

1 device per node

14 Vector units @ 32 cores (448 total cores), 4 SFUs

3 GB device memory + ECC (2.625GB usable)

SP MAD: 1030.4 Gflops, DP FMA: 515.2



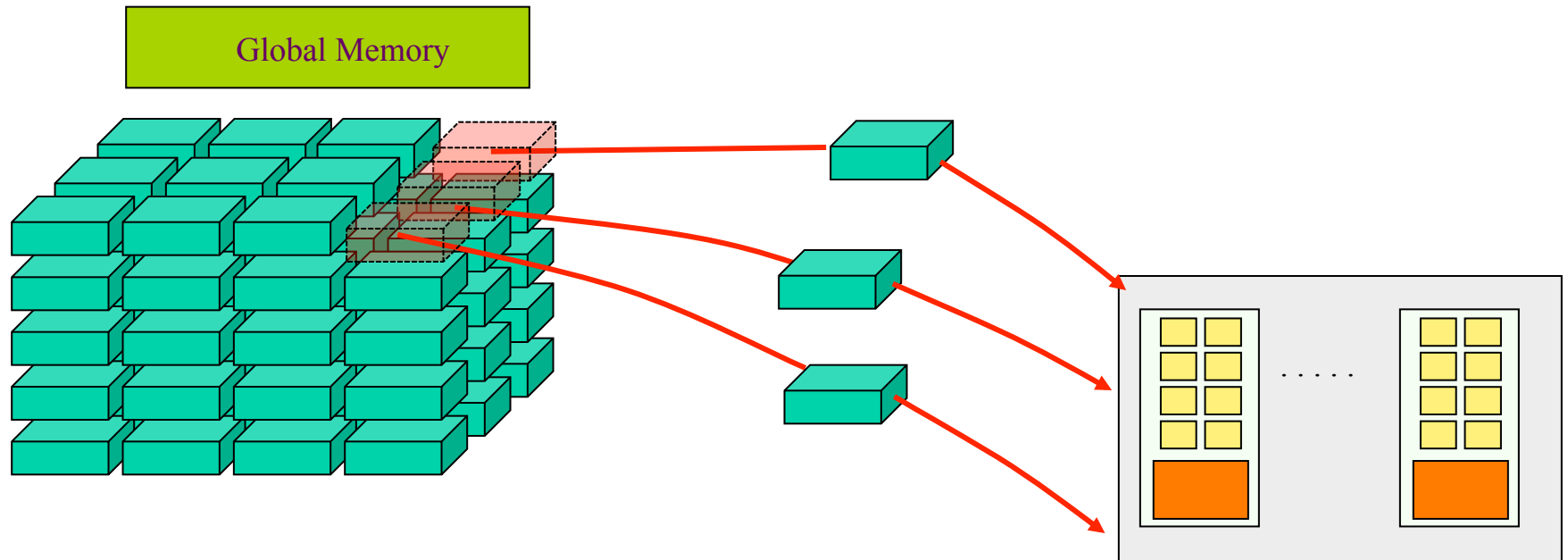
www.anandtech.com/show/3809/nvidias-geforce-gtx-460-the-200-king/2

NVIDIA

CUDA

- Programming environment + C extensions
- Under control of the *host*, run a sequence of multi-threaded GPU kernels on the *device*
- Many lightweight virtualized threads

Grid $\supset$ Block $\supset$ Thread



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`

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);
```

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);
```

Experiments - increment benchmark

- Total time: timing taken from the host, includes copying data to the device
- Device only: time taken on device only

N = 8388480, block size = 128, times in *milliseconds*, cseclass02

Reps = 10 100 1000 10^4 10^5

3.3	36	358	3.58s	35.8s	Device time
71	102	429	3.64s	35.9s	Kernel launch + data xfer
92	730	7.06s	--	--	Host
6.8	52	500			a[i] = 1 + sin(a[i]) : Device)
6.4s	23.9s	200s			Sine function (Host)

Measuring performance

- Two ways
 - ♦ Use an ordinary timer, e.g. gettimeofday()
 - ♦ Use Cuda events/elapsed time (`#ifdef CUDA_TIMER`)
- See incrArray
- Note that kernel invocation is asynchronous

```
cudaThreadSynchronize();  
double t_device_compute = -getTime();  
    incr<<< nBlocks, bSize >>> (a_d, N);  
cudaThreadSynchronize();  
t_device_compute +=getTime();
```

CUDA Error Handling

Cuda error: Can't run kernel: invalid device function.

- Cuda can silently fail, you can observe misleading performance
- E.g. if you specify an invalid grid / thread block dimensions
- Note: the last error can be cleared by successive kernel calls, so check frequently

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

```
checkCUDAError("Unable to allocate storage on the device");
```

- Consult `checkCUDAError()` in `utils.cu` (`incrArr`)
- What about asynchronous calls?
- cf CUDA Programming Guide, “Error Handling”

Getting information about the binary

- Compiler will report a kernel's register usage along with that of local, shared and constant memory

--ptxas-options=-v

incrementArrays (float *a, int N)

int idx = blockIdx.x*blockDim.x + threadIdx.x;

if (idx<N) a[idx] = a[idx]+1.f;

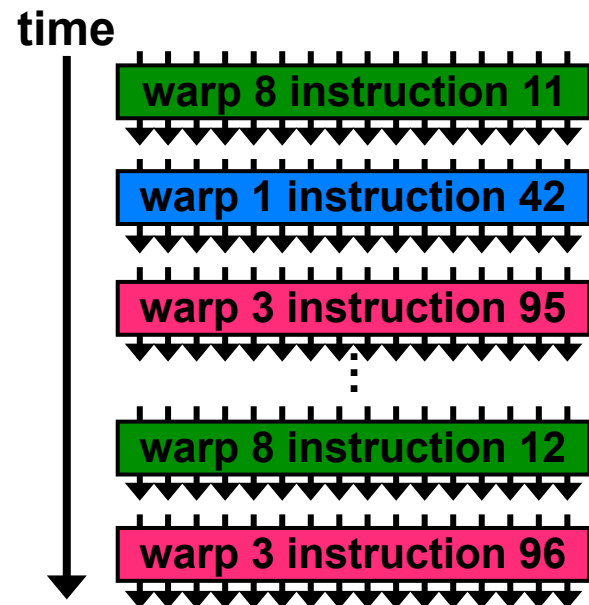
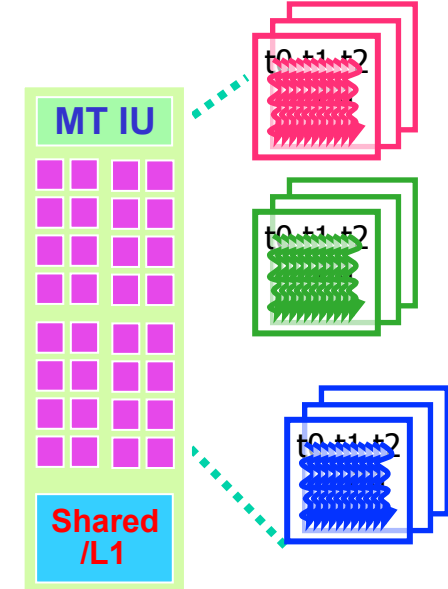
ptxas info : Compiling entry function

'_Z22incrementArrayOnDevicePfii' for 'sm_13'

ptxas info : Used 4 registers, 16+16 bytes smem, 4 bytes cmem[1]

Warp Scheduling (Fermi)

- Threads assigned to an SM in units of a thread block, multiple blocks
- Each block divided into *warps* of 32 (SIMD) threads, a schedulable unit
 - ♦ A warp becomes eligible for execution when all its operands are available
 - ♦ Dynamic instruction reordering: eligible warps selected for execution using a prioritized scheduling policy
 - ♦ All threads in a Warp execute the same instruction, branches serialize execution
- Multiple warps simultaneously active, hiding data transfer delays
- All registers in all the warps are available, 0 overhead scheduling
- Hardware is free to assign blocks to any SM
- Will discuss scheduling later



Today's lecture

- CUDA Programming
- Matrix Multiplication on the GPU

Naïve Host Code

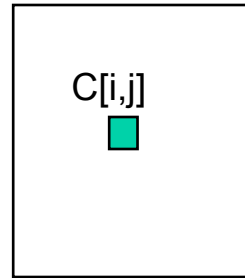
// “ijk” kernel

for i := 0 **to** n-1

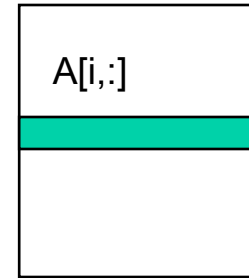
for j := 0 **to** n-1

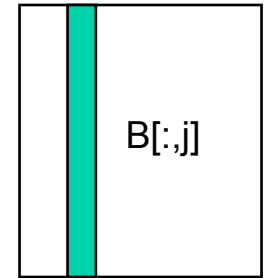
for k := 0 **to** n-1

$C[i,j] += A[i,k] * B[k,j]$



+=

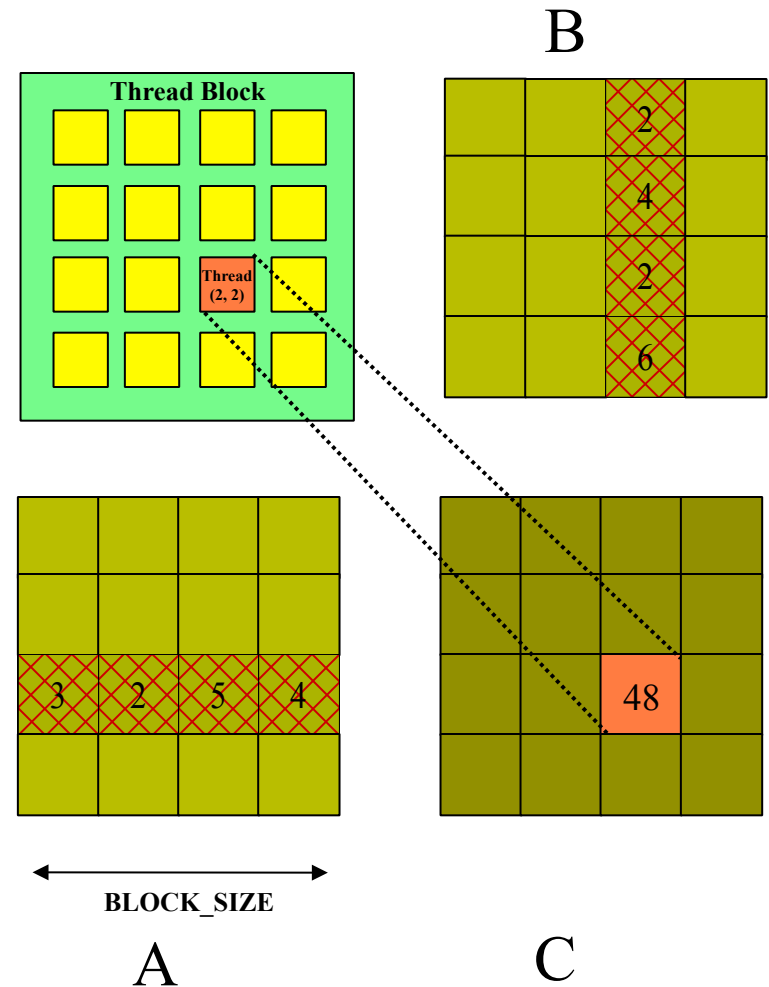




```
for (unsigned int i = 0; i < N; i++)  
    for (unsigned int j = 0; j < N; j++) {  
        DOUBLE sum = 0;  
        for (unsigned int k = 0; k < N; k++)  
            sum += A[i * N + k] * B[k * N + j];  
        C[i * N + j] = (DOUBLE) sum;  
    }
```

Naïve kernel implementation

- Each thread computes one element of C
 - ◆ Loads a row of matrix A
 - ◆ Loads a column of matrix B
 - ◆ Computes a dot product
- Every value of A and B is loaded N times from global memory



Courtesy DavidKirk/NVIDIA and Wen-mei Hwu/UIUC

Naïve Kernel

```
__global__ void
matMul(DOUBLE* C, DOUBLE* A, DOUBLE* B) {
    int I = blockIdx.x*blockDim.x + threadIdx.x;
    int J = blockIdx.y*blockDim.y + threadIdx.y;
    int N = blockDim.y*gridDim.y; // Assume a square matrix
    if ((I < N) && (J < N)){
        float _c = 0;
        for (unsigned int k = 0; k < N; k++) {
            float a = A[I * N + k];
            float b = B[k * N + J];
            _c += a * b;
        }
        C[I * N + J] = _c;
    }
}
```

```
for (unsigned int i = 0; i < N; i++)
    for (unsigned int j = 0; j < N; j++) {
        DOUBLE sum = 0;
        for (unsigned int k = 0; k < N; k++)
            sum += A[i * N + k] * B[k * N + j];
        C[i * N + j] = (DOUBLE) sum;
    }
```

CUDA code on the host side

```
unsigned int n2 = N*N*sizeof(DOUBLE);  
DOUBLE *h_A = (DOUBLE*) malloc(n2);  
DOUBLE *h_B = (DOUBLE*) malloc(n2);  
// Check that allocations went OK  
assert(h_A); assert(h_B);  
  
genMatrix(h_A, N, N); genMatrix(h_B, N, N); // Initialize matrices  
  
DOUBLE *d_A, *d_B, *d_C;  
cudaMalloc((void**) &d_A, n2); ... &d_A ... &d_B  
checkCUDAError("Error allocating device memory arrays");  
  
// copy host memory to device  
cudaMemcpy(d_A, h_A, n2, cudaMemcpyHostToDevice);  
checkCUDAError("Error copying data to device");  
cudaMemcpy(d_B, h_B, n2, cudaMemcpyHostToDevice);  
checkCUDAError("Error copying data to device");
```

Host code - continued

```
// setup execution configurations
```

```
dim3 threads(ntx, nty, 1);    // ntx & nty are user input  
dim3 grid(N / threads.x, N / threads.y);
```

```
// launch the kernel
```

```
matMul<<< grid, threads >>>(d_C, d_A, d_B);
```

```
// retrieve result
```

```
cudaMemcpy(h_C, d_C, n2, cudaMemcpyDeviceToHost);  
checkCUDAError("Unable to retrieve result from device");
```

```
// Free device storage
```

```
assert(cudaSuccess == cudaFree(d_A));  
assert(cudaSuccess == cudaFree(d_B));  
assert(cudaSuccess == cudaFree(d_C));
```

Configuration variables

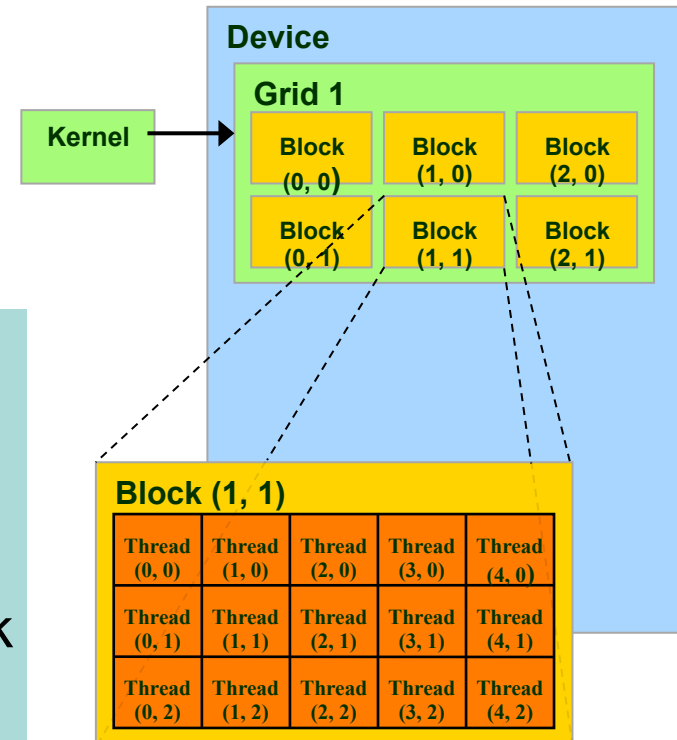
- Types to manage thread geometries
- `dim3 gridDim, blockDim`
 - ♦ Dimensions of the grid in blocks
(`gridDim.z` not used)
 - ♦ Dimensions of a thread block in threads
- `dim3 blockIdx, threadIdx;`
 - ♦ Block index within the grid
 - ♦ Thread index within the block

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

Execution Configurations

- Grid $\supset$ Block $\supset$ Thread

```
__global__ void Kernel (...);  
  
dim3 DimGrid(2,3);    // 6 thread blocks  
  
dim3 DimBlock(3,5,1); // 15 threads /block  
  
Kernel<<< DimGrid, DimBlock, >>>(...);
```



DavidKirk/NVIDIA & Wen-mei Hwu/UIUC

Performance

- Baseline [N=512, double precision]
 - ♦ Lilliput, C1060, 2.0 GHz Intel Xeon E5504, 4MB L3, peak 8.0 GF / core
 - ♦ **Forge, M2070 14×32 cores**
 - ♦ 21 GF on 4 CPU cores (MPI), 25 Gflops for N=2K

Gflops dp, C1060	9.8	8.5	7.4	5.9	5.3	5.1	3.0	2.7
Geometry	2×256	2×128	2×64	4×1 28	4×64 2×32	4×32	8×64	8×32

Gflops sp, C1060	8.6	7.7	6.2	4.6	3.9	3.5	2.0	1.8
Geometry	2×256	2×128	2×32 2×64	4×128	4×64	4×32	8×64	8×32

Gflops sp	50,49,46,..., 9.5
Dirac dp	69,69,68,..., 6.6
Geometry	2×256, 2×128, 2×64,..., 16×16 2×256, 2×128, 2×64,..., 16×16

Summary - Programming issues

- Branches serialize execution within a warp
- Tradeoff: more blocks with fewer threads or more threads with fewer blocks
 - ♦ Locality: want small blocks of data (and hence more plentiful warps) that fit into fast memory
 - ♦ Register consumption
 - ♦ Scheduling: hide latency
- Shared memory and registers do not persist across kernel invocations
- Next time: using shared memory