NCSA GPU programming tutorial day 3

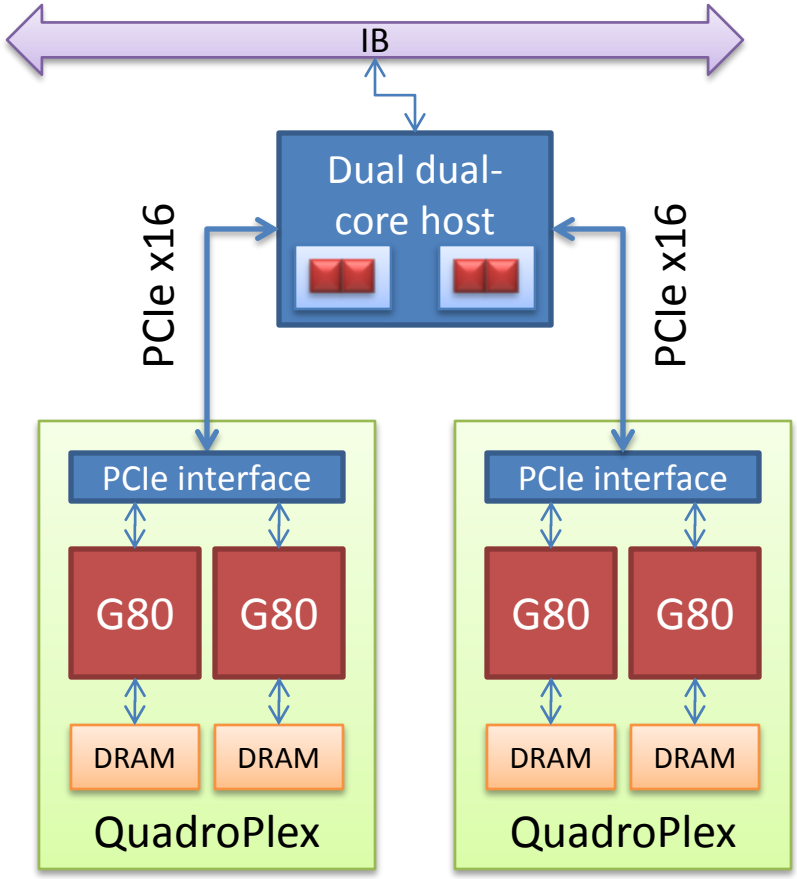
Vlad Kindratenko
kindr@ncsa.uiuc.edu

Tutorial outline

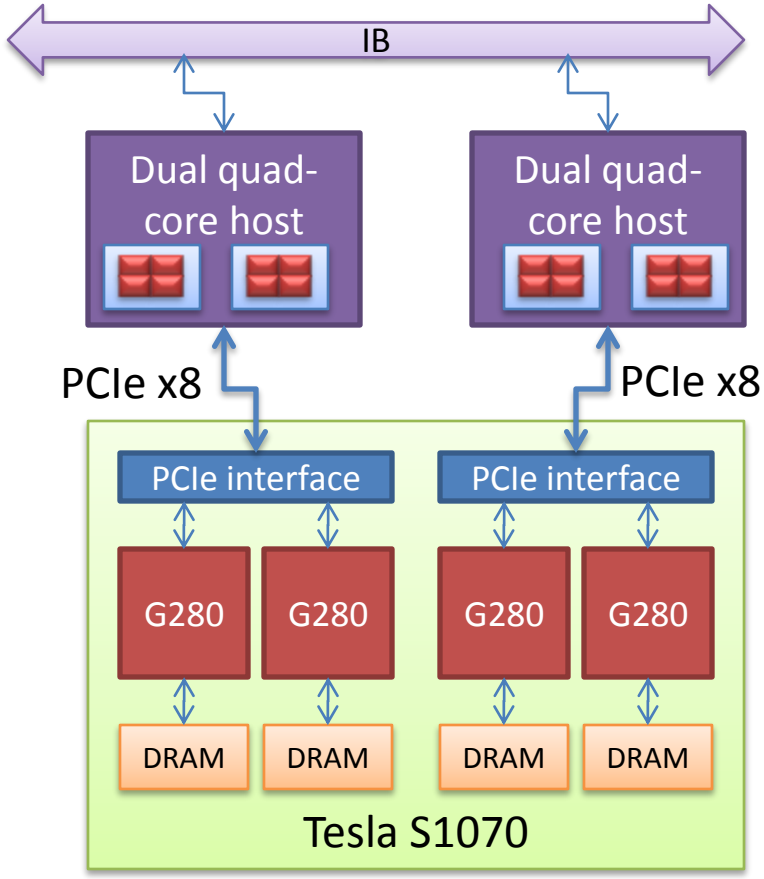
- Random facts about NCSA systems, GPUs, and CUDA
  - QP & Lincoln cluster configurations
  - Tesla S1070 architecture
  - Memory alignment for GPU
  - CUDA APIs
- Matrix-matrix multiplication example
  - K1: 27 GFLOPS
  - K2: 44 GFLOPS
  - K3: 43 GFLOPS
  - K4: 169 GFLOPS
  - K3+K4: 173 GFLOPS
  - Other implementations

QP & Lincoln node configurations

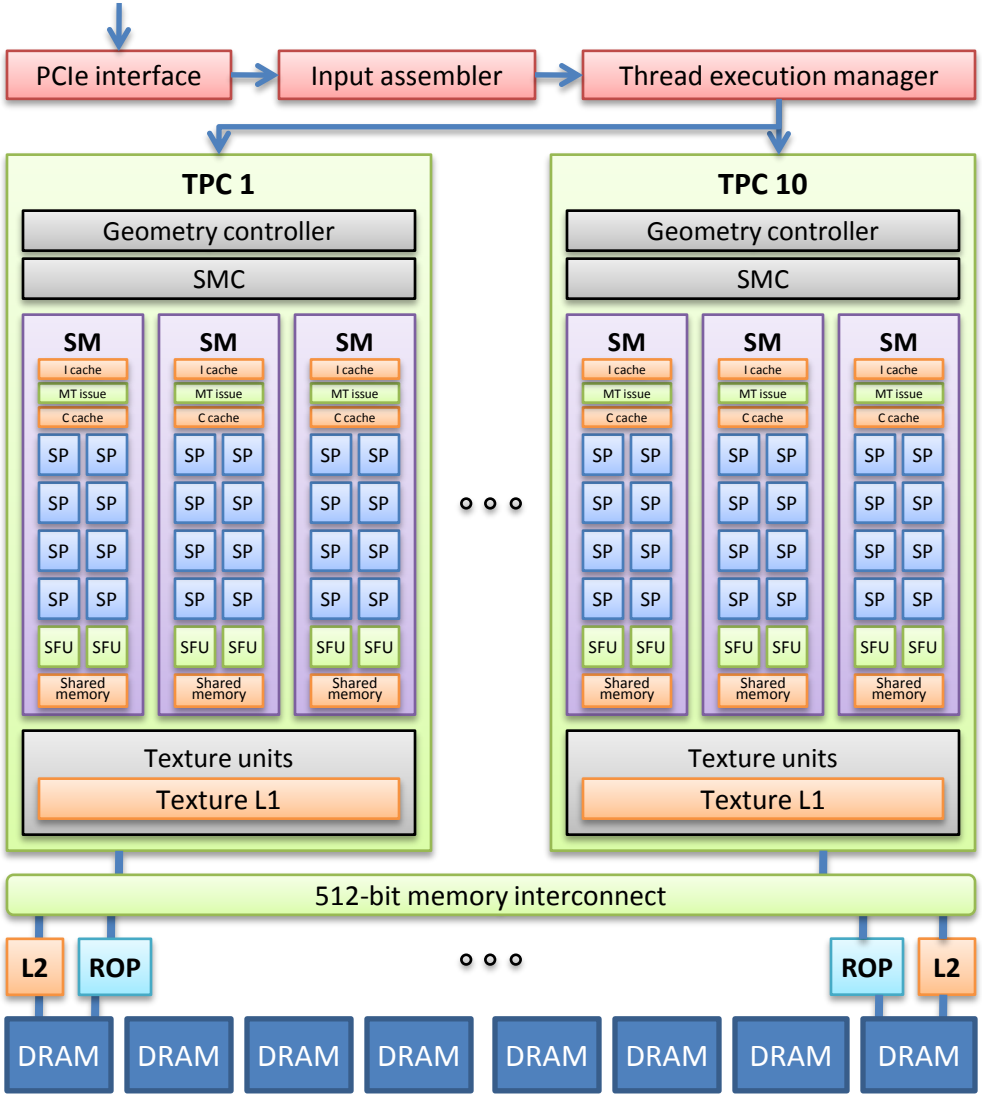
- QP



- Lincoln



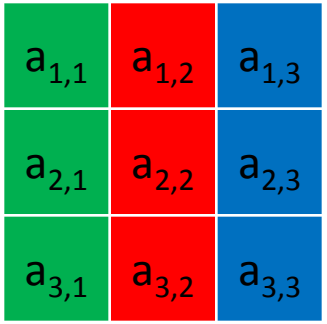
Tesla S1070 architecture



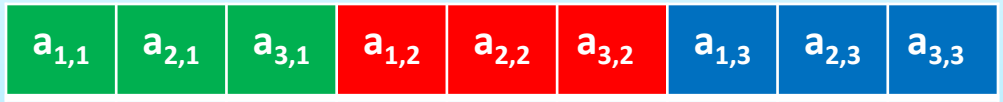
The basic processing unit is the streaming processor (SP), a fully pipelined, single-issue, in-order microprocessor complete with two ALUs and an FPU. A group of 8 SPs, 2 additional special function units (SFUs) for transcendental operations, and 16kB of shared memory form a streaming multiprocessor (SM). Each SM has a small instruction cache and a read only data cache. A group of 3 SMs with some additional memory form texture/processor cluster (TPC). Ten such clusters form a streaming processor array (SPA). In total, a T10 has 240 SPs that run at 1.44 GHz. A 512-bit interface to off-chip GDDR3 memory provides 102 GB/s bandwidth.

"To manage hundreds of threads running several different programs, the multiprocessor employs a new architecture we call SIMT (single-instruction, multiple-thread). The multiprocessor maps each thread to one scalar processor core, and each scalar thread executes independently with its own instruction address and register state. The multiprocessor SIMT unit creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps. (This term originates from weaving, the first parallel thread technology. A half-warp is either the first or second half of a warp.) Individual threads composing a SIMT warp start together at the same program address but are otherwise free to branch and execute independently." (CUDA Programming Guide)

Memory alignment for GPU

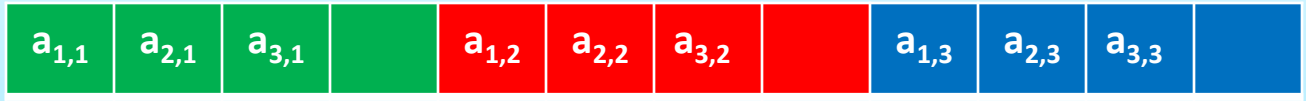


cudaMalloc(&dev_a, m*n*sizeof(float));



Matrix columns are not aligned at 64-bit boundary

cudaMallocPitch(&dev_a, &n, n*sizeof(float), m);

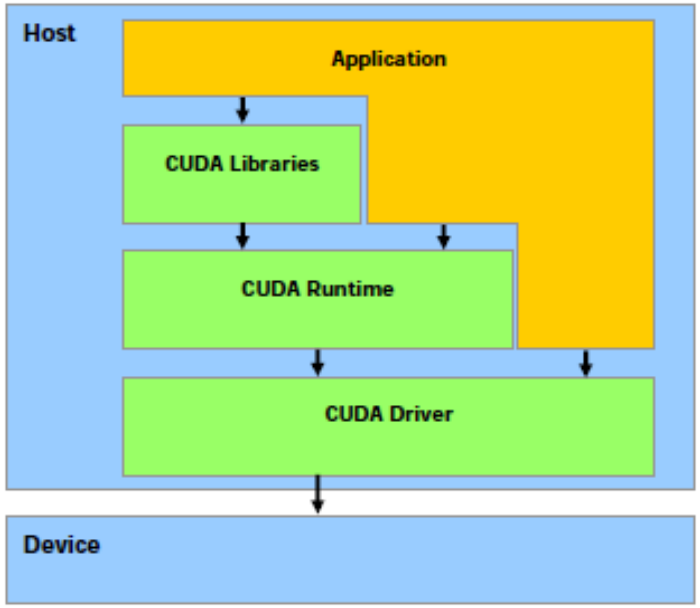


Matrix columns are aligned at 64-bit boundary

n is the allocated (aligned) size for the first dimension (the *pitch*), given the requested sizes of the two dimensions.

CUDA APIs

- higher-level API called the **CUDA runtime API**
  - myKernelunsigned char*)devPtr, width, <<<Dg, Db>>>((height, pitch);



- low-level API called the **CUDA driver API**
  - cuModuleLoad(&module, binfile);
  - cuModuleGetFunction(&func, module, "mmkernel");
  - …
  - cuParamSetv(func, 0, &args, 48);
  - cuParamSetSize(func, 48);
  - cuFuncSetBlockShape(func, ts[0], ts[1], 1);
  - cuLaunchGrid(func, gs[0], gs[1]);

The rest of the presentation is based on ***Compilers and More: Optimizing GPU Kernels*** *by* Michael Wolfe, Compiler Engineer, The Portland Group, Inc.

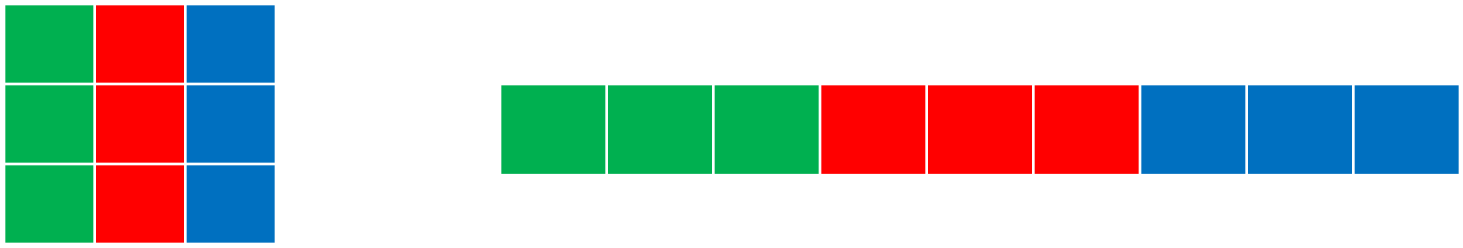
Original source:
http://www.hpcwire.com/features/Compilers_and_More_Optimizing_GPU_Kernels.html

Files for the tutorial:
{honest2|qp}:/tmp/gpututorial/mmexample.tar

Matrix-matrix multiplication example (BLAS SGEMM)

```
for (i = 0; i < n; ++i)
    for (j = 0; j < m; ++j)
        for (k = 0; k < p; ++k)
            a[i+n*j] += b[i+n*k] * c[k+p*j];
```

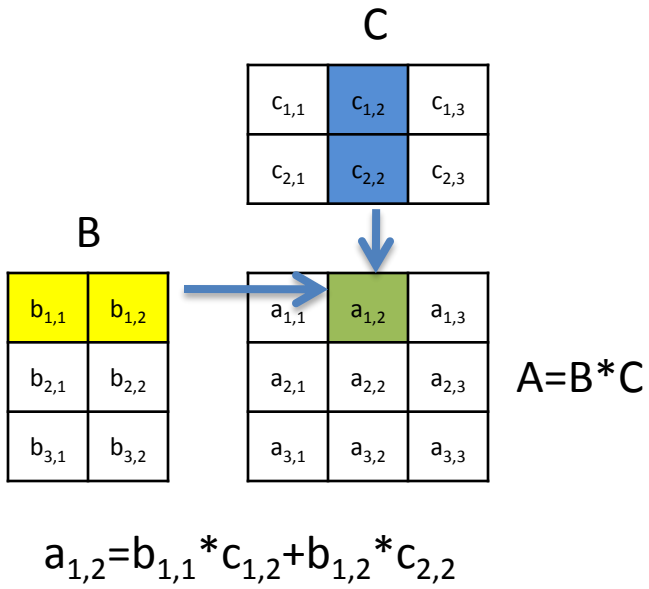
- Matrices are stored in column-major order



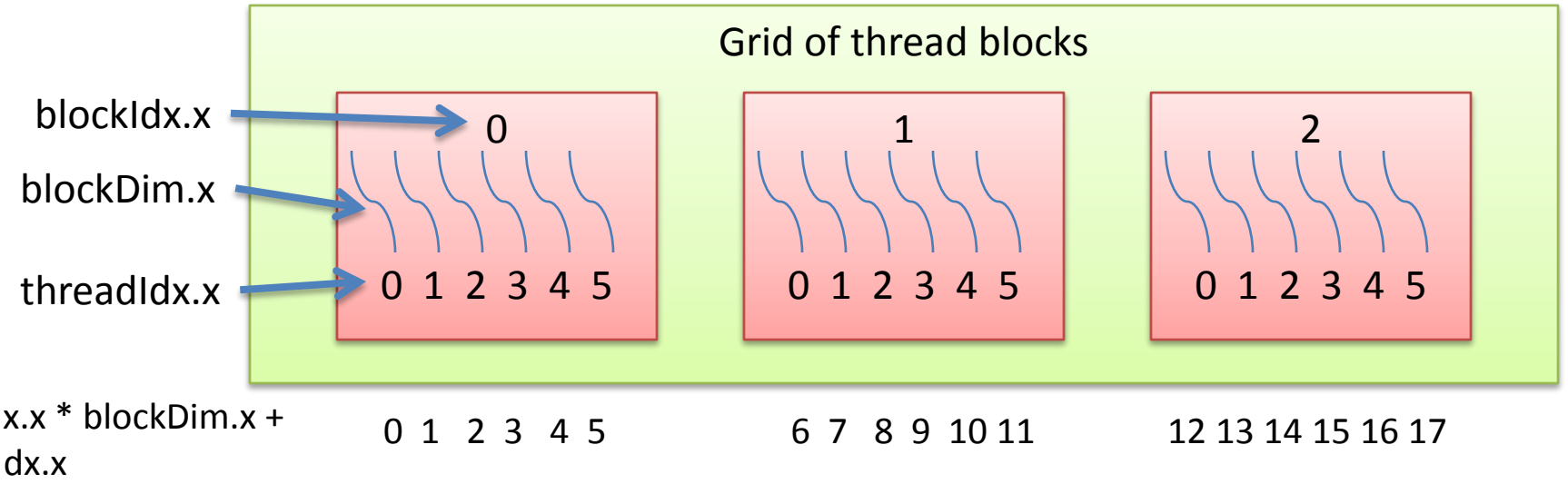
- For reference, jki-ordered version runs at 1.7 GFLOPS on 3 GHz Intel Xeon (single core)

Map this code:

```
for (i = 0; i < n; ++i)
    for (j = 0; j < m; ++j)
        for (k = 0; k < p; ++k)
            a[i+n*j] += b[i+n*k] * c[k+p*j];
```



into this (logical) architecture:



Strip-mined the stride-1 *i* loop to SIMD width of 32

```
for (i = 0; i < n; ++i)
   for (j = 0; j < m; ++j)
      for (k = 0; k < p; ++k)
         a[i+n*j] += b[i+n*k] * c[k+p*j];
```

```
for (is = 0; i < n; is+=32)
  for (i = is; i < is+32; ++i)
    for (j = 0; j < m; ++j)
      for (k = 0; k < p; ++k)
        a[i+n*j] += b[i+n*k] * c[k+n*j];
```

Run the *i* element as a thread block and the *is* strip loop and *j* loop in parallel

```
for (is = 0; i < n; is+=32)
  for (i = is; i < is+32; ++i)
    for (j = 0; j < m; ++j)
      for (k = 0; k < p; ++k)
        a[i+n*j] += b[i+n*k] * c[k+p*j];
```

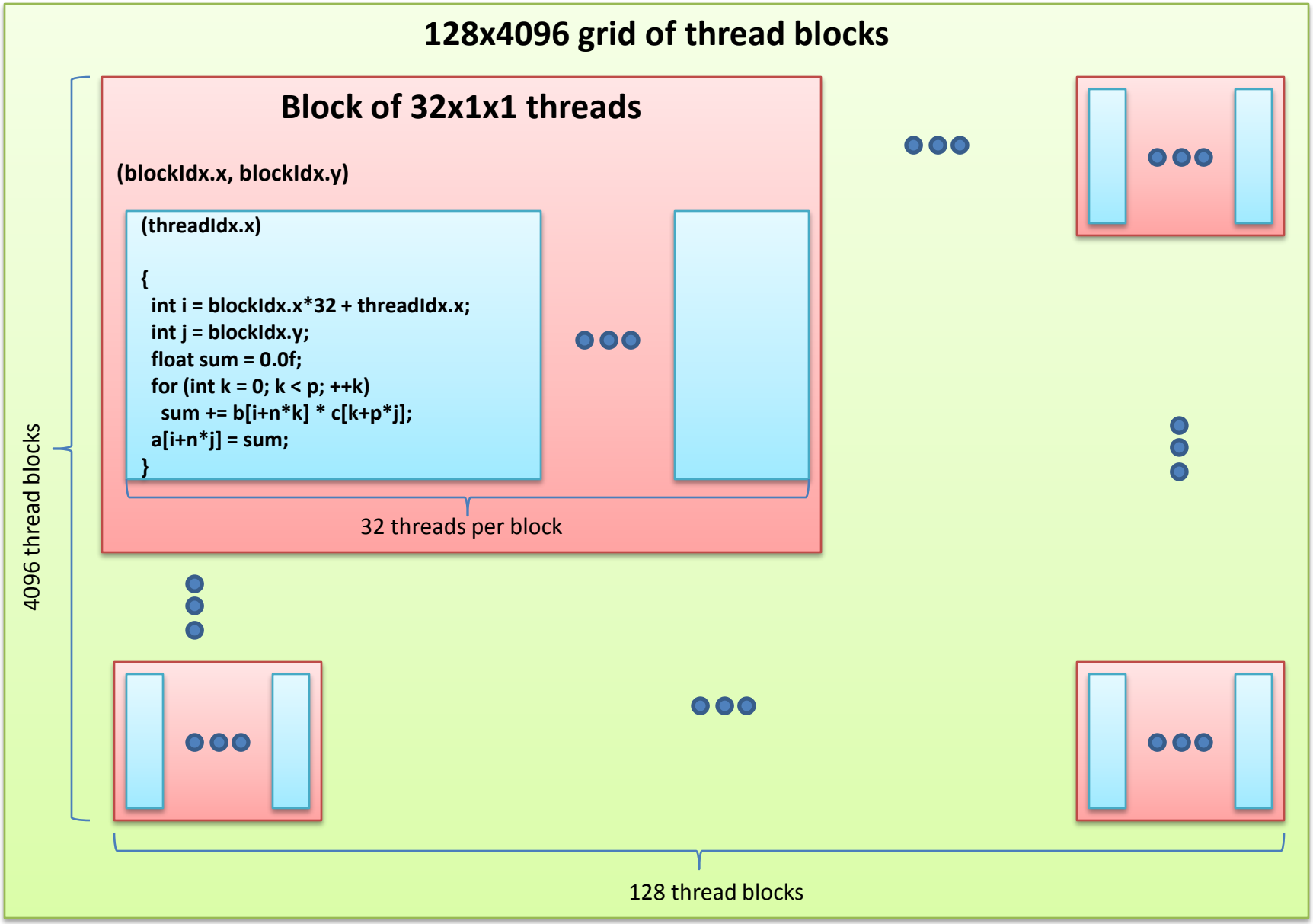
```
parfor (is = 0; i < n; is+=32)
  parfor (j = 0; j < m; ++j)
    SIMDfor (i = is; i < is+32; ++i)
      for (k = 0; k < p; ++k)
        a[i+n*j] += b[i+n*k] * c[k+p*j];
```

Parallel (grid) loops and SIMD (thread block) loop are handled implicitly by the GPU hardware

```
parfor (is = 0; i < n; is+=32)
  parfor (j = 0; j < m; ++j)
    SIMDfor (i = is; i < is+32; ++i)
      for (k = 0; k < p; ++k)
        a[i+n*j] += b[i+n*k] * c[k+p*j];
```

```
extern "C" __global__ void mmkernel (float* a, float* b, float* c, int n, int m,
int p)
{
   int i = blockIdx.x*32 + threadIdx.x;
   int j = blockIdx.y;
   float sum = 0.0f;
   for (int k = 0; k < p; ++k)  sum += b[i+n*k] * c[k+p*j];
   a[i+n*j] = sum;
}
```

dim3 threads (32);
dim3 grid(4096/32, 4096);



K1

```
int i = blockIdx.x*32 + threadIdx.x;
int j = blockIdx.y;
float sum = 0.0f;
for (int k = 0; k < p; ++k)  sum += b[i+n*k] * c[k+p*j];
a[i+n*j] = sum;
```

./mmdriver -bin k1.bin -block 128 4096 -thread 32 1 -mat 4096 -size 4096 -iter 3
binfile=k1.bin array=4096x4096 matrix=4096x4096 block=<128x4096> thread=<32x1>
matrix = 4096x4096
array = 4096x4096
grid = 128x4096
block = 32x1x1
flops = 137438953472
msec = 5890912 GFLOPS = 23.33, 24.19 (kernel)
msec = 5898023 GFLOPS = 23.30, 24.16 (kernel)
msec = 5882167 GFLOPS = 23.37, 24.23 (kernel)

Hands-on

- On honest2|qp **host** node
  - mkdir gpuclass3
  - cd gpuclass3
  - cp /tmp/gpututorial/mmexample.tar .
  - tar -xvf mmexample.tar
  - make
  - qsub -I -V -l walltime=01:00:00,nodes=1:ppn=4

- On honest2|qp GPU **compute** node
  - cat k1.cu
  - ./krun k1 4096
  - diff k1.cu k1-64.cu
  - ./krun k1-64 4096
  - diff k1.cu k1-128.cu
  - ./krun k1-128 4096
  - diff k1.cu k1-256.cu
  - ./krun k1-256 4096

K1

Threads per block (SIMD width)	Grid size	performance (GFLOPS)		
		Overall	kernel	
32	128x4096	23.3	24.2	One warp per thread block. Thus, at most only 8 thread blocks are active on each multiprocessor, out of 32 max.
64	64x4096	26.0	27.0	If 8 thread blocks are scheduled on each multiprocessor, we get up to 16 warps, so the multithreading is more efficient.
128	32x4096	24.5	25.3	
256	16x4096	24.9	25.9	

Strip-mine *k* loop and load a strip of *c* into the multiprocessor local memory

```
parfor (is = 0; i < n; is+=32)
  parfor (j = 0; j < m; ++j)
    SIMDfor (i = is; i < is+32; ++i)
      for (k = 0; k < p; ++k)
        a[i+n*j] += b[i+n*k] * c[k+p*j];
```

```
parfor (is = 0; i < n; is+=32)
  parfor (j = 0; j < m; ++j)
    SIMDfor (i = is; i < is+32; ++i)
      for (ks=0; ks<p; ks+=32)
        cb[ks:ks+31]=c[ks+p*j:ks+31+p*j];
        for (k = ks; k < ks+32; ++k)
          a[i+n*j] += b[i+n*k] * cb[k-ks];
```

K2

```
int tx = threadIdx.x;    int i = blockIdx.x*32 + tx;   int j = blockIdx.y;
__shared__ float cb[32];
float sum = 0.0f;
for (int ks = 0; ks < p; ks += 32) {
  cb[tx] = c[ks+tx+p*j];
  for (int k = ks; k < ks+32; ++k) sum += b[i+n*k] * cb[k-ks];
}
a[i+n*j] = sum;
```

./mmdriver -bin k2.bin -block 128 4096 -thread 32 1 -mat 4096 -size 4096
binfile=k2.bin array=4096x4096 matrix=4096x4096 block=<128x4096> thread=<32x1>
matrix = 4096x4096
array = 4096x4096
grid = 128x4096
block = 32x1x1
flops = 137438953472
msec = 4833055 GFLOPS = 28.44, 29.74 (kernel)

Hands-on

- On honest2|qp GPU **compute** node
  - cat k2.cu
  - ./krun k2 4096

  - k2.cu k2-64.cu k2-128.cu k2-256.cu

  - k2-64x2.cu k2-64x4.cu k2-64x16.cu
  - k2-128x2.cu k2-128x4.cu k2-128x16.cu

K2

Threads per block (SIMD width)	Grid size	performance (GFLOPS)		
		Overall	kernel	
32	128x4096	28.5	29.8	
64	64x4096	40.4	43.1	
128	32x4096	40.5	43.2	
256	16x4096	41.2	44.0	

Each kernel instance computes 2 values of the *i* loop

```
parfor (is = 0; i < n; is+=32)
  parfor (j = 0; j < m; ++j)
    SIMDfor (i = is; i < is+32; ++i)
      for (ks=0; ks<p; ks+=32)
        cb[ks:ks+31]=c[ks+p*j:ks+31+p*j];
        for (k = ks; k < ks+32; ++k)
          a[i+n*j] += b[i+n*k] * cb[k-ks];
```

```
parfor (is = 0; i < n; is+=64)
  parfor (j = 0; j < m; ++j)
    SIMDfor (i = is; i < is+32; ++i)
      for (ks=0; ks<p; ks+=32)
        cb[ks:ks+31]=c[ks+p*j:ks+31+p*j];
        for (k = ks; k < ks+32; ++k)
          a[i+n*j] += b[i+n*k] * cb[k-ks];
          a[i+32+n*j] += b[i+32+n*k] * cb[k-ks];
```

K3

```
int tx = threadIdx.x;   int i = blockIdx.x*64 + tx;   int j = blockIdx.y;
__shared__ float cb[32];
float sum0 = 0.0f, sum1 = 0.0f;
for (int ks = 0; ks < p; ks += 32) {
  cb[tx] = c[ks+tx+p*j];
  __syncthreads();
  for (int k = ks; k < ks+32; ++k) {  sum0 += b[i+n*k] * cb[k-ks];  sum1 += b[i+32+n*k] * cb[k-ks];  }
  __syncthreads();
}
a[i+n*j] = sum0;
a[i+32+n*j] = sum1;
```

./mmdriver -bin k3.bin -block 64 4096 -thread 32 1 -mat 4096 -size 4096
binfile=k3.bin array=4096x4096 matrix=4096x4096 block=<64x4096> thread=<32x1>
matrix = 4096x4096
array = 4096x4096
grid = 64x4096
block = 32x1x1
flops = 137438953472
msec = 3399447 GFLOPS = 40.43, 43.11 (kernel)

Hands-on

- On honest2|qp GPU **compute** node
  - cat k3.cu
  - ./krun k3 4096

  - k3.cu k3x4.cu
  - k3-64.cu k3-64x4.cu
  - k3-128.cu k3-128x4.cu

K3

x2

Threads per block (SIMD width)	Grid size	performance (GFLOPS)		
		overall	kernel	
32	128x4096	40.4	43.1	
64	64x4096	40.7	43.4	
128	32x4096	40.8	43.6	

x4

Threads per block (SIMD width)	Grid size	performance (GFLOPS)		
		overall	kernel	
32	128x4096	40.8	43.4	
64	64x4096	40.9	43.6	
128	32x4096	39.6	42.1	

Each kernel instance computes 2 values of the *j* loop

```
parfor (is = 0; i < n; is+=32)
  parfor (j = 0; j < m; ++j)
    SIMDfor (i = is; i < is+32; ++i)
      for (ks=0; ks<p; ks+=32)
        cb[ks:ks+31]=c[ks+p*j:ks+31+p*j];
        for (k = ks; k < ks+32; ++k)
          a[i+n*j] += b[i+n*k] * cb[k-ks];
```

```
parfor (is = 0; i < n; is+=32)
  parfor (j = 0; j < m; j+=2)
    SIMDfor (i = is; i < is+32; ++i)
      for (ks=0; ks<p; ks+=32)
        cb0[ks:ks+31]=c[ks+p*j:ks+31+p*j];
        cb1[ks:ks+31]=c[ks+p*(j+1):ks+31+p*(j+1)];
        for (k = ks; k < ks+32; ++k)
          a[i+n*j] += b[i+n*k] * cb0[k-ks];
          a[i+32+n*(j+1)] += b[i+32+n*k] * cb1[k-ks];
```

K4

```
int tx = threadIdx.x;    int i = blockIdx.x*32 + tx;    int j = blockIdx.y*2;
__shared__ float cb0[32], cb1[32];
float sum0 = 0.0f, sum1 = 0.0f;
for (int ks = 0; ks < p; ks += 32) {
  cb0[tx] = c[ks+tx+p*j];
  cb1[tx] = c[ks+tx+p*(j+1)];
  __syncthreads();
  for (int k = ks; k < ks+32; ++k) { float rb = b[i+n*k];  sum0 += rb * cb0[k-ks];  sum1 += rb * cb1[k-ks]; }
  __syncthreads();
}
a[i+n*j] = sum0;
a[i+n*(j+1)] = sum1;
```

./mmdriver -bin k4.bin -block 128 2048 -thread 32 1 -mat 4096 -size 4096
binfile=k4.bin array=4096x4096 matrix=4096x4096 block=<128x2048> thread=<32x1>
matrix = 4096x4096
array = 4096x4096
grid = 128x2048
block = 32x1x1
flops = 137438953472
msec = 2608078 GFLOPS = 52.70, 57.25 (kernel)

Hands-on

- On honest2|qp GPU **compute** node
  - cat k4.cu
  - ./krun k4 4096

  - k4.cu k4x4.cu
  - k4-64.cu k4-64x4.cu
  - k4-128.cu k4-128x4.cu k4-128x8.cu
  - k4-256.cu

K4

x2

Threads per block (SIMD width)	Grid size	performance (GFLOPS)		
		overall	kernel	
32	128x4096	52.7	57.3	
64	64x4096	75.2	84.8	
128	32x4096	76.2	86.3	

x4

Threads per block (SIMD width)	Grid size	performance (GFLOPS)		
		overall	kernel	
32	128x4096	92.3	107.4	
64	64x4096	131.3	163.8	
128	32x4096	134.6	169.1	

Hands-on

- On honest2|qp GPU **compute** node
  - cat k4-128x4x24.cu
  - ./krun k4-128x4x2 4096

  - k4-128x4x2.cu
  - k4-128x4x4.cu
  - k4-128y2x4x4.cu

K3+K4

x4x4

Threads per block (SIMD width)	Grid size	performance (GFLOPS)		
		overall	kernel	
128	32x4096	137.3	173.3	x4x2
128	32x4096	133.0	167.0	x4x4

Other implementations

- CUDA programming guide, page 71: Example of Matrix Multiplication
  - http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf
- CUDA BLAS library, page 83: cublasSgemm
  - http://developer.download.nvidia.com/compute/cuda/2_0/docs/CUBLAS_Library_2.0.pdf
- Vasily Volkov, UC Berkeley
  - http://www.cs.berkeley.edu/~volkov/volkov08-sc08talk.pdf

Bottom line

- It is easy enough to get something to run on a GPU
- But it is difficult to get it to run fast
  - Things to consider
    - which algorithm to use; some algorithms are better suited for GPUs than others
    - understand if the kernel is compute-bound or memory I/O bound and optimize it accordingly