GPGPUGPGPU: Multi-GPU Programming

Fall 2012

CITA|ICAT

SciNet

HW4

```
__global__ void cuda_transpose(const float *ad, const int n,
                               float *atd) {

    int i = threadIdx.y + blockIdx.y*blockDim.y;
    int j = threadIdx.x + blockIdx.x*blockDim.x;
    if (i<n && j<n)
        atd[i*n + j] = ad[j*n + i];
    return;

}
```

HW4

```
__global__ void cuda_transpose_shared(const float *ad, const int n, float *atd) {

    extern __shared__ float atile[];

    int loci = threadIdx.y, locj = threadIdx.x;
    int bi = blockIdx.y,    bj = blockIdx.x;
    int blocksize = blockDim.x;
    int i = loci + bi*blocksize;
    int j = locj + bj*blocksize;

    /* read in shared data */
    if (i<n && j<n) {
        atile[loci*blocksize + locj] = ad[i*n + j];
    }
    __syncthreads();

    if (i<n && j<n) {
        atd[(loci+bj*blocksize)*n + (locj+bi*blocksize)] = atile[locj*blocksize+loci];
    }
    return;
}
```

HW4

- Advantages: coallesced reads, writes
- (what is optimum blocksize for coallescing? Can get 100% memory efficiency)
- Downsides: __syncthreads()

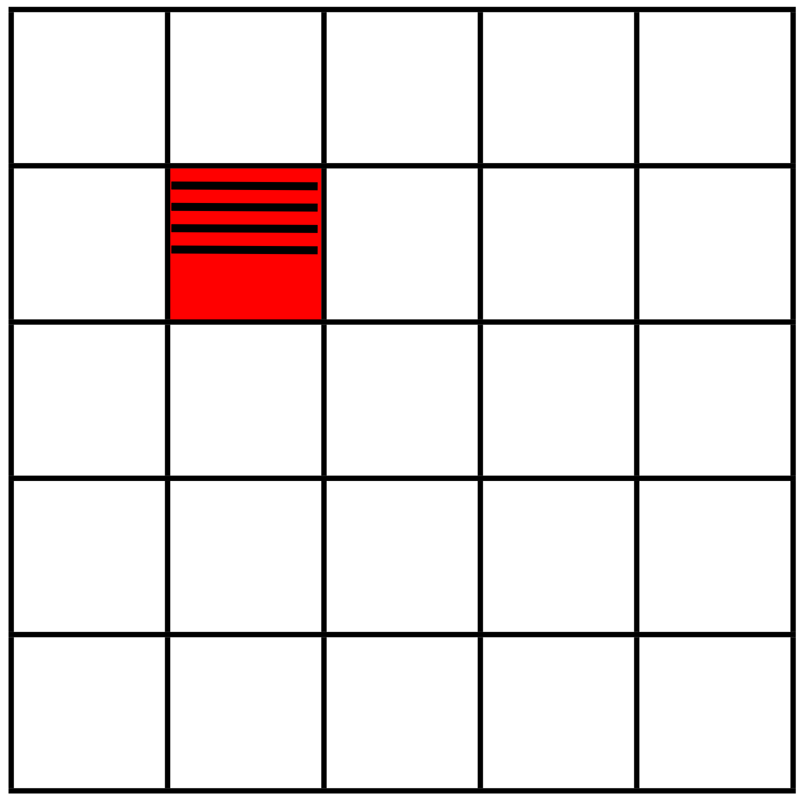
```
__global__ void cuda_transpose_shared(const float *ac

    extern __shared__ float atile[];

    int loci = threadIdx.y, locj = threadIdx.x;
    int bi = blockIdx.y,    bj = blockIdx.x;
    int blocksize = blockDim.x;
    int i = loci + bi*blocksize;
    int j = locj + bj*blocksize;

    /* read in shared data */
    if (i<n && j<n) {
        atile[loci*blocksize + locj] = ad[i*n + j];
    }
    __syncthreads();

    if (i<n && j<n) {
        atd[(loci+bj*blocksize)*n + (locj+bi*blocksiz
    }
    return;
}
```

CITA|ICAT

SciNet

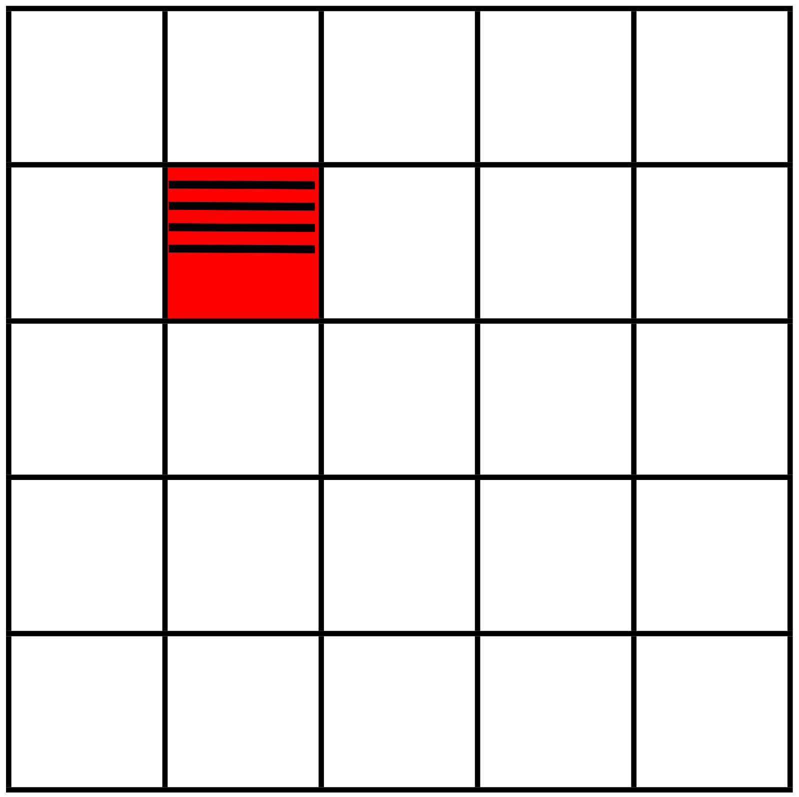
HW4

- 32x32 blocks -
  - perfect coallescing
  - have to wait for 32 memory transactions to complete before proceed
- At least 132 cycles for scheduling
- Can get around with more blocks...
- But 4kB/block shared; past 12 blocks per SM (=168, 400^2 matrix) occupancy suffers.



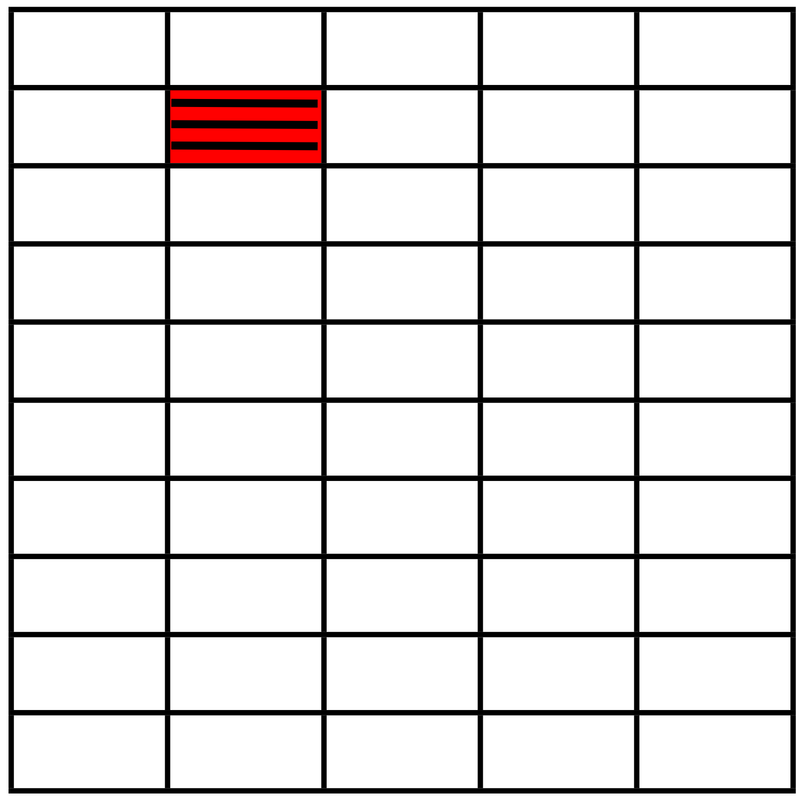
HW4

- What can we do?
- Want to either reduce blocks shared memory footprint, or more fewer waits before __syncthreads() without impacting coallescing.



HW4

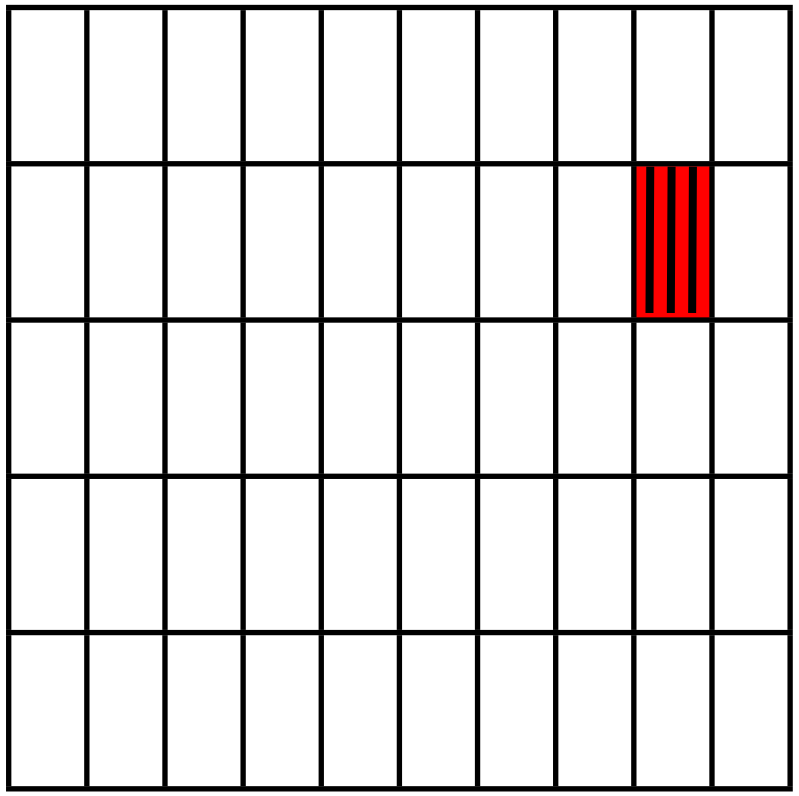
- Narrower blocks?



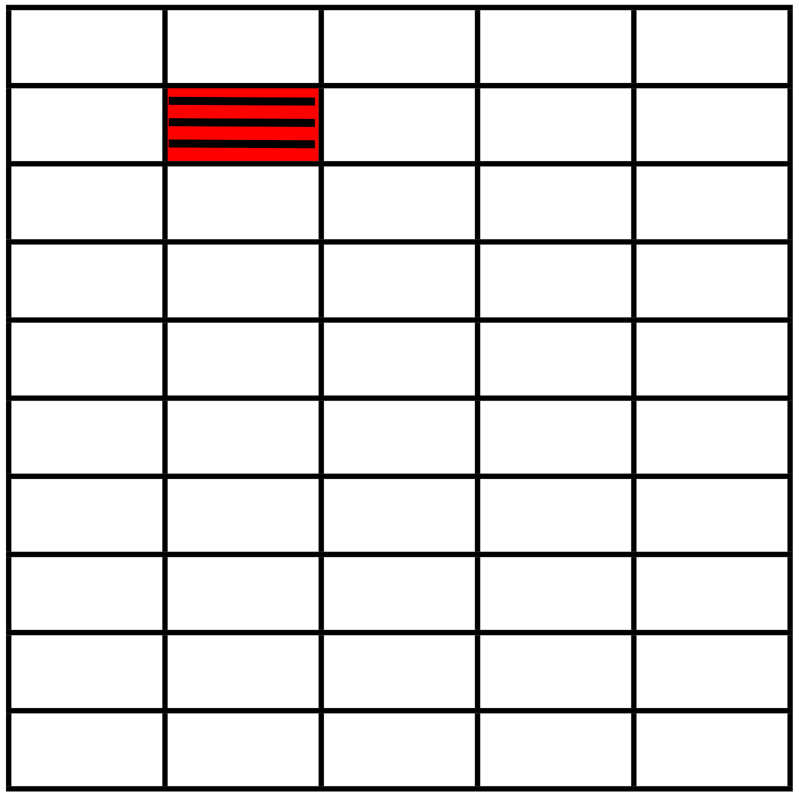
CITA|ICAT

HW4

- Narrower blocks?
- Means outputting the transpose is inefficient



HW4



- Multiple transactions per thread...

CITA|ICAT

SciNet

HW4

```
    if (j<n) {
        for (int iter=0; iter<multi; iter++) {
            int inrow  =multi*i+iter;
            int tilerow=multi*loci+iter;
            if ( inrow < n) {
                atile[tilerow*(blockDim.x+1) + locj] = ad[inrow*n + j];
            }
        }
    }

    __syncthreads();

    for (int iter=0; iter<multi; iter++) {
        int outrow=blockDim.x*bj + loci/multi;
        int outcol=blockDim.y*bi*multi + (loci%multi)*blockDim.y + locj;

        if (outrow < n && outcol < n) {
            atd[outrow*n + outcol] = atile[(multi*locj+iter)*(blockDim.x+1)+loci];
        }
    }
    return;
```

HW4

```
arc01-$ ./transpose --matsize=384 --nblocks=12 --multi=4
Matrix size = 384, Number of blocks = 12, Blocksize = 32.
CPU time = 1.161 millisec.
GPU time = 0.529 millisec (global mem).
CUDA global mem and CPU results differ by 0.000000
GPU time = 0.571 millisec (shared mem).
CUDA shared mem and CPU results differ by 0.000000
GPU time = 0.583 millisec (shared mem).
CUDA shared mem - bank and CPU results differ by 0.000000
GPU time = 0.499 millisec (multiple transactions).
CUDA multiple transactions and CPU results differ by 0.000000

arc01-$ ./transpose --matsize=1024 --nblocks=32 --multi=4
Matrix size = 1024, Number of blocks = 32, Blocksize = 32.
CPU time = 11.133 millisec.
GPU time = 2.606 millisec (global mem).
CUDA global mem and CPU results differ by 0.000000
GPU time = 2.841 millisec (shared mem).
CUDA shared mem and CPU results differ by 0.000000
GPU time = 2.815 millisec (shared mem).
CUDA shared mem - bank and CPU results differ by 0.000000
GPU time = 2.333 millisec (multiple transactions).
```

Optimizizing around Tradeoffs

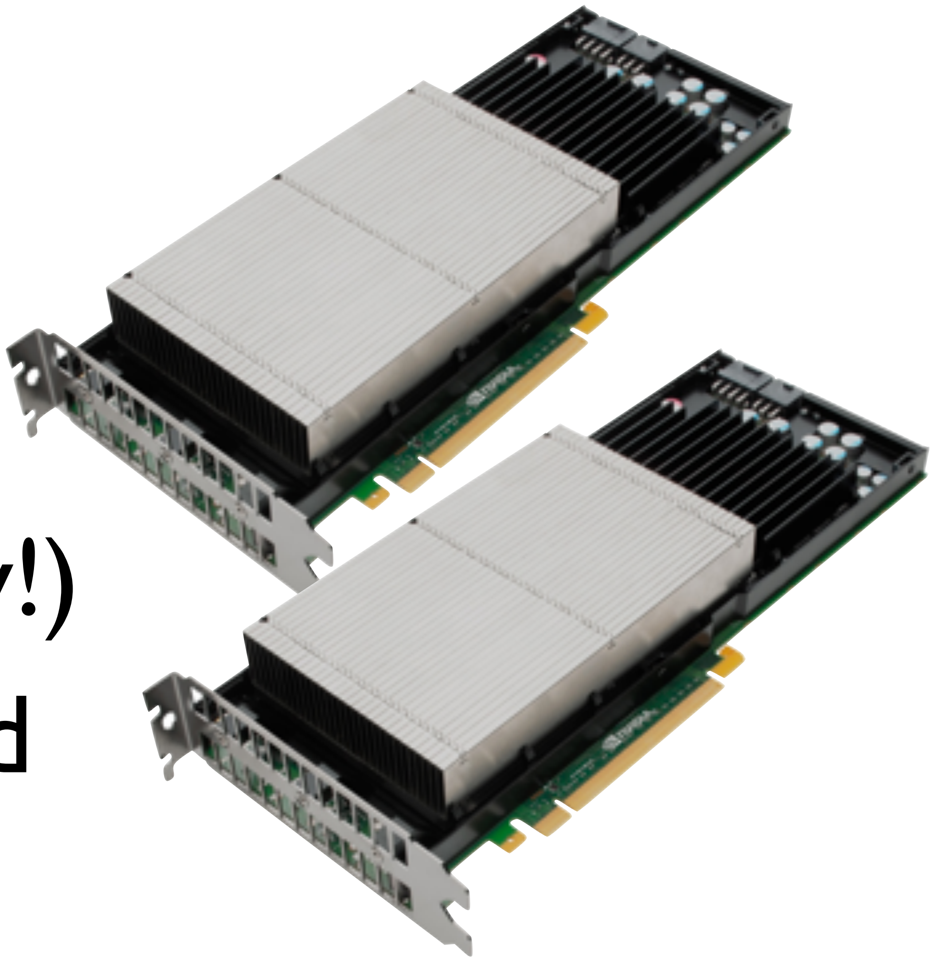
- Want memory accesses that are 32-floats wide
- Want many blocks (latency)
- Want few blocks (shared mem/node: occupancy)
- Not clear a priori what best choices are; have to experiment (and different hardware can shift balance.)
- For sticky problems, people have often already found good solutions:
- http://www.cs.colostate.edu/~cs675/MatrixTranspose.pdf

CITA|ICAT

SciNet

Multiple GPUS

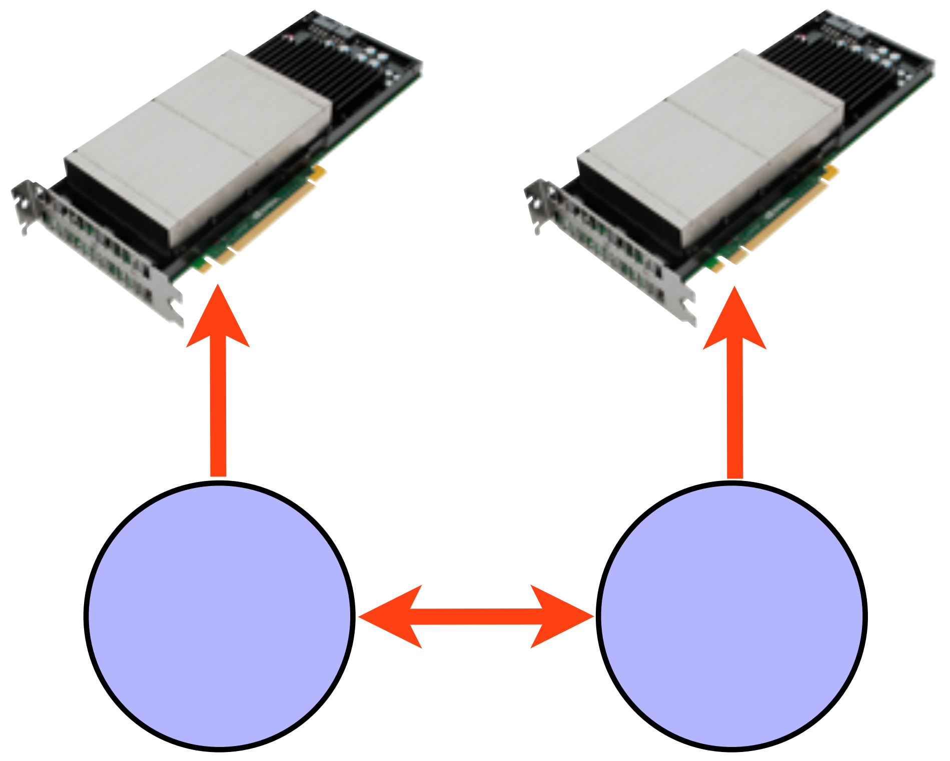
- Might want to use multiple GPUs in a node for:
  - More **cores** (you have enough work; but occupancy!)
  - More **memory** (~6GB/card isn't a tonne)
  - More memory **bandwidth** between global + cores



CITA|ICAT

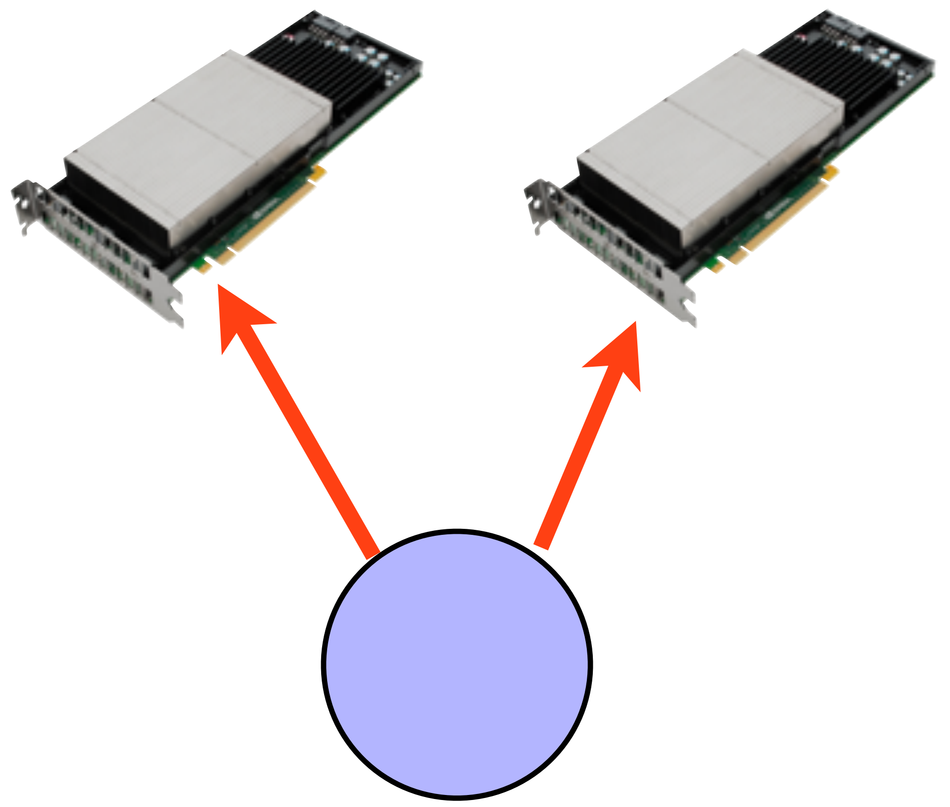
Multiple GPUS

- If you have an existing parallel program, relatively easy
  - Each thread/process "drives" its own GPU
  - Communicate amongst themselves as before
  - MPI programs lend themselves particularly well to this



Multiple GPUS

- Multi-GPU machines becoming more common
- Haven't talked about CPU parallelism in this class
- Will talk about driving multiple GPUs from a single CPU program



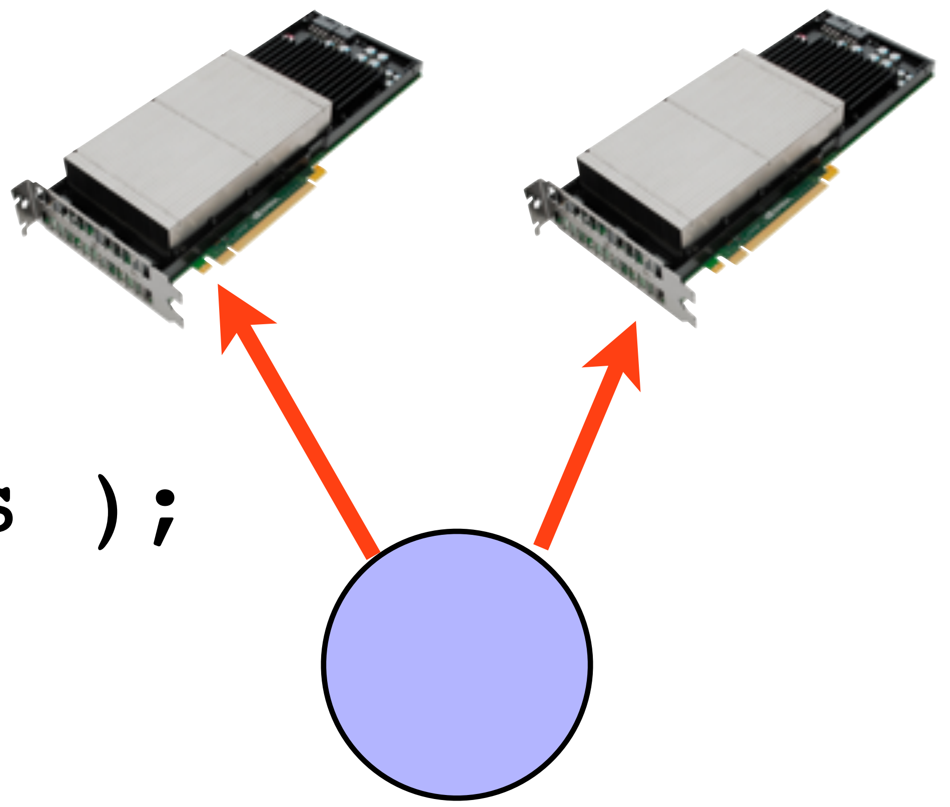
Very simple since CUDA 4.0

- All you really need to know is:

```
int ngpus;
cudaGetDeviceCount( &ngpus );
```

- and

```
cudaSetDevice(gpuNum);
```

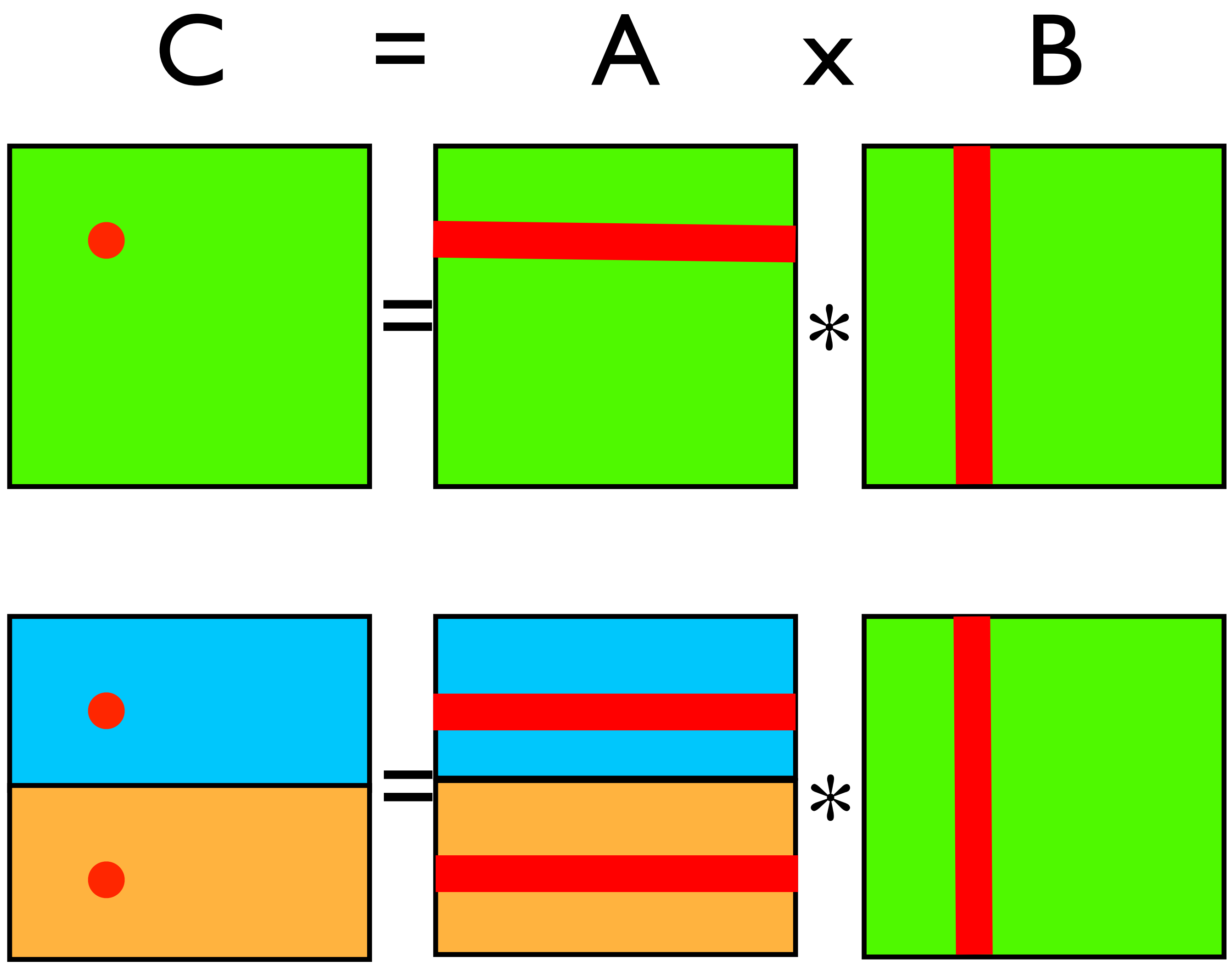


CITA|ICAT

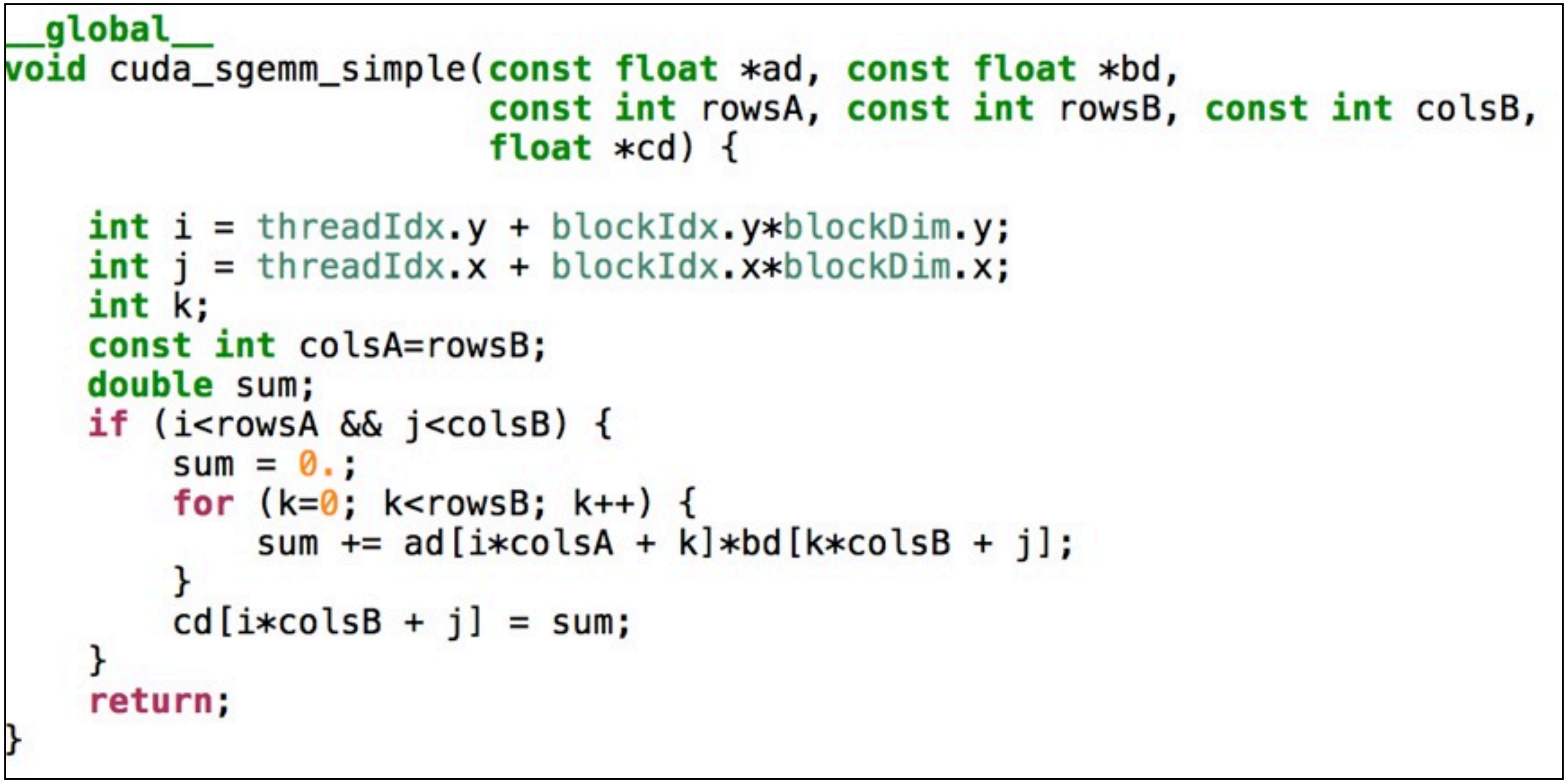
SciNet

Very simple since CUDA 4.0

- Consider our simple GPU matrix multiply
- Note that we can recast this as a block-matrix multiply, with C and A both broken down by rows
- Each chunk of A, C can be assigned to a different GPU



matmult.cu



```
__global__
void cuda_sgemm_simple(const float *ad, const float *bd,
                       const int rowsA, const int rowsB, const int colsB,
                       float *cd) {

    int i = threadIdx.y + blockIdx.y*blockDim.y;
    int j = threadIdx.x + blockIdx.x*blockDim.x;
    int k;
    const int colsA=rowsB;
    double sum;
    if (i<rowsA && j<colsB) {
        sum = 0.;
        for (k=0; k<rowsB; k++) {
            sum += ad[i*colsA + k]*bd[k*colsB + j];
        }
        cd[i*colsB + j] = sum;
    }
    return;
}
```

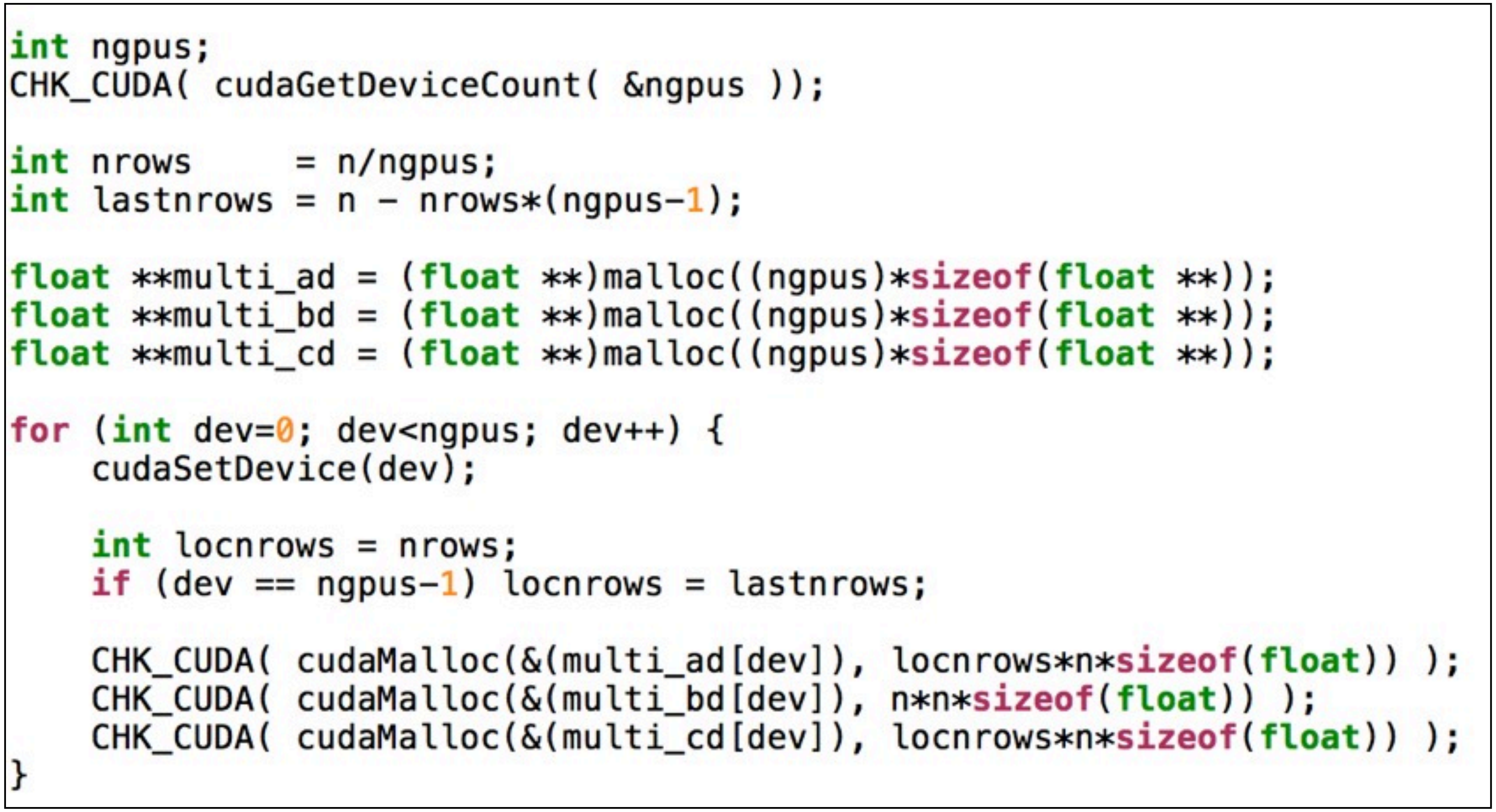
matmult.cu

```
/* run GPU code */
CHK_CUDA( cudaMalloc(&ad, n*n*sizeof(float)) );
CHK_CUDA( cudaMalloc(&bd, n*n*sizeof(float)) );
CHK_CUDA( cudaMalloc(&cd, n*n*sizeof(float)) );

tick(&gputimer);
CHK_CUDA( cudaMemcpy(ad, a, n*n*sizeof(float), cudaMemcpyHostToDevice) );
CHK_CUDA( cudaMemcpy(bd, b, n*n*sizeof(float), cudaMemcpyHostToDevice) );

blocksize = make_uint3( (n+nblocks-1)/nblocks, (n+nblocks-1)/nblocks, 1);
gridsize  = make_uint3( nblocks, nblocks, 1);

cuda_sgemm_simple<<<gridsize, blocksize>>>(ad, bd, n, n, n, cd);
CUDA_ERRCHECK();

CHK_CUDA( cudaMemcpy(ccuda, cd, n*n*sizeof(float), cudaMemcpyDeviceToHost) );
gputime = tock(&gputimer);

CHK_CUDA( cudaFree(ad) );
CHK_CUDA( cudaFree(bd) );
CHK_CUDA( cudaFree(cd) );
```

CITA|ICAT

SciNet

Matmult-multi.cu



```
int ngpus;
CHK_CUDA( cudaGetDeviceCount( &ngpus ));

int nrows     = n/ngpus;
int lastnrows = n - nrows*(ngpus-1);

float **multi_ad = (float **)malloc((ngpus)*sizeof(float **));
float **multi_bd = (float **)malloc((ngpus)*sizeof(float **));
float **multi_cd = (float **)malloc((ngpus)*sizeof(float **));

for (int dev=0; dev<ngpus; dev++) {
    cudaSetDevice(dev);

    int locnrows = nrows;
    if (dev == ngpus-1) locnrows = lastnrows;

    CHK_CUDA( cudaMalloc(&(multi_ad[dev]), locnrows*n*sizeof(float)) );
    CHK_CUDA( cudaMalloc(&(multi_bd[dev]), n*n*sizeof(float)) );
    CHK_CUDA( cudaMalloc(&(multi_cd[dev]), locnrows*n*sizeof(float)) );
}
```

Matmult-multi.cu

```
for (int dev=0; dev<ngpus; dev++) {
    cudaSetDevice(dev);

    int locnrows = nrows;
    if (dev == ngpus-1) locnrows = lastnrows;

    CHK_CUDA( cudaMemcpy(multi_ad[dev], &(a[n*(nrows*dev)]),
                locnrows*n*sizeof(float), cudaMemcpyHostToDevice) );

    CHK_CUDA( cudaMemcpy(multi_bd[dev], b,
                n*n*sizeof(float), cudaMemcpyHostToDevice) );
}
```

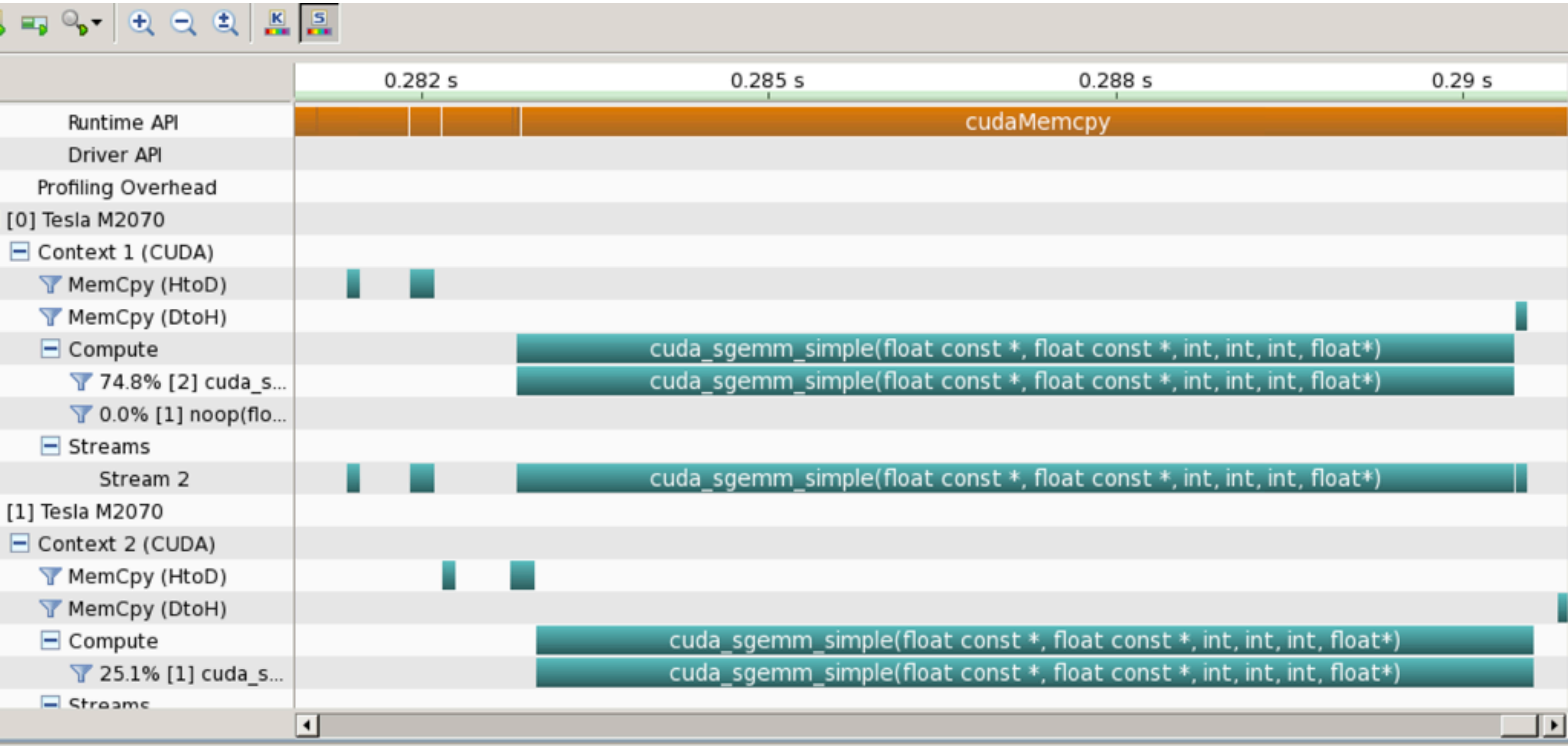
Matmult-multi.cu

```
for (int dev=0; dev<ngpus; dev++) {
    cudaSetDevice(dev);

    int locnrows = nrows;
    if (dev == ngpus-1) locnrows = lastnrows;

    int gridrows = locnrows/blocksize.y;
    int gridcols = n/blocksize.x;
    gridsize  = make_uint3( gridcols, gridrows, 1);

    cuda_sgemm_simple<<<gridsize, blocksize>>>(multi_ad[dev], multi_bd[dev],
                                               locnrows, n, n, multi_cd[dev]);
    CUDA_ERRCHECK();
}
```

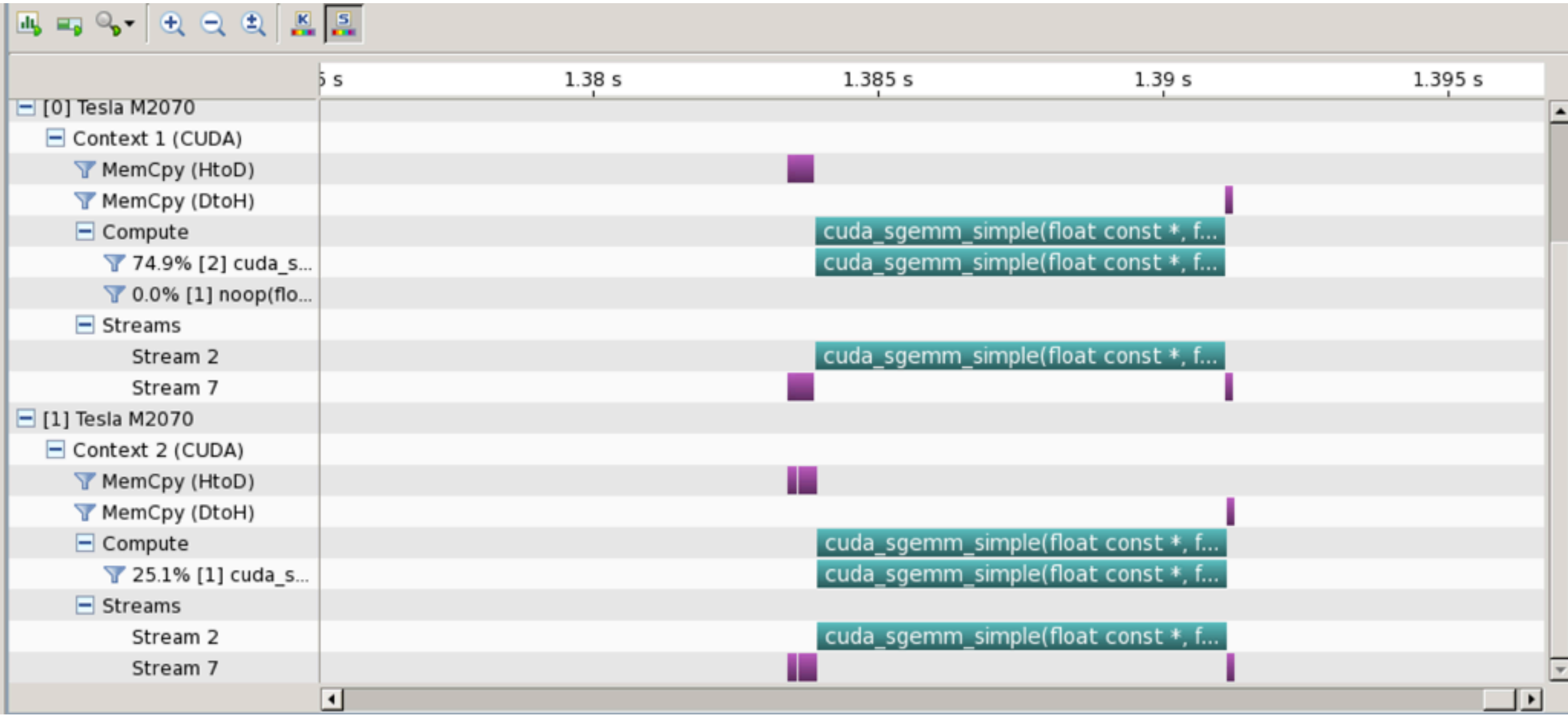
Matmult-multi.cu

```
for (int dev=0; dev<ngpus; dev++) {
    cudaSetDevice(dev);

    int locnrows = nrows;
    if (dev == ngpus-1) locnrows = lastnrows;

    CHK_CUDA( cudaMemcpy(&(ccuda[nrows*dev*n]), multi_cd[dev],
                        locnrows*n*sizeof(float), cudaMemcpyDeviceToHost) );
}

gputime = tock(&gputimer);

for (int dev=0; dev<ngpus; dev++) {
    cudaSetDevice(dev);

    CHK_CUDA( cudaFree(multi_ad[dev]) );
    CHK_CUDA( cudaFree(multi_bd[dev]) );
    CHK_CUDA( cudaFree(multi_cd[dev]) );
}
```

matmult-multi.cu


0.282 s
0.285 s
0.288 s
0.29 s
Runtime API
cudaMemcpy
Driver API
Profiling Overhead
[0] Tesla M2070
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
Compute
cuda_sgemm_simple(float const *, float const *, int, int, int, float*)
74.8% [2] cuda_s...
cuda_sgemm_simple(float const *, float const *, int, int, int, float*)
0.0% [1] noop(flo...
Streams
Stream 2
cuda_sgemm_simple(float const *, float const *, int, int, int, float*)
[1] Tesla M2070
Context 2 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
Compute
cuda_sgemm_simple(float const *, float const *, int, int, int, float*)
25.1% [1] cuda_s...
cuda_sgemm_simple(float const *, float const *, int, int, int, float*)
Streams

matmult-stream.cu



CITA|ICAT

SciNet

Blocking vs Async

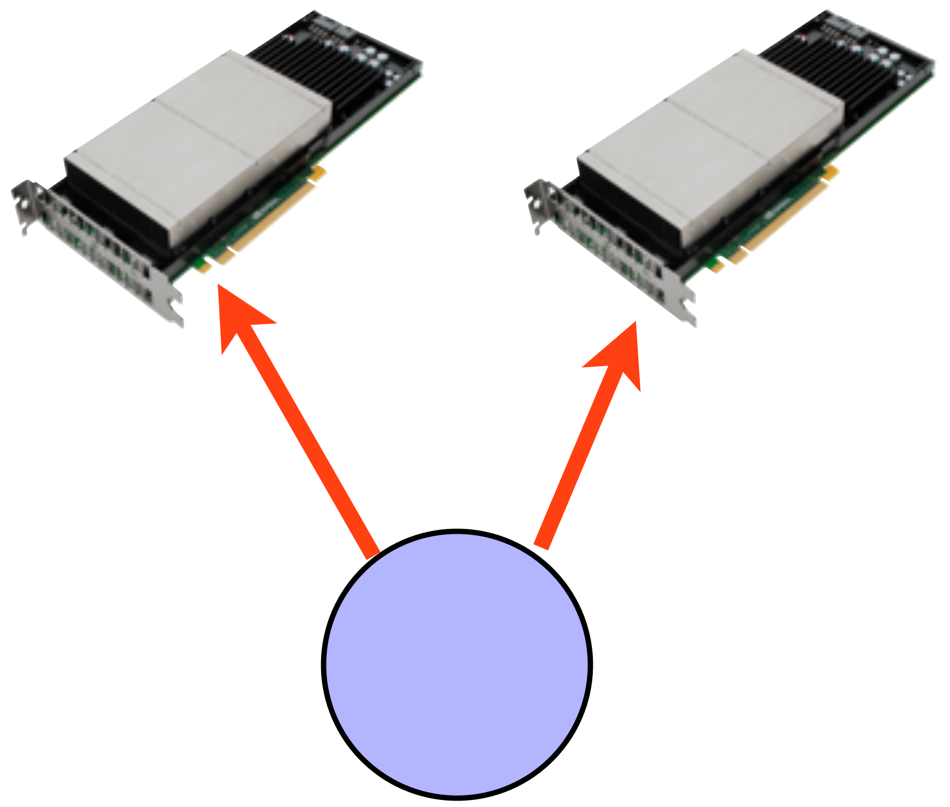
```
arc01-$ ./matmult-multi --matsize=512 --nblocks=32
Matrix size = 512, Number of blocks = 32.
CPU time = 1087.42 millisec.
GPU time = 15.894 millisec (1 gpu).
Single GPU and CPU results differ by 0.000000
GPU time = 9.163 millisec (2 gpu).
2-GPU and CPU results differ by 0.000000

arc01-$ ./matmult-stream --matsize=512 --nblocks=32
Matrix size = 512, Number of blocks = 32.
CPU a = 1087.06 millisec.
GPU time = 14.755 millisec (1 gpu).
Single GPU and CPU results differ by 0.000000
GPU time = 7.925 millisec (2 gpu).
2-GPU and CPU results differ by 0.000000
```

CITA|ICAT

SciNet

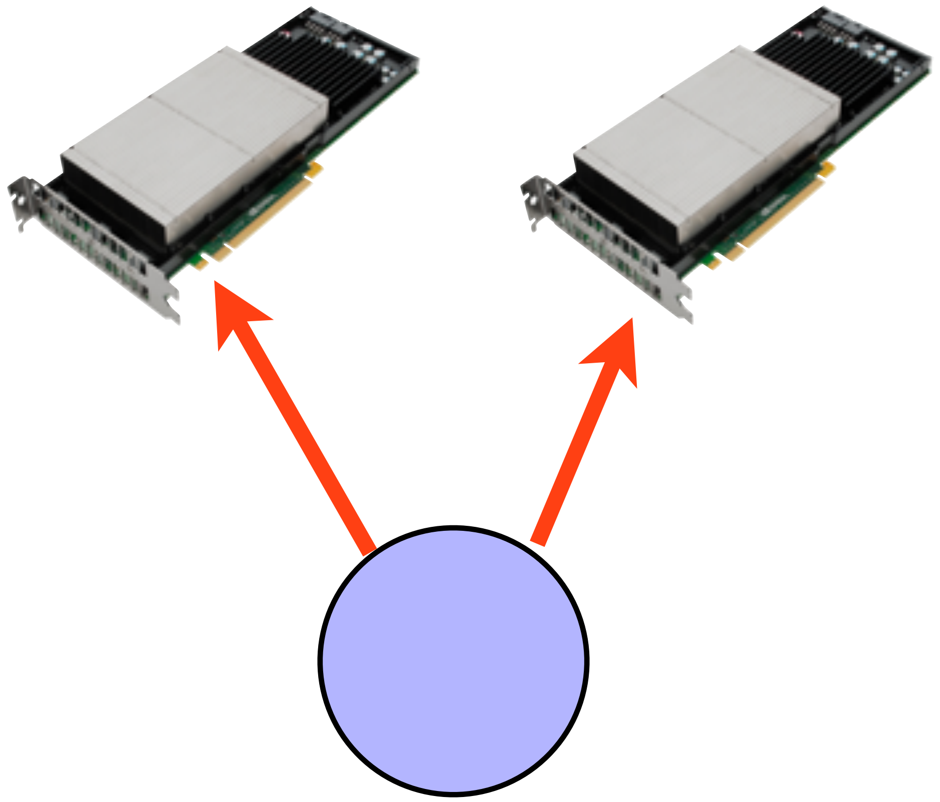
Memory exchange

- In general, don't just fire off one kernel and then done
- Iterative methods, timesteps, etc.
- Dealing with multiple GPUs introduces more complex CPU/GPU memory issues
  - More bookkeeping
  - Efficiency (waiting for CPU to copy data out of one GPU, into another)
- CUDA 4,5 introduce nicer ways to deal with this



Memory exchange

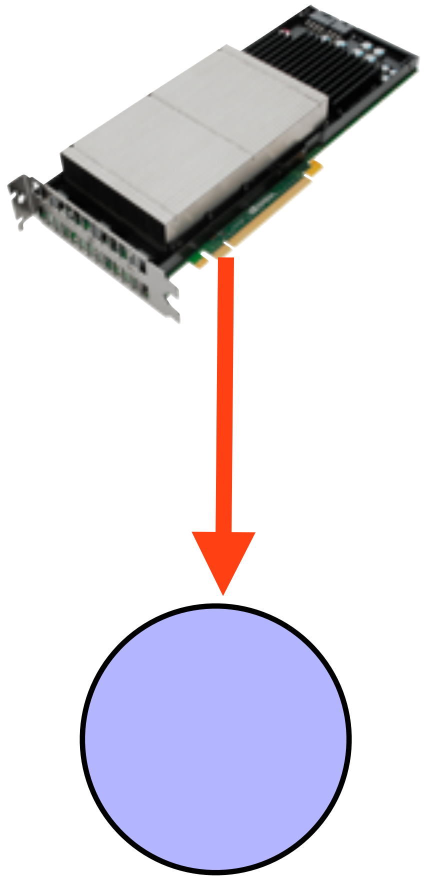
- Options:
  - Explicit CPU copying
  - Zero-copy from host (bookkeeping, maybe lower efficiency)
  - Peer-to-peer transfers
  - Peer-to-peer memory access



CITA|ICAT

Zero (explicit) copy

- GPU can "see" region of host memory
- As with async memory copy, and for same reasons, must be pinned memory, allocated with cudaHostAlloc (or registered)
- Access through slow PCIe bus
- Host/global like global/shared; fine if only going to access once, slow if you need to access it repeatedly.



Zero Copy

Let's use this for (say) B

```
    cudaHostAlloc(&a, n*n*sizeof(float), cudaHostAllocDefault);
    cudaHostAlloc(&b, n*n*sizeof(float), cudaHostAllocMapped |
                                            cudaHostAllocPortable );
    cudaHostAlloc(&c, n*n*sizeof(float), cudaHostAllocDefault);
    cudaHostAlloc(&ccuda, n*n*sizeof(float), cudaHostAllocDefault);
...
    for (int dev=0; dev<ngpus; dev++) {
        cudaSetDevice(dev);

        ...

        CHK_CUDA( cudaMalloc(&(multi_ad[dev]), locnrows*n*sizeof(float)) );
        CHK_CUDA( cudaHostGetDevicePointer(&(multi_bd[dev]), b, 0) );
        CHK_CUDA( cudaMalloc(&(multi_cd[dev]), locnrows*n*sizeof(float)) );
    }
...
      cuda_sgemm_simple<<<gridsize, blocksize>>>(multi_ad[dev], multi_bd[dev], locnrows,
n, n, multi_cd[dev]);
```

CITA|ICAT

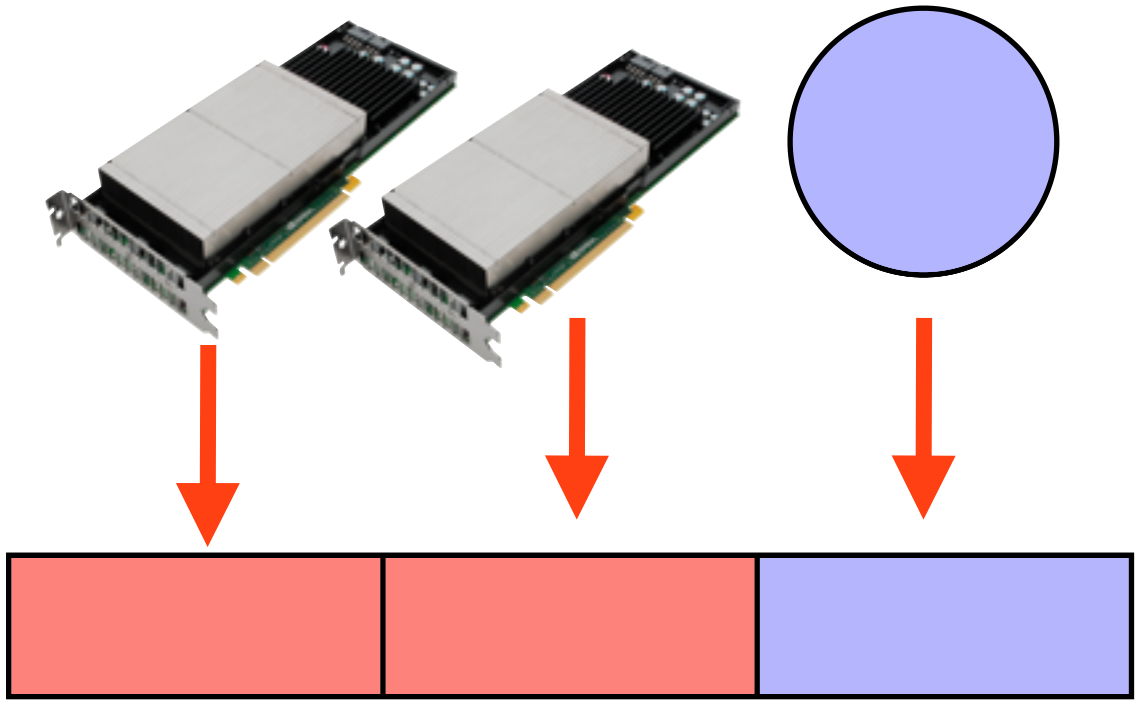
Zero Copy

Slower, but less explicit copying, memory usage on GPU

```
arc01-$ ./matmult-zero --matsize=512 --nblocks=128
Matrix size = 512, Number of blocks = 128.
CPU time = 1086.17 millisec.
GPU time = 78.694 millisec (1 gpu).
Single GPU and CPU results differ by 0.000000
GPU time = 82.103 millisec (2 gpu).
2-GPU and CPU results differ by 0.000000
```

Unified Virtual Adressing

- Tesla Fermis and later
- CUDA 4.0 and later
- From programmer's point of view, *one* global memory space
- Can just use cudaMemcpy for everything
- Mainly convenience



CITA|ICAT

Direct Peer Copes

- With UVA, can enable copies directly from GPU to GPUfirst enable peer access for each peer pair:
- cudaDeviceEnablePeerAccess(int peerDevice, unsigned int flags)
- flags always zero (for now)
- Give appropriate device ptr

Homework

- Goal: repeatedly smooth the large image from previous homework (models a PDE calculation) on multiple GPUs.
  - Starting from smoothimage HW, just using global memory for image, loop over the memcopy/kernel/memcopy with 1 gpu
  - Then split over multiple gpus, as with simple matmult
  - Then enable peer copying, and have kernels start with copy from neighbour.
  - How to synchronize?