Blocks, Grids, and Shared Memory

GPU Course, Fall 2012

CITA|ICAT

SciNet

Last week: ax+b Homework

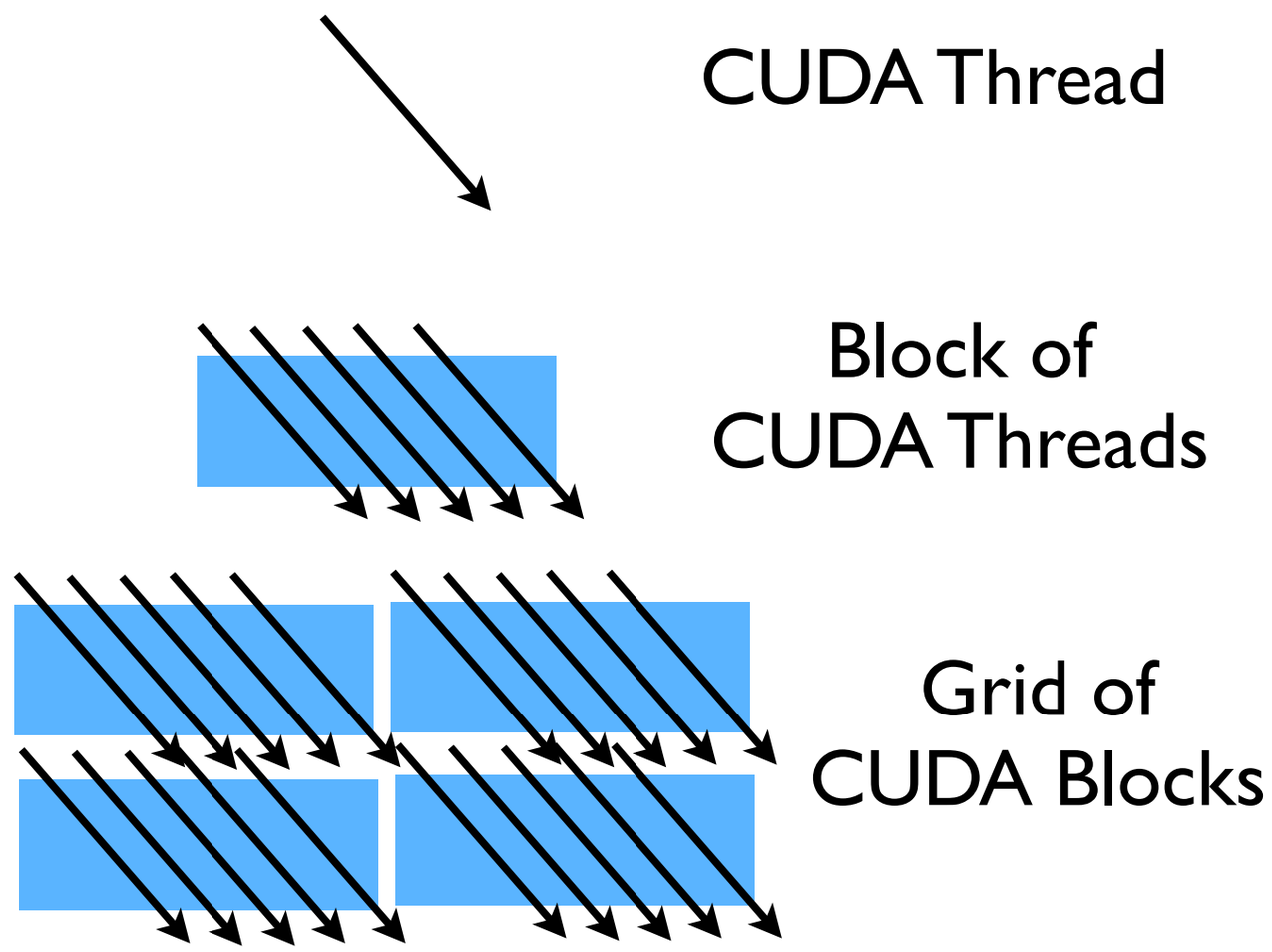
```
__global__ void MultAddKernel(const int N, double* vD, const double* uD,
                              const double a, const double b) {
  int idx=threadIdx.x;
  vD[idx] = a*uD[idx]+b;
};



// (3) compute on device
MultAddKernel<<<1, N>>>(N, vD, uD, a, b);
```

CITA|ICAT

SciNet

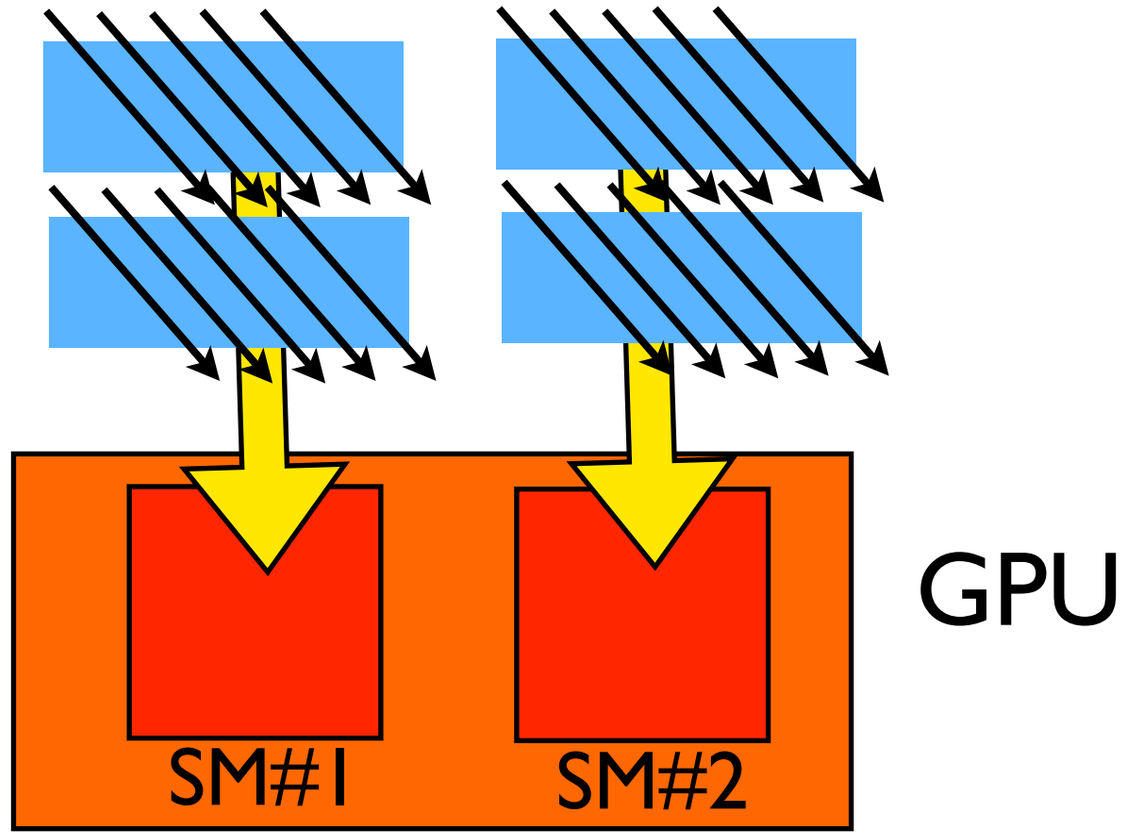
Threads, Blocks, Grids

- CUDA threads are organized into blocks
- Threads operate in SIMD(ish) manner -- each executing same instructions in lockstep.
- Only difference are thread ids
- Can have a grid of multiple blocks



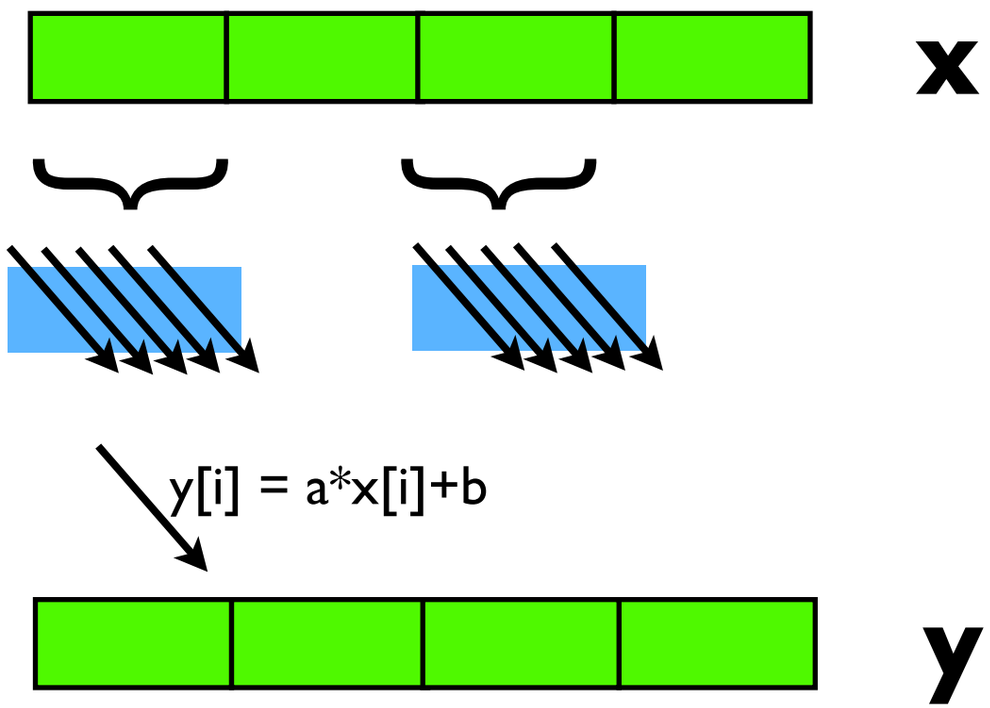
CUDA - H/W mapping

- Blocks are assigned to a particular SM
  - Executed there one ‘warp’ at a time (typically 32 threads)
- Multiple blocks may be on SM concurrently
  - Good; latency hiding
  - Bad - SM resources must be divided between blocks
- If only use 1 Block - 1 SM



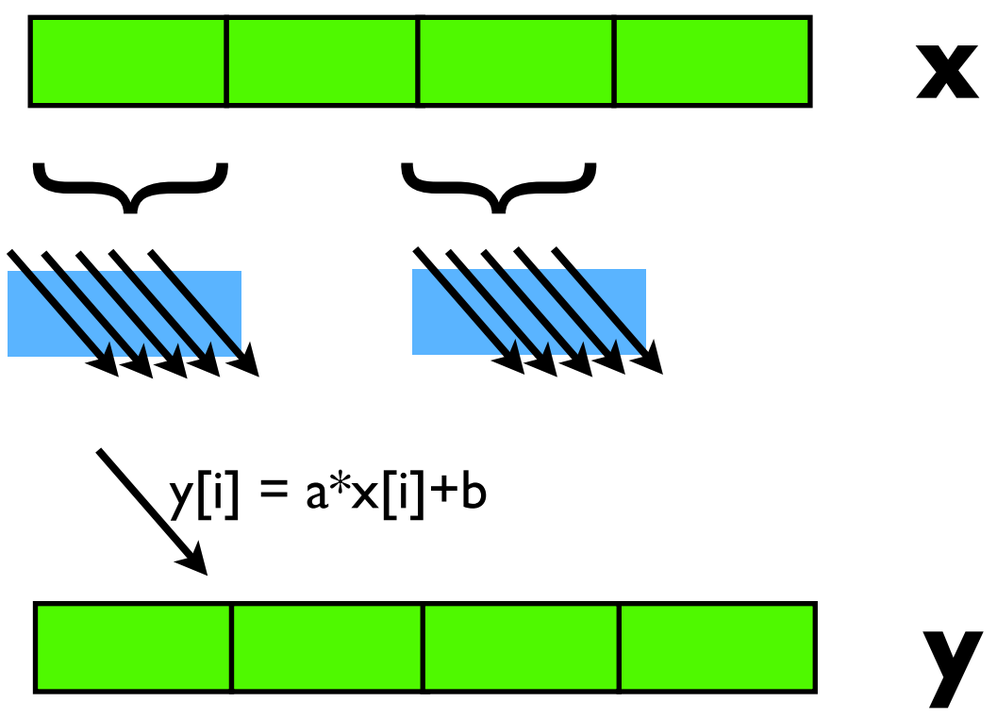
Multi-block y=ax+b

- Break input, output vectors into blocks
- Within each block, thread index specifies which item to work on
- Each thread does one update, puts results in y[i]



Multi-block y=ax+b

```
__global__ void cuda_saxpb(const float *xd,
                           const float a,
                           const float b,
                           float *yd,
                           const int n) {

    int i = threadIdx.x + blockIdx.x*blockDim.x;
    if (i<n) {
        yd[i] = a*xd[i]+b;
    }
    return;
}
```




```
get_options(argc, argv, &n, &nblocks, &a, &b);

...

blocksize = (n+nblocks-1)/nblocks;
cuda_saxpb<<<nblocks, blocksize>>>(xd, a, b, yd, n);
```

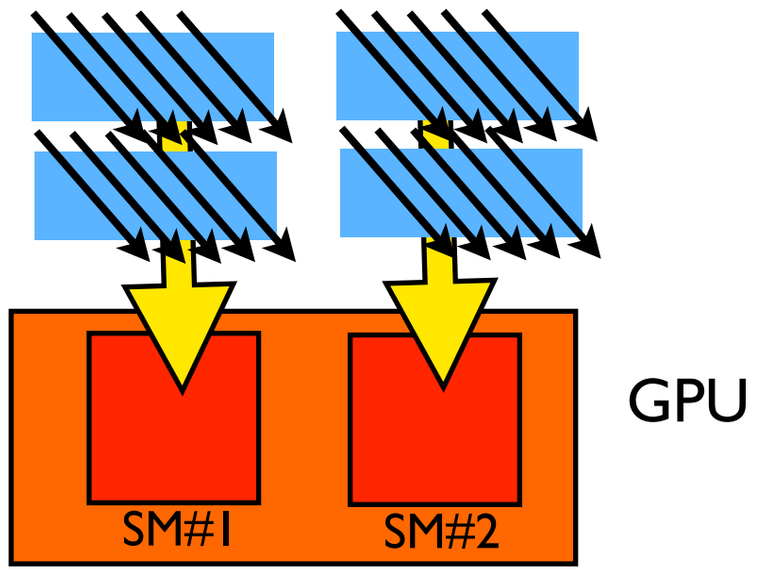
block-saxpb.cu

More blocks → more SMs → more FLOPs

- On newer cards, where we can use 1024 threads/ block:

Multiple calcs, so timing not dominated by memory copy

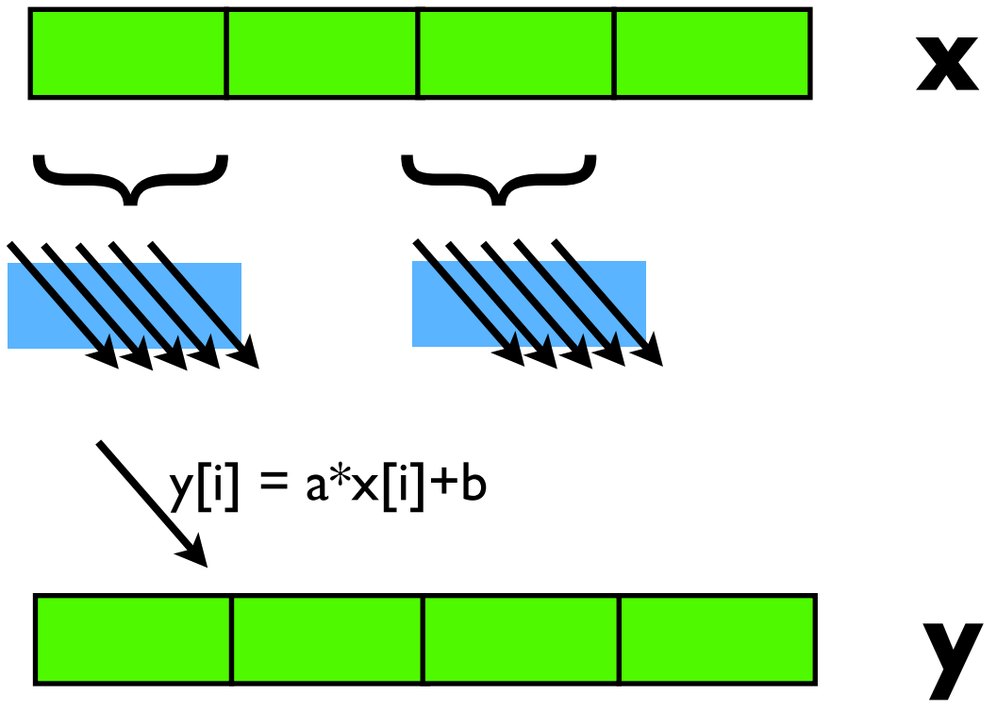
```
$ ./block-saxpb --nblocks=1 --nvals=1024 --niters=100
CPU time = 0.455 millisec.
GPU time = 0.511 millisec.
CUDA and CPU results differ by 0.000000
$
$ ./block-saxpb --nblocks=8 --nvals=8192 --niters=100
CPU time = 3.62 millisec.
GPU time = 0.546 millisec.
CUDA and CPU results differ by 0.000000
```



Multi-block y=ax+b

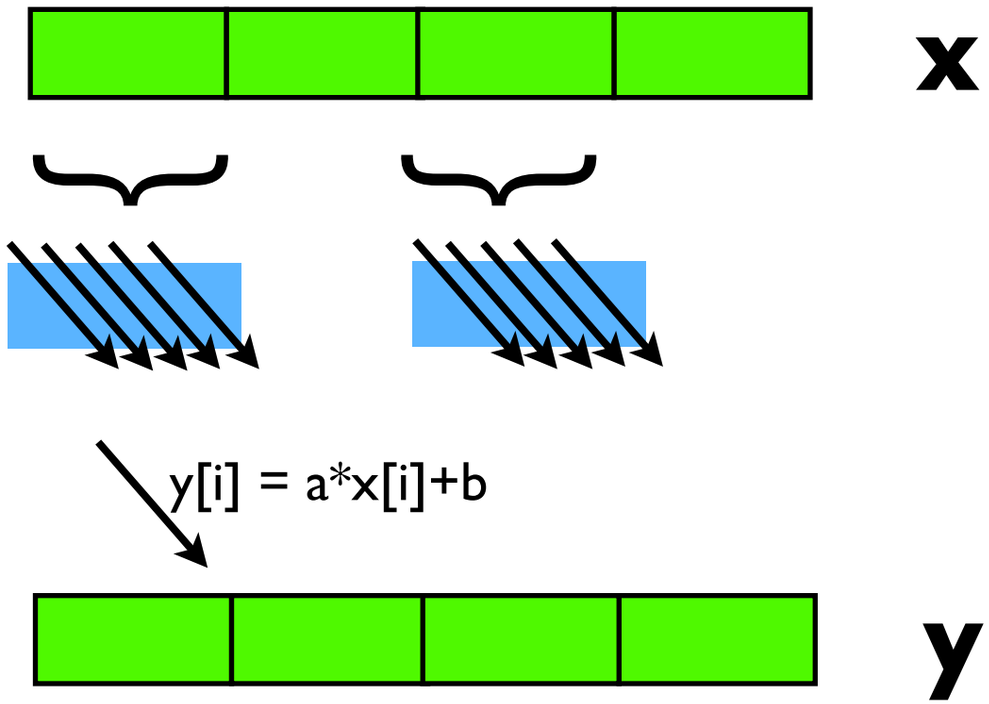
```
__global__ void cuda_saxpb(const float *xd,
                           const float a,
                           const float b,
                           float *yd,
                           const int n) {

    int i = threadIdx.x + blockIdx.x*blockDim.x;
    if (i<n) {
        yd[i] = a*xd[i]+b;
    }
    return;
}
```

Index *within* block
(0..blocksize-1)



Multi-block y=ax+b

```
__global__ void cuda_saxpb(const float *xd,
                           const float a,
                           const float b,
                           float *yd,
                           const int n) {

    int i = threadIdx.x + blockIdx.x*blockDim.x;
    if (i<n) {
        yd[i] = a*xd[i]+b;
    }
    return;
}
```

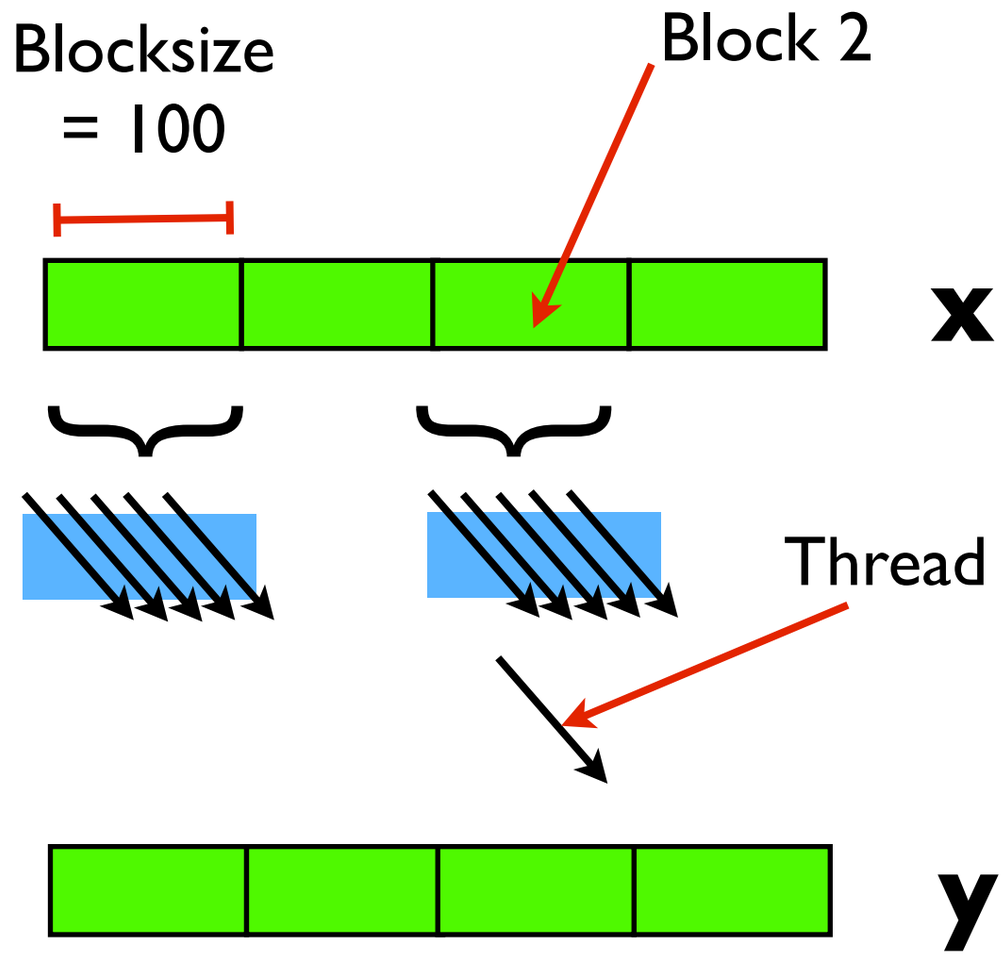


Index *of* block
(0..nblocks-1)

Size of block
(blocksize)

Multi-block y=ax+b

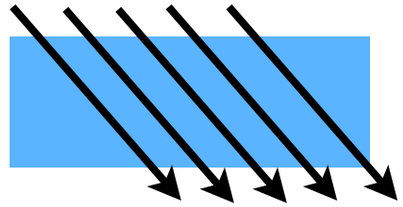
```
__global__ void cuda_saxpb(const float *xd,
                           const float a,
                           const float b,
                           float *yd,
                           const int n) {

    int i = threadIdx.x + blockIdx.x*blockDim.x;
    if (i<n) {
        yd[i] = a*xd[i]+b;
    }
    return;
}
```




```
i = 10 + 2*100   = 210
yd[210] = a*xd[210] + b
```

How many threads/ block?

- Should be integral multiple of warp (32)
- No more than max allowed by scheduling hardware
- Can get last number from hardware specs
- But what if will be needed on several machines?
- API can return it:



cudaGetDeviceProperty

```
int i, count;
cudaDeviceProp prop;

CHK_CUDA( cudaGetDeviceCount( &count ));
for (i=0; i<count; i++) {
    CHK_CUDA( cudaGetDeviceProperties( &prop, i ));
    printf("Device %d has:\n",i);
    printf("\tName                    %s,\n",prop.name);
    printf("\tNumber of SMs           %d,\n",prop.multiProcessorCount);
    printf("\tWarp Size               %d,\n",prop.warpSize);
    printf("\tMax Threads/block       %d,\n",prop.maxThreadsPerBlock);
```

querydevs.cu

cudaGetDeviceProperty

```
#define CHK_CUDA(e) {if (e != cudaSuccess) { \
                       fprintf(stderr,"Error: %s\n", cudaGetErrorString(e)); \
                       exit(-1);}\
                     }
```

All CUDA calls return cudaSuccess on successful completion.

GPU hardware does not try very hard to catch errors/notify you; testing return codes important!

Common to see simple automation like this wrapping all CUDA calls; bare minimum for sensible operation.

Test early, fail often.

Why the .xs?

- For convenience, CUDA allows thread, block indicies to be multidimensional
- Thread blocks can be 3 dimensional (512,512,64)
- Grids of blocks can be 2 dimensional (64k, 64k, 1)
- These variables are of type dim3 or uint3
- CUDA has int1, int2, int3, int4, float1, float2, float3, float4, etc.

```
__global__ void cuda_saxpb(const float *xd,
                           const float a,
                           const float b,
                           float *yd,
                           const int n) {

int i = threadIdx.x + blockIdx.x*blockDim.x;
if (i<n) {
    yd[i] = a*xd[i]+b;
}
return;
}
```

Why the .xs?

- threadIdx.{x,y,z} - thread index
- blockDim.{x,y,z} - size of block (# of threads in each dim)
- blockIdx.{x,y,z} - block index
- gridDim.{x,y,z} - size of grid (# of blocks in each dim)
- warpsize - size of warp (int)

```
__global__ void cuda_saxpb(const float *xd,
                           const float a,
                           const float b,
                           float *yd,
                           const int n) {

    int i = threadIdx.x + blockIdx.x*blockDim.x;
    if (i<n) {
        yd[i] = a*xd[i]+b;
    }
    return;
}
```

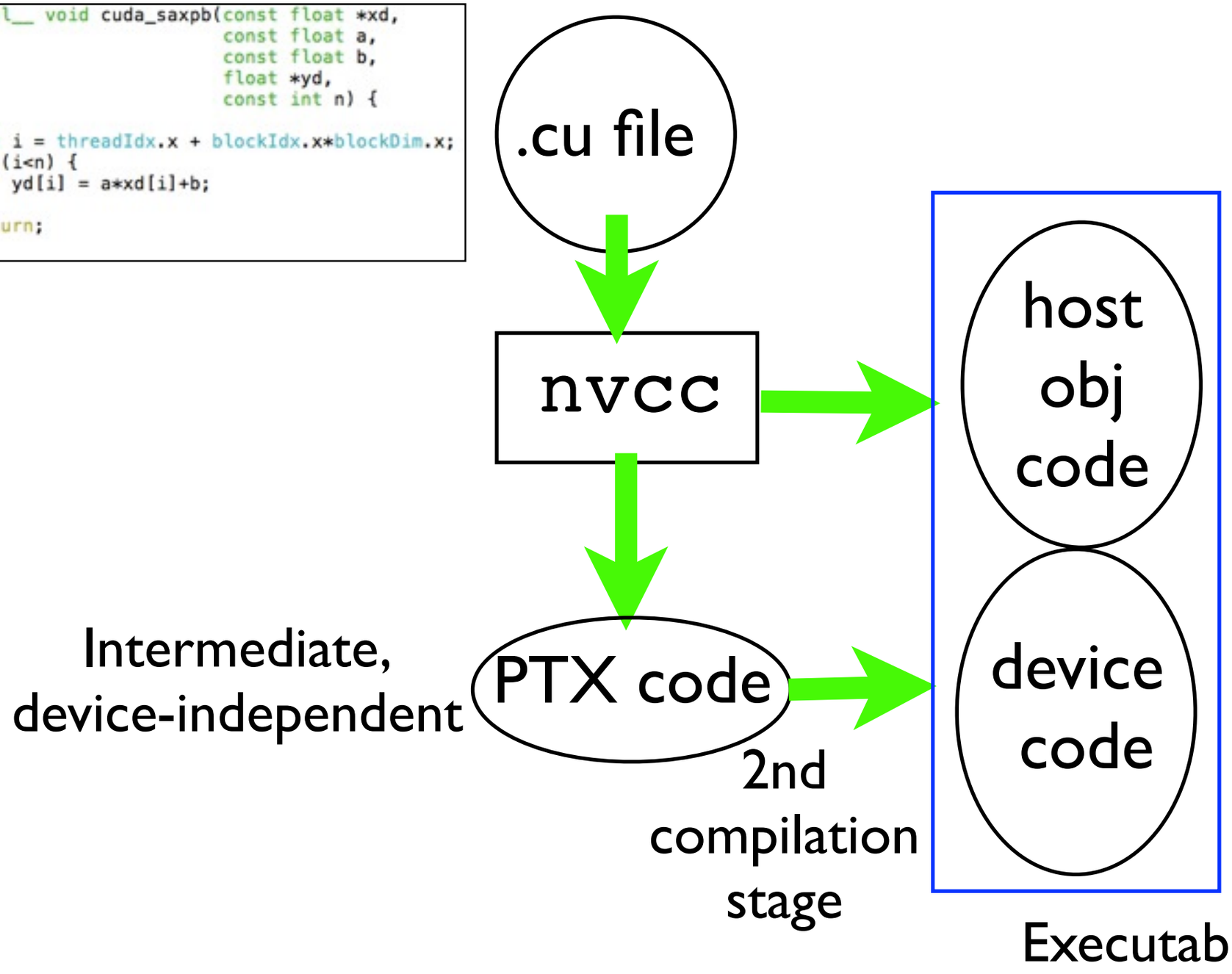
Why the .xs?

- `__global__` - device code that can be seen (invoked) from host.
- `__host__` - default. Not usually interesting.
- `__device__` - device code. Can be called only from other device code.
- `__host__ __device__` - compiled for both host and device.

```
__global__ void cuda_saxpb(const float *xd,
                           const float a,
                           const float b,
                           float *yd,
                           const int n) {

    int i = threadIdx.x + blockIdx.x*blockDim.x;
    if (i<n) {
        yd[i] = a*xd[i]+b;
    }
    return;
}
```

Compilation process

```
__global__ void cuda_saxpb(const float *xd,
                           const float a,
                           const float b,
                           float *yd,
                           const int n) {

    int i = threadIdx.x + blockIdx.x*blockDim.x;
    if (i<n) {
        yd[i] = a*xd[i]+b;
    }
    return;
}
```



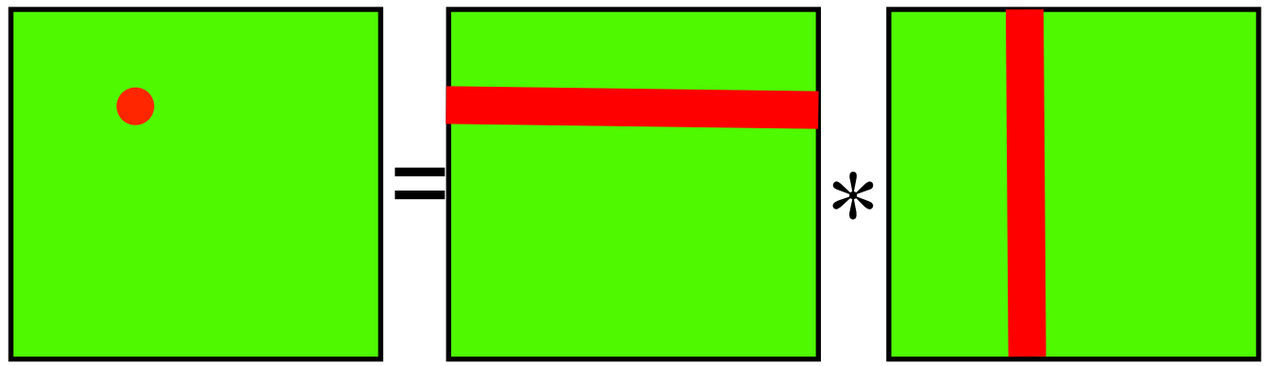
Restrictions

- __global__ functions can't recurse, neither can __device__ on non-Fermis
- No function pointers to __device__ functions on non-fermis, can't take address of __device__ function
- Can't have static variables in __global__, __device__ functions
- Can't use varargs with device code

```
__global__ void cuda_saxpb(const float *xd,
                           const float a,
                           const float b,
                           float *yd,
                           const int n) {

    int i = threadIdx.x + blockIdx.x*blockDim.x;
    if (i<n) {
        yd[i] = a*xd[i]+b;
    }
    return;
}
```

2-Dimensional Blocks

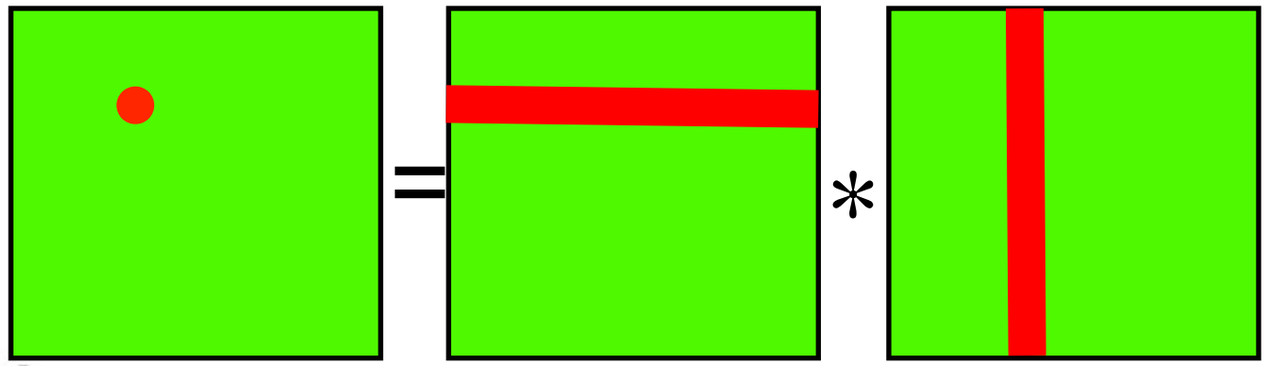
- Use of 2/3d thread blocks, or 2d grids, never strictly necessary...
  - But can make code clearer, shorter.
  - Matrix multiplication



$$C_{i,j} = \sum_k A_{i,k} B_{k,j}$$

2-Dimensional Blocks

```
void cpu_sgemm(const float *a, const float *b,
               const int n, float *c) {

    /* this, of course, is a
       terrible implementation */
    int i, j, k;
    double sum;
    for (i=0;i<n;i++) {
        for (j=0;j<n;j++) {
            sum = 0.;
            for (k=0;k<n;k++) {
                sum += a[i*n + k]*b[k*n + j];
            }
            c[i*n + j] = sum;
        }
    }
    return;
}
```



$$C_{i,j} = \sum_k A_{i,k} B_{k,j}$$

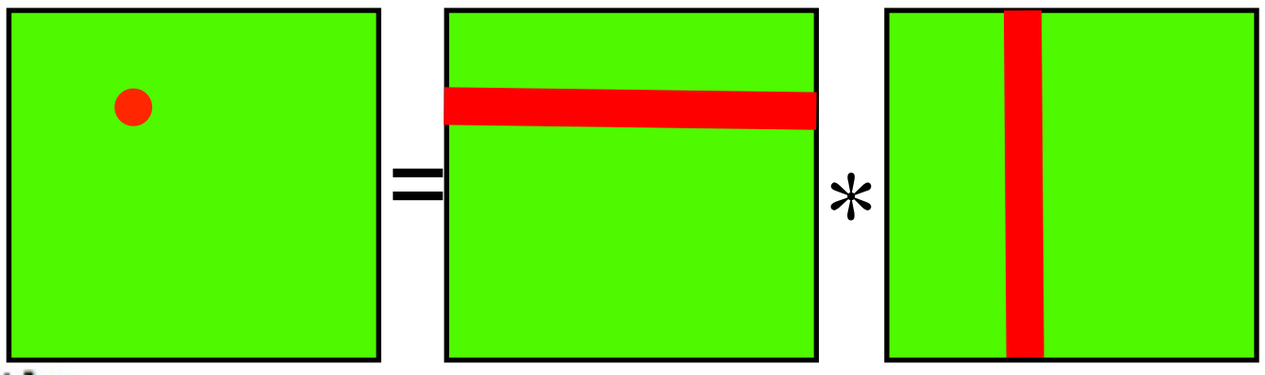
matmult.cu

CITA|ICAT

SciNet

2-Dimensional Blocks

```
__global__
void cuda_sgemm(const float *ad, const float *bd,
                const int n, float *cd) {

    int i = threadIdx.x + blockIdx.x*blockDim.x;
    int j = threadIdx.y + blockIdx.y*blockDim.y;
    int k;
    if (i<n && j<n) {
        cd[i*n + j] = 0;
        for (k=0; k<n; k++) {
            cd[i*n + j] += ad[i*n + k]*bd[k*n + j];
        }
    }
    return;
}
```



$$C_{i,j} = \sum_k A_{i,k} B_{k,j}$$

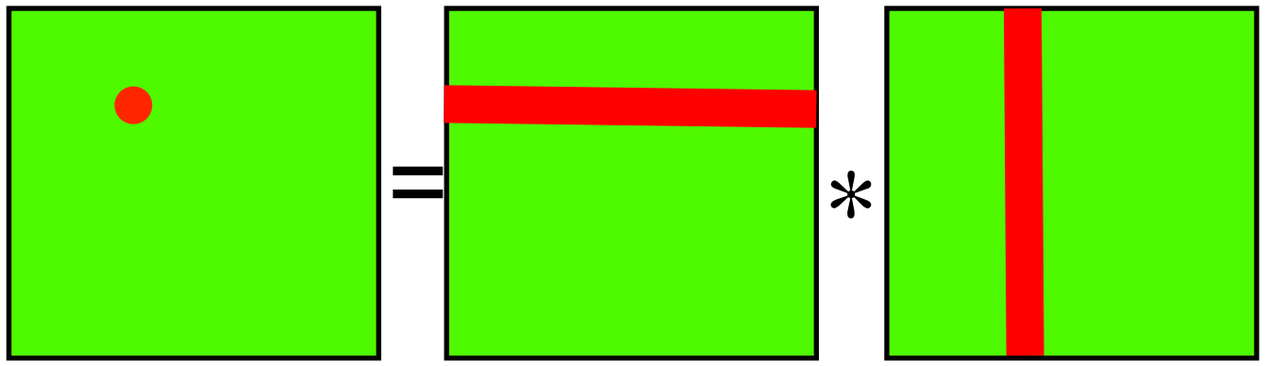
```
blocksize = make_uint3( (n+nblocks-1)/nblocks, (n+nblocks-1)/nblocks, 1);
gridsize  = make_uint3( nblocks, nblocks, 1);

cuda_sgemm<<<gridsize, blocksize>>>(ad, bd, n, cd);
```

CITA|ICAT

SciNet

2-Dimensional Blocks

```
__global__
void cuda_sgemm_reg(const float *ad, const float *bd,
                    const int n, float *cd) {

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



$$C_{i,j} = \sum_k A_{i,k} B_{k,j}$$

CITA|ICAT

SciNet

Timings:

Orig

```
$ ./matmult --matsize=160 --nblocks=10
Matrix size = 160, Number of blocks = 10.
CPU time = 14.093 millisec.
GPU time = 4.416 millisec.
CUDA and CPU results differ by 0.162872
```

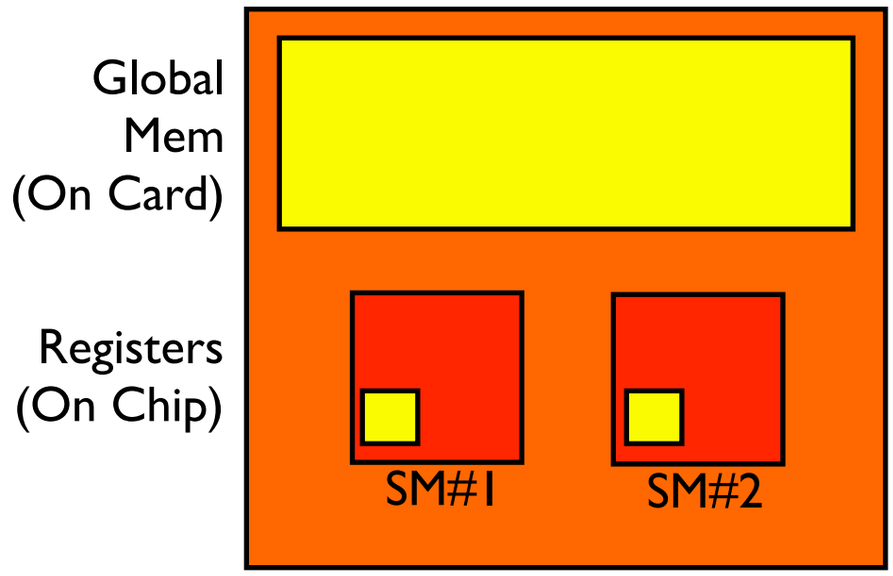
Double Prec. sum

```
$ ./matmult --matsize=160 --nblocks=10
Matrix size = 160, Number of blocks = 10.
CPU time = 14.047 millisec.
GPU time = 2.219 millisec.
CUDA and CPU results differ by 0.000000
```

Faster, even with double precision sums - why?

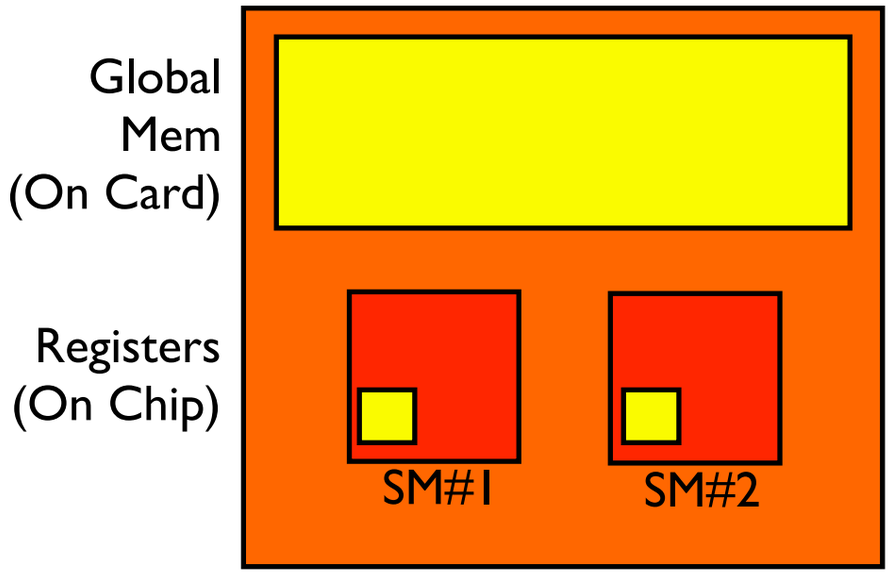
CUDA Memories

- All HPC, but especially GPU, all about planning memory access to be fast
- Global mem is off the GPU chip (but on the card); ~100 cycle latency
- Thread-local variables get put into registers on each SM - fast (~1 cycle) but small



CUDA Memories

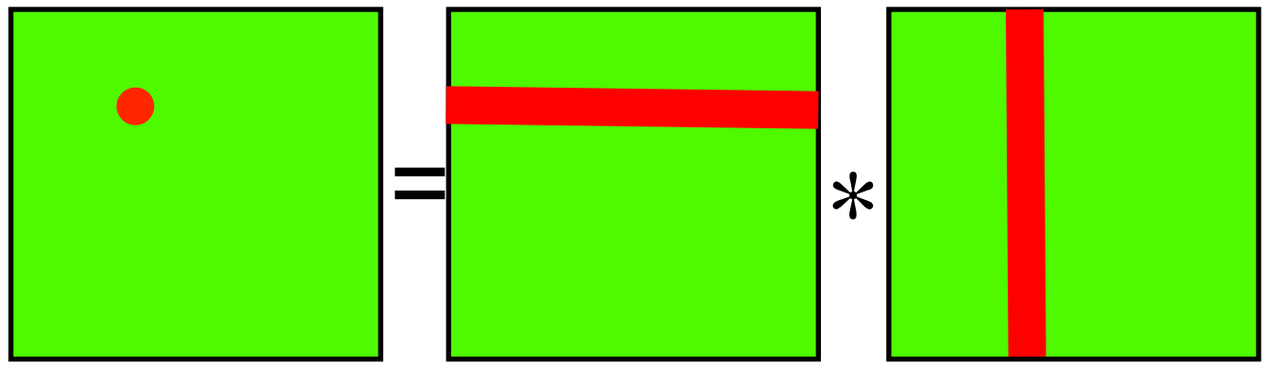
Memory	On Chip?	Cached?	R/W	Scope
Register	On	No	R/W	**Thread**
Shared	On	No	R/W	**Block**
Global	Off	No	R/W	Kernel, Host
Constant	Off	Yes	R	Kernel, Host
Texture	Off	Yes	R(W?)	Kernel, Host
'Local'*	Off	No	R/W	Thread



* if you run out of registers, will put 'local' mem in global.

Memory usage in SGEMM

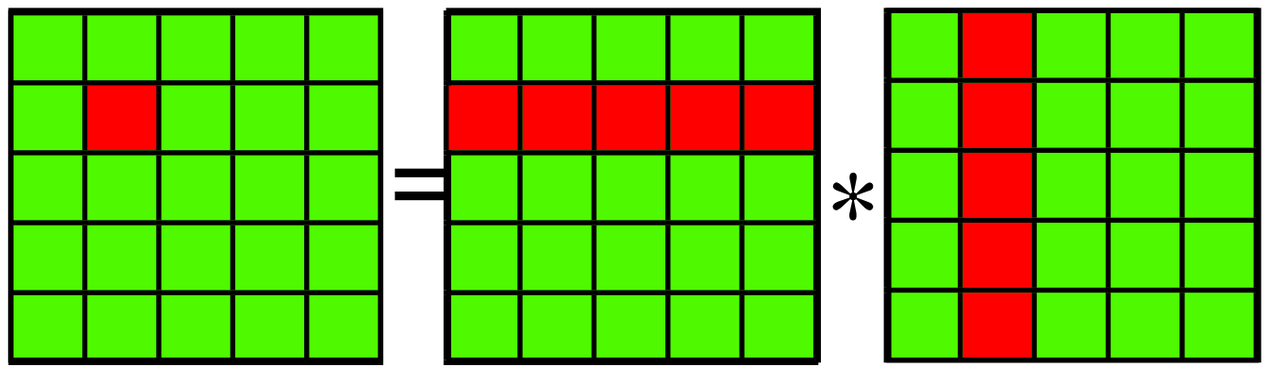
- How can we exploit this?
- N^3 multiplies, adds
- $2N^2$ data
- Regular access
- Opportunity for high memory re-use
- Need to find ways to bring data into shared memory (incurring global mem overhead once), use it several times



$$C_{i,j} = \sum_k A_{i,k} B_{k,j}$$

Memory usage in SGEMM

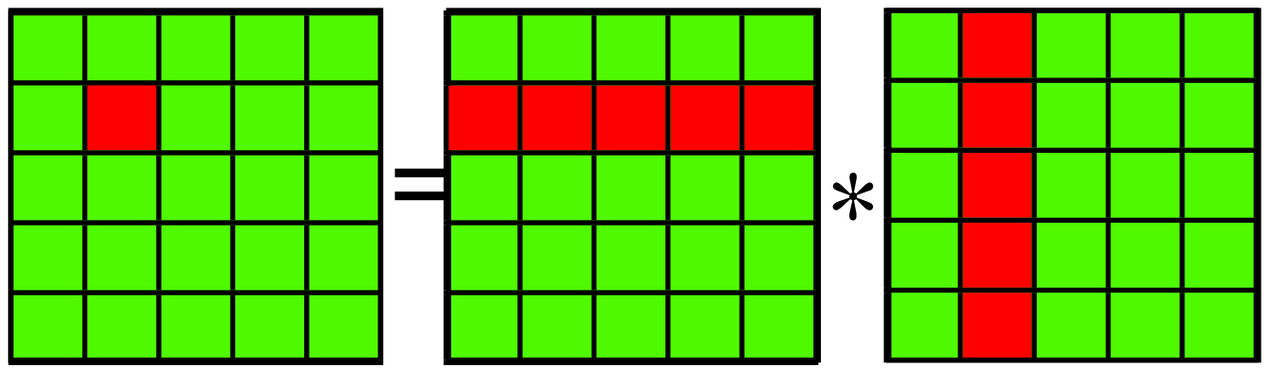
- One nice thing about matrix multiplication - same as block multiplication, each sub-block is a matrix mult
- Neighbouring threads within block all see nearby rows, columns
- Pull whole block in
- If *b* blocks in each dim, each data only pulled in *2b* times, not *2n* times



$$\mathrm{C}_{bi,bj} = \sum_k \mathrm{A}_{bi,bk}\mathrm{B}_{bk,bj}$$

Memory usage in SGEMM

```
__global__
void cuda_sgemm_shared(const float *ad, const float *bd,
                       const int n, float *cd) {

    int i = threadIdx.x + blockIdx.x*blockDim.x;
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    int locj = threadIdx.y;
    int locj = threadIdx.y;
    int locn = blockDim.x;
    __shared__ atile[TILESIZE][TILESIZE];
    __shared__ btile[TILESIZE][TILESIZE];
    //...

    double sum = 0;

    for (each tile) {
        //..load in tiles

        for (k=0; k<locn; k++) {
            sum += atile[loci*locn + k]*
                btile[k*locn + locj];
        }
    }

    c[i*n + j] = sum;
```



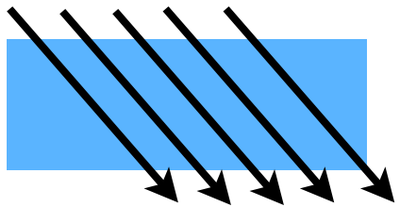
$$C_{bi,bj} = \sum_k A_{bi,bk} B_{bk,bj}$$

CITA|ICAT

SciNet

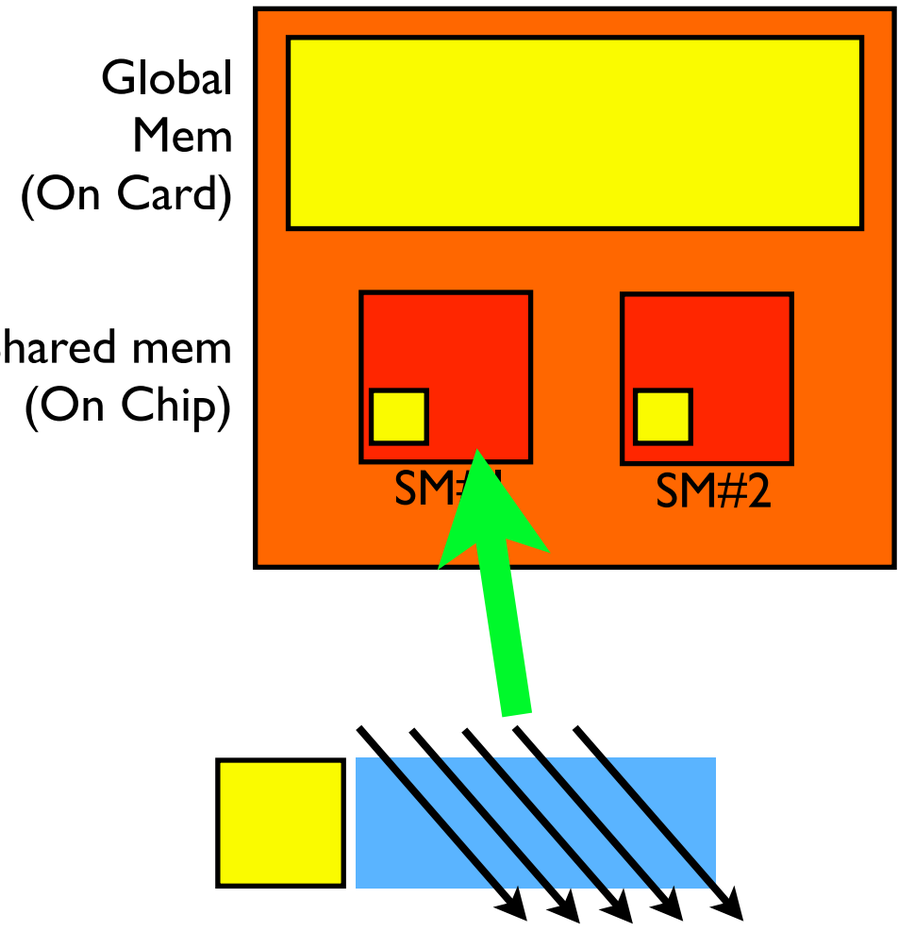
__syncthreads()

- Computation must wait until all threads have brought in their data
- Not all memory accesses may take same length of time
- __syncthreads() - waits until all threads in block are at same point.
- *No* equivalent between blocks
- Loop must similarly wait for computation



__shared__ arrays

- If declared in device code, must be sized at compile time.
- No sharedMalloc (all threads in block would have to agree)
- can use consts or #defines to size array, but we want to maintain flexibility



extern __shared__

```
__global__ void cuda_sgemm_shared(const float *ad, const float *bd,
                                  const int n, float *cd) {

    extern __shared__ float shared_data[];

    //...

    float *atile = &(shared_data[0]);
    float *btile = &(shared_data[tilesize*tilesize]);


void main() {
    //..
    cuda_sgemm_shared<<<gridsize, blocksize,
              (2*blocksize.x*blocksize.y*sizeof(float))>>>(ad, bd, n, cd);
```

Optional 3rd argument - size (in bytes) of shared memory to allocate per block

extern __shared__

```
__global__ void cuda_sgemm_shared(const float *ad, const float *bd,
                                  const int n, float *cd) {

    extern __shared__ float shared_data[];

    //...

    float *atile = &(shared_data[0]);
    float *btile = &(shared_data[tilesize*tilesize]);


void main() {
    //..
    cuda_sgemm_shared<<<gridsize, blocksize,
                (2*blocksize.x*blocksize.y*sizeof(float))>>>(ad, bd, n, cd);
```

Comes in as *one* array; can type, name it anything you like

extern __shared__

```
__global__ void cuda_sgemm_shared(const float *ad, const float *bd,
                                  const int n, float *cd) {

    extern __shared__ float shared_data[];

    //...

    float *atile = &(shared_data[0]);
    float *btile = &(shared_data[tilesize*tilesize]);


void main() {
    //..
    cuda_sgemm_shared<<<gridsize, blocksize,
                (2*blocksize.x*blocksize.y*sizeof(float))>>>(ad, bd, n, cd);
```

If you want to use it for 2 things, you have to deal with that yourself.

Timings (tpb2):

Orig

```
$ ./matmult --matsize=160 --nblocks=10
Matrix size = 160, Number of blocks = 10.
CPU time = 14.093 millisec.
GPU time = 4.416 millisec.
CUDA and CPU results differ by 0.162872
```

Double Prec. sum

```
$ ./matmult --matsize=160 --nblocks=10
Matrix size = 160, Number of blocks = 10.
CPU time = 14.047 millisec.
GPU time = 2.219 millisec.
CUDA and CPU results differ by 0.000000
```

Shared

```
$/matmult
Matrix size = 160, Number of blocks = 10.
CPU time = 14.041 millisec.
GPU time = 0.998 millisec.
CUDA and CPU results differ by 0.000000
```

Homework

- Using matmult.cu as a template, look at smoothimage.c in the code I'll send out; implements image convolution.
- Implement a CUDA version using shared memory, and make sure it gets same answer as CPU version.
- How does data reuse vary as a function of the halo size?