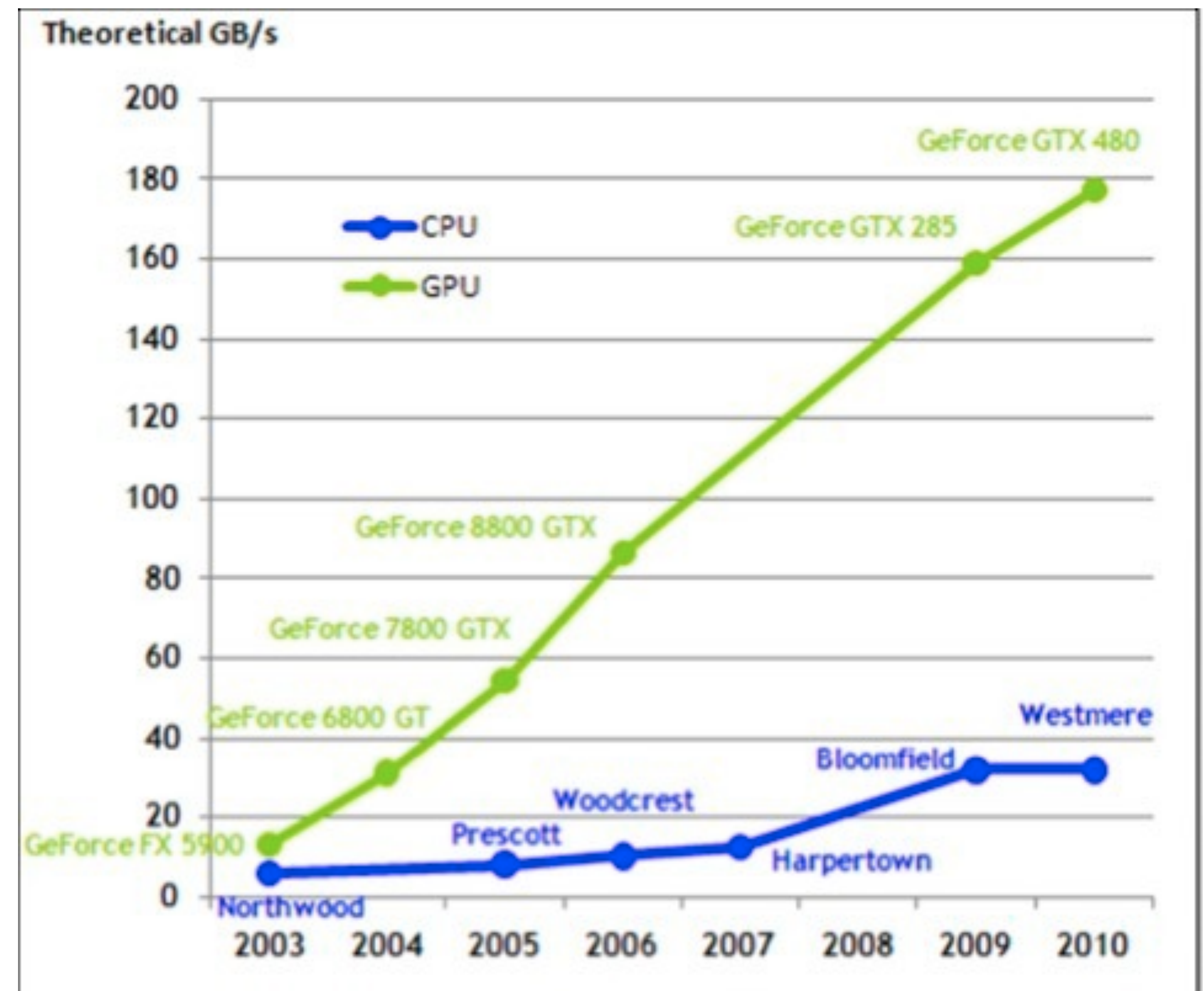
Intro to GPGPU

SciNet

Your graphics card is probably faster than your computer.

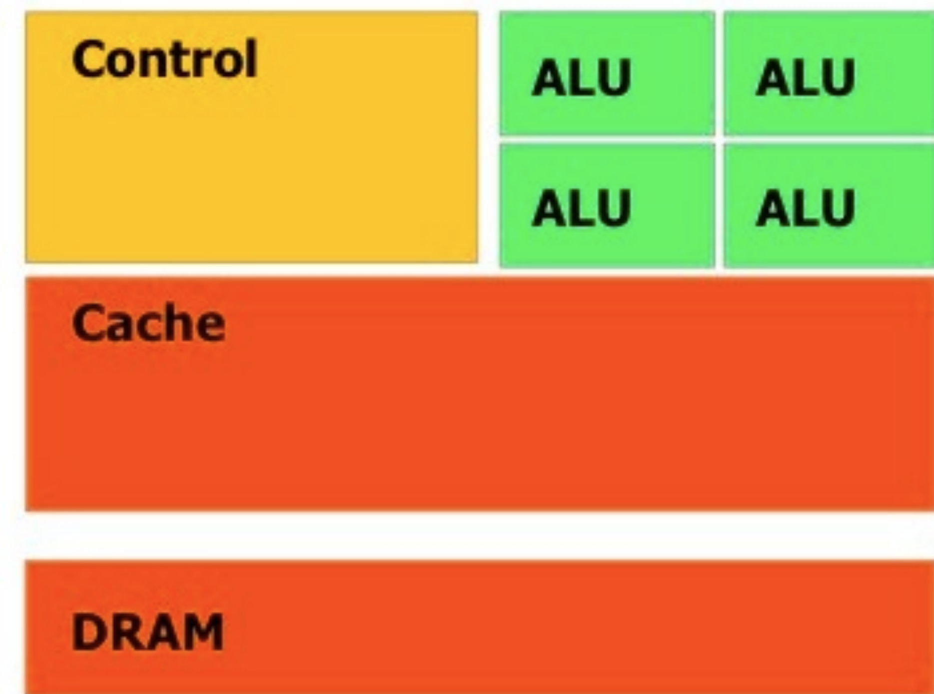
- Graphics performance has grown by leaps and bounds
- Driven by gamers

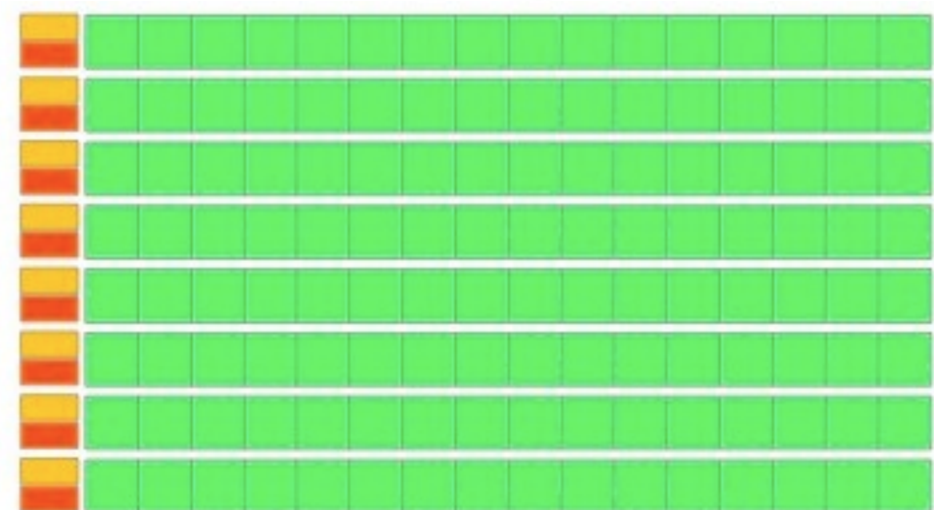


SciNet

...but it's not magic

- CPU - very flexible, easy to program
- GPU - almost all transistors go to cores and mathematics.

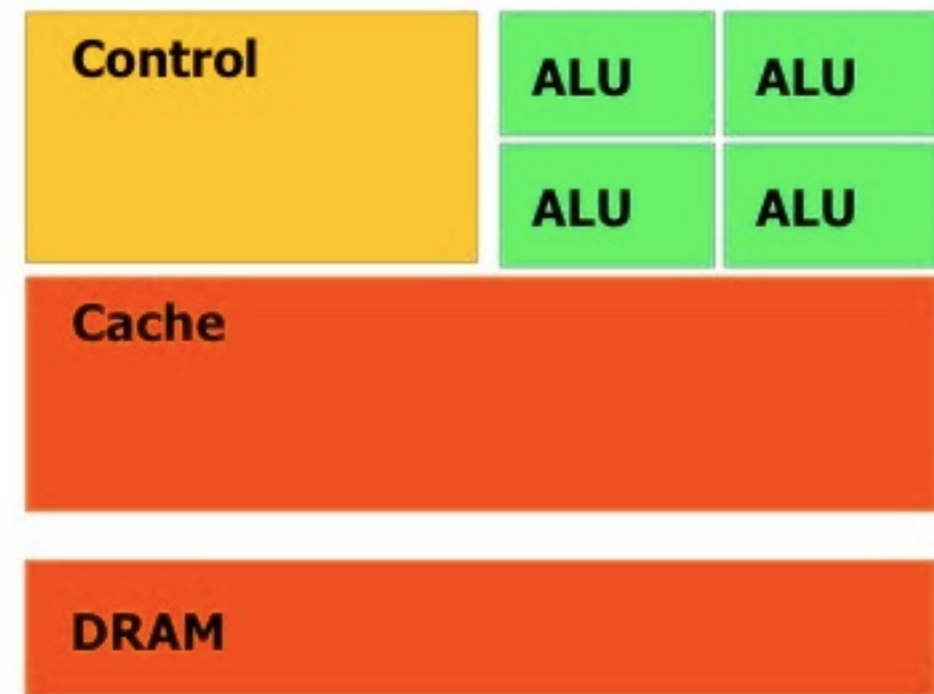


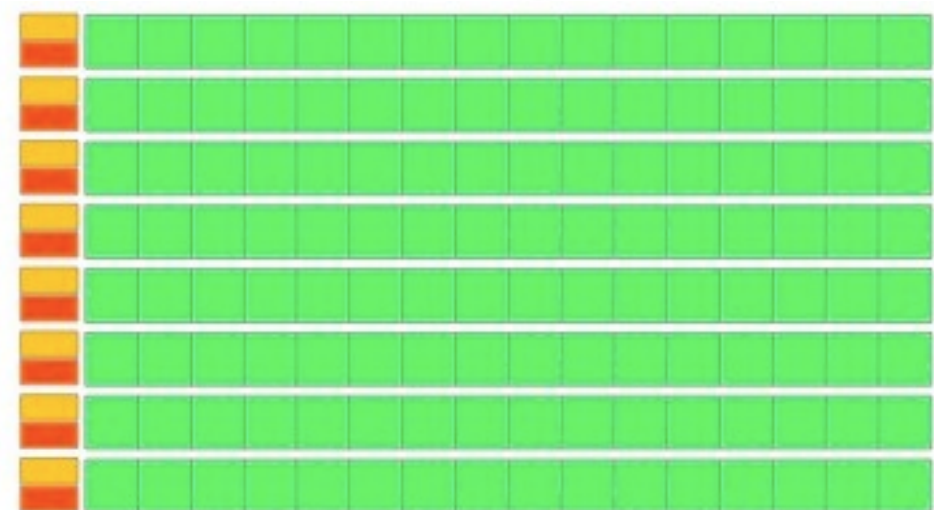


iNet

...but it's not magic

- All cores in a "multiprocessor unit" have same control, cache
- Act in lock step
- Do same computations on different data
- "Data parallel"

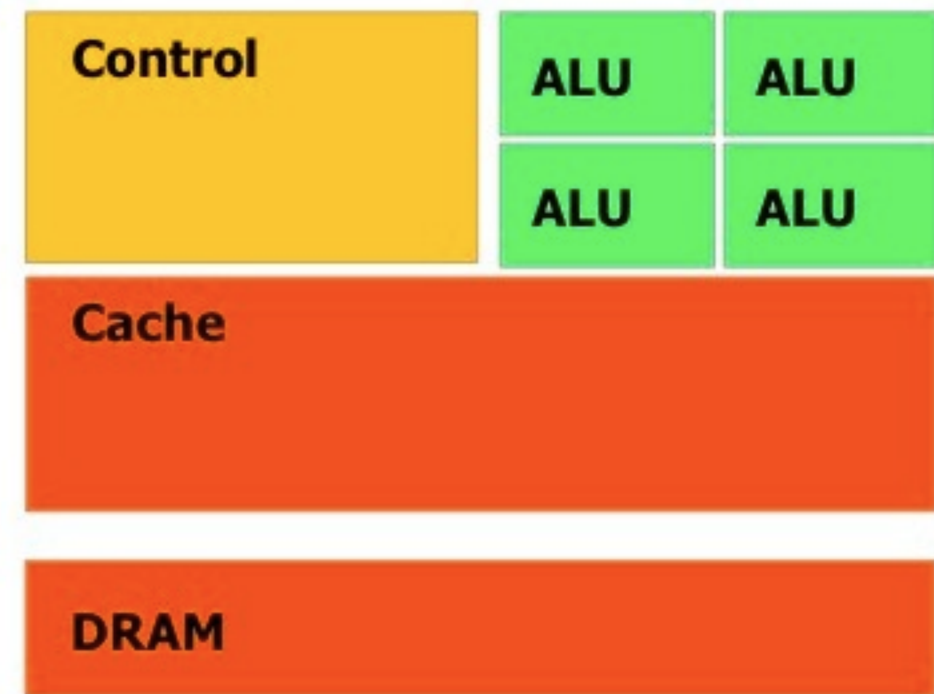


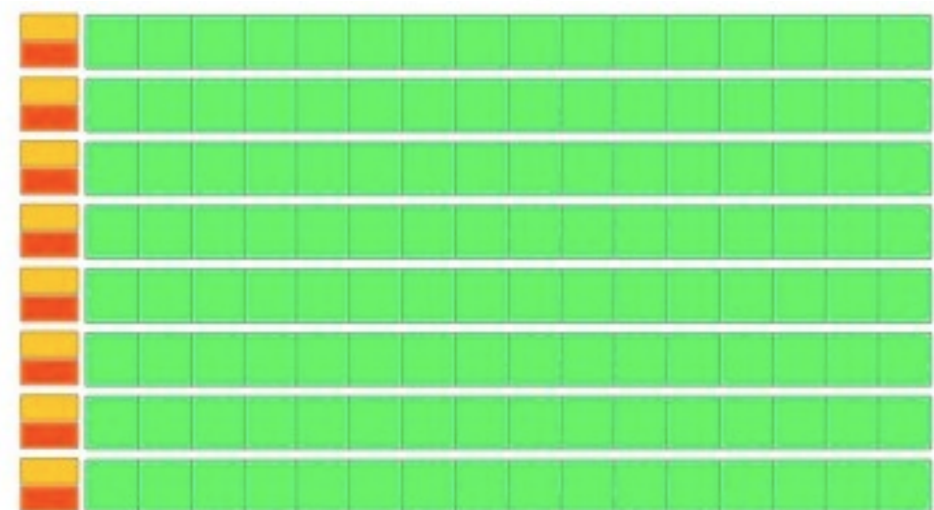


iNet

...but it's not magic

- All cores in a "multiprocessor unit" have same control, cache
- Act in lock step
- Do same computations on different data
- "Data parallel"

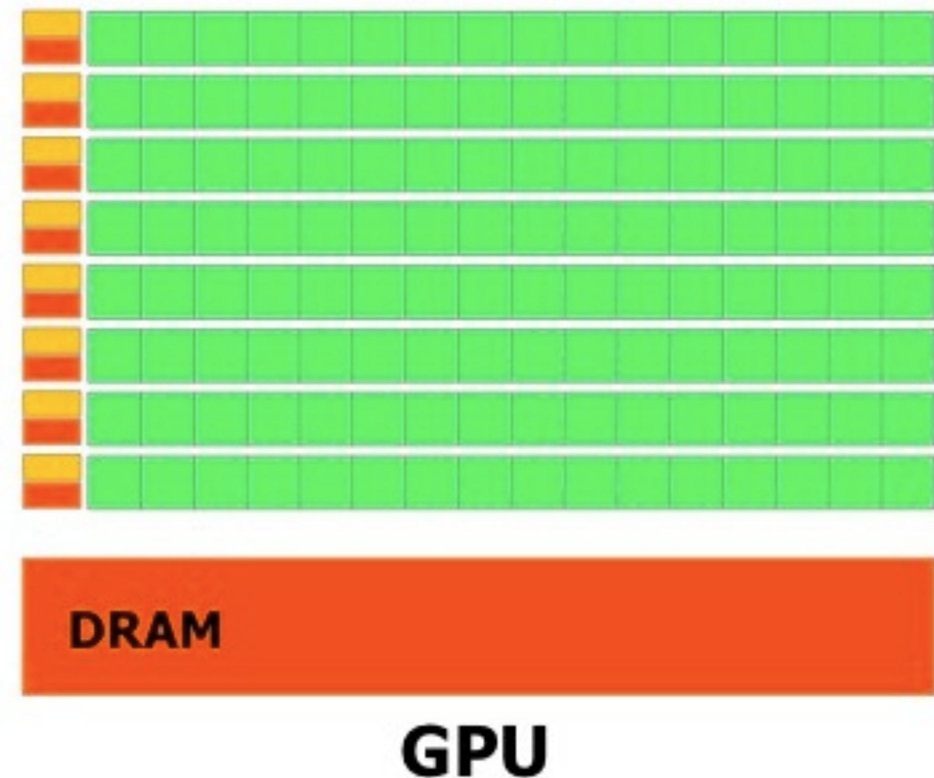




iNet

If it works, it's great..

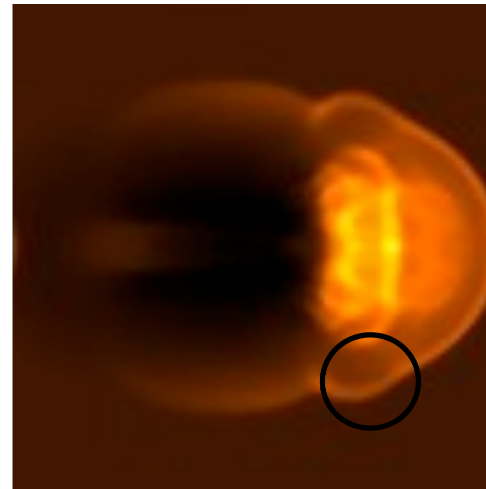
- GPU: ~448 compute cores, into ~14 streaming multiprocessors (SM)
- ~32 threads operate at once
- Very small cache (48KB/SM)

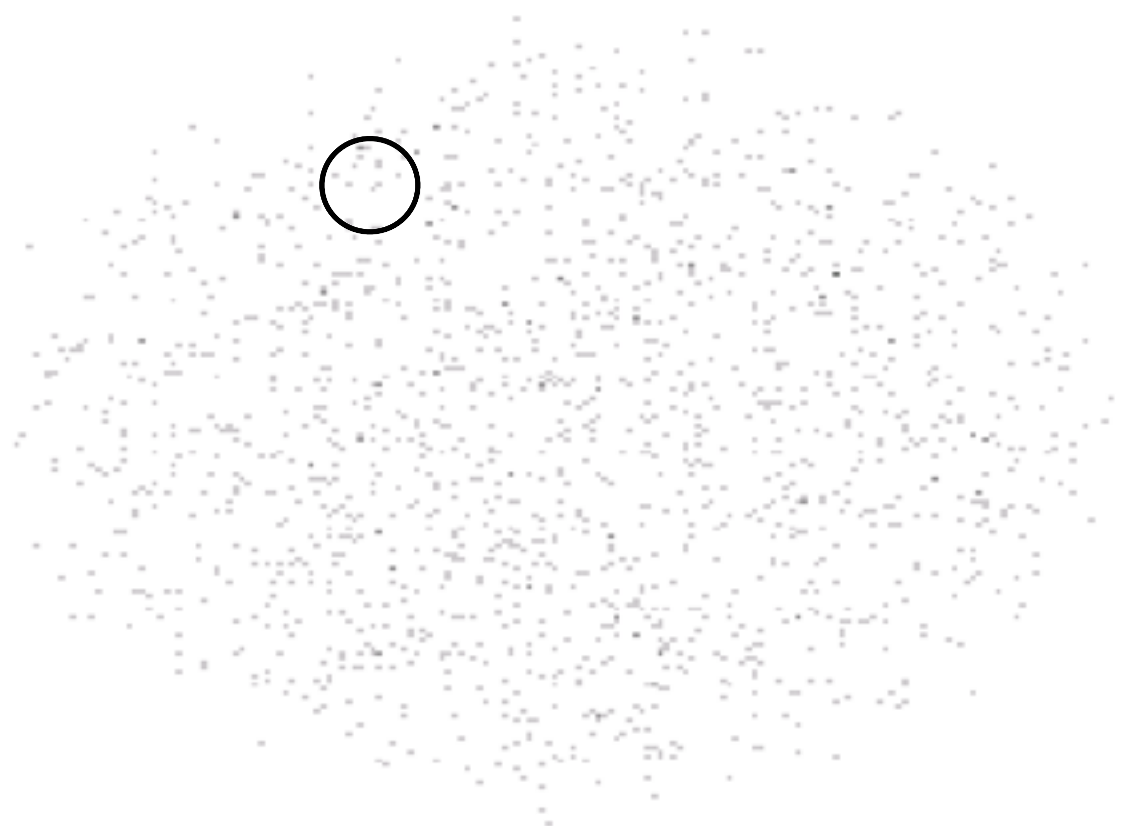


SciNet

..and it often does.

- Much of scientific computing is "data parallel"
- Same operation on each cell of grid, on each particle in domain.





SCINet

Let's get straight to it

- From login node, ssh to arc01 (devel node of accelerator research cluster)
- `cd ~ppp; cp -r ~ljdursi/ppp/cuda .; cd cuda`
- `source ../setup`
- `module load cuda`
- `make`
- `./saxpy`

SciNet

$$\vec{y} = a\vec{x} + b$$

```
void cpu_saxpb(const float *x, float a, float b, float *y, int n) {

    int i;
    for (i=0;i<n;i++) {
        y[i] = a*x[i]+b;
    }
    return;
}
```

(run several times for timing)

```
tick(&cputimer);
for (i=0; i<niters;i++)
    cpu_saxpb(x, a, b, y, n);
cputime = tock(&cputimer);
```

saxpy.cu

Question: How would we OpenMP this? MPI this?

SciNet

$$\vec{y} = a\vec{x} + b$$

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

    int i = threadIdx.x;
    yd[i] = a*xd[i]+b;
    return;
}
```

saxpy.cu

Very fine-grained parallelism.
Each core does one (or few) tasks.

SciNet

$$\vec{y} = a\vec{x} + b$$

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

    int i = threadIdx.x;
    yd[i] = a*xd[i]+b;
    return;
}
```

```
    cuda_saxpb<<<1, n>>>(xd, a, b, yd, n);
```

saxpy.cu

For loop over elements is implied by the call;
n in the <<<>>>'s invokes n of these kernels in parallel.

SciNet

```
/* run GPU code */
CHK_CUDA( cudaMalloc(&xd, n*sizeof(float)) );
CHK_CUDA( cudaMalloc(&yd, n*sizeof(float)) );

tick(&gputimer);
CHK_CUDA( cudaMemcpy(xd, x, n*sizeof(float), cudaMemcpyHostToDevice) );

for (i=0; i<niters; i++) {
    cuda_saxpb<<<1, n>>>(xd, a, b, yd, n);
}
CHK_CUDA( cudaMemcpy(ycuda, yd, n*sizeof(float), cudaMemcpyDeviceToHost) );
gputime = tock(&gputimer);

CHK_CUDA( cudaFree(xd) );
CHK_CUDA( cudaFree(yd) );
```

saxpy.cu

GPU Memory is separate from system memory (on card).
Have to allocate/free it, and copy data GPU↔CPU

SciNet

```
/* run GPU code */
CHK_CUDA( cudaMalloc(&xd, n*sizeof(float)) );
CHK_CUDA( cudaMalloc(&yd, n*sizeof(float)) );

tick(&gputimer);
CHK_CUDA( cudaMemcpy(xd, x, n*sizeof(float), cudaMemcpyHostToDevice) );

for (i=0; i<niters; i++) {
    cuda_saxpb<<<1, n>>>(xd, a, b, yd, n);
}
CHK_CUDA( cudaMemcpy(ycuda, yd, n*sizeof(float), cudaMemcpyDeviceToHost) );
gputime = tock(&gputimer);

CHK_CUDA( cudaFree(xd) );
CHK_CUDA( cudaFree(yd) );
```

saxpy.cu

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

    int i = threadIdx.x;
    yd[i] = a*xd[i]+b;
    return;
}
```

SciNet

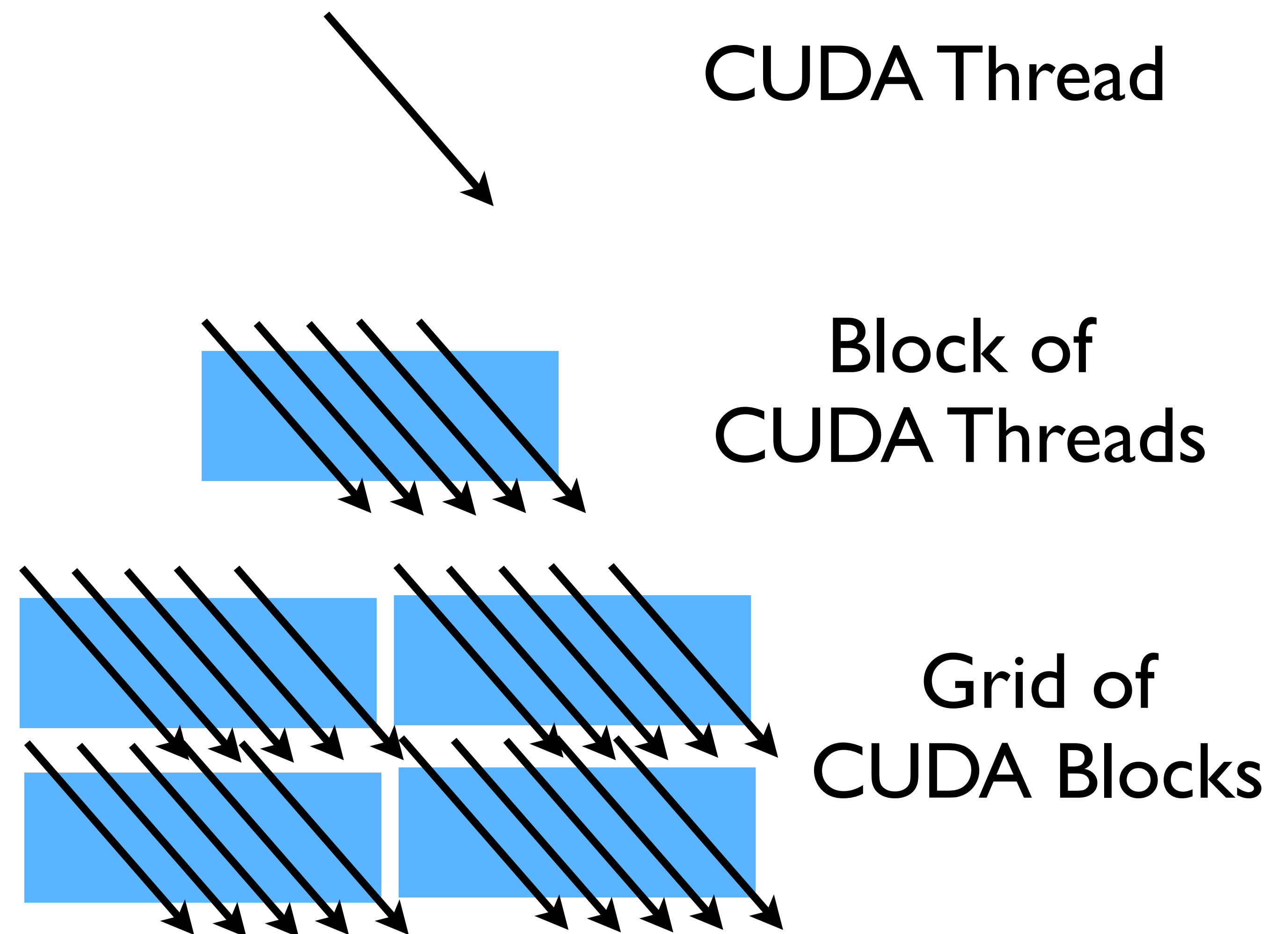
Notes:

- CHK_CUDA -- test for error cord. More later.
- Allocating, copying to GPU memory: SLOW compared to computing capability of GPU. Avoid wherever possible.
- What happens if you try ./saxpb --nvals=200 ? ./saxpb --nvals=2048 ?

SciNet

Threads, Blocks, Grids

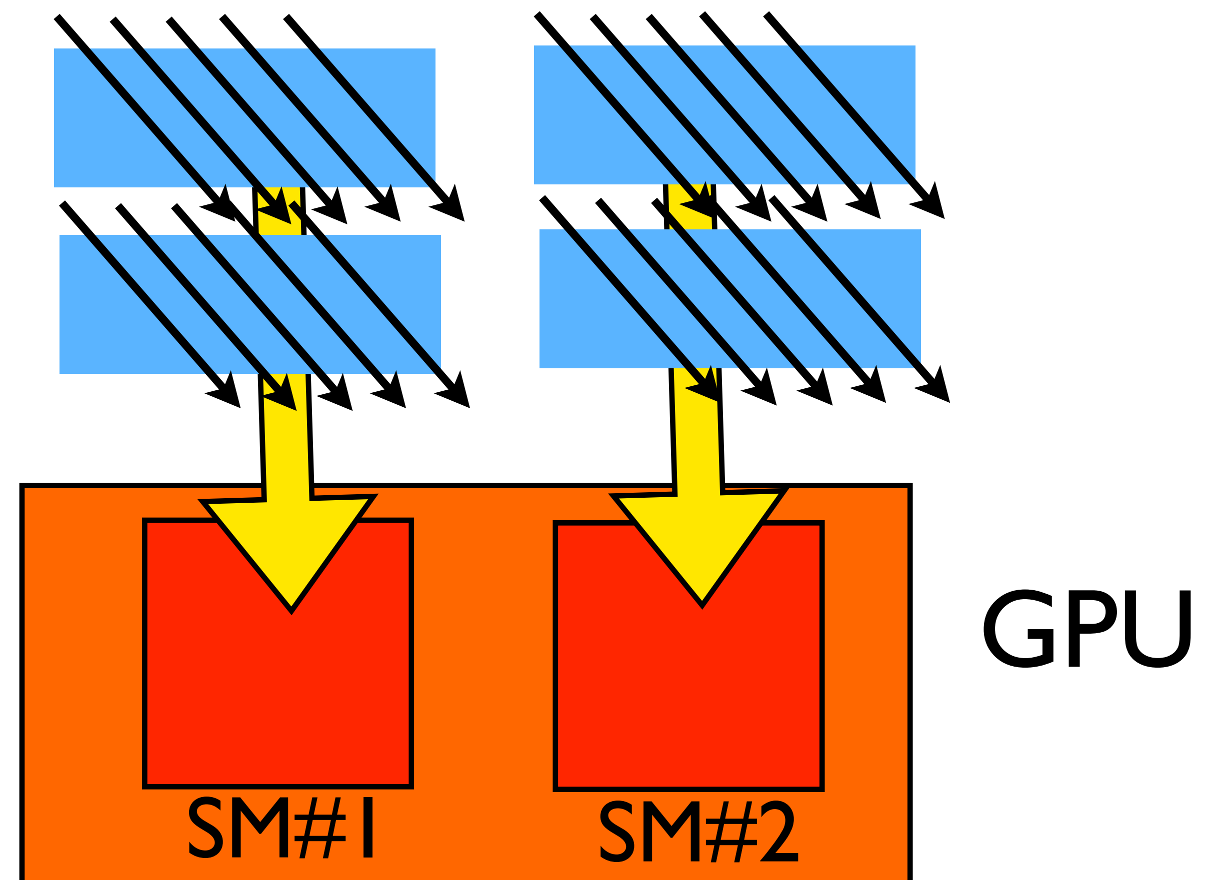
- CUDA threads are organized into blocks
- Threads operate in SIMD (ish) manner -- each executing same instructions in lockstep.
- Only difference are thread ids
- Can have a grid of multiple blocks



SciNet

CUDA - H/W mapping

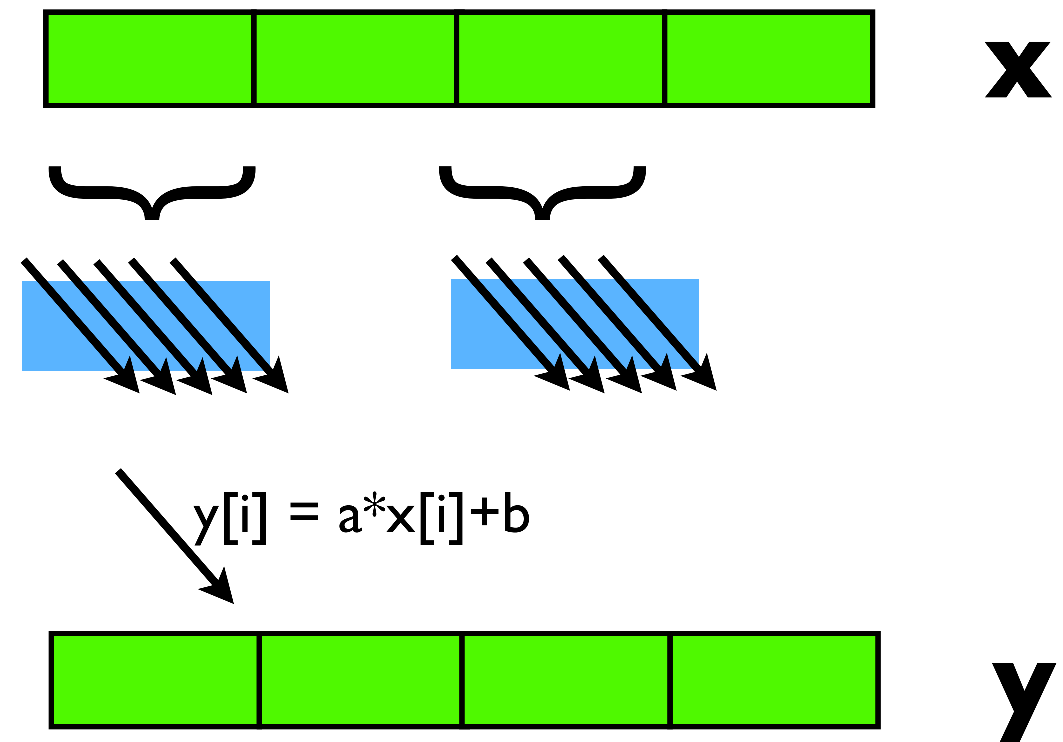
- Blocks are assigned to a particular SM
  - Executed there one ‘warp’ at a time (typically 32 threads)
- Multiple blocks may be on SM concurrently
  - Good; latency hiding
  - Bad - SM resources must be divided between blocks
- If only use 1 Block - 1 SM



SciNet

Multi-block y=ax+b

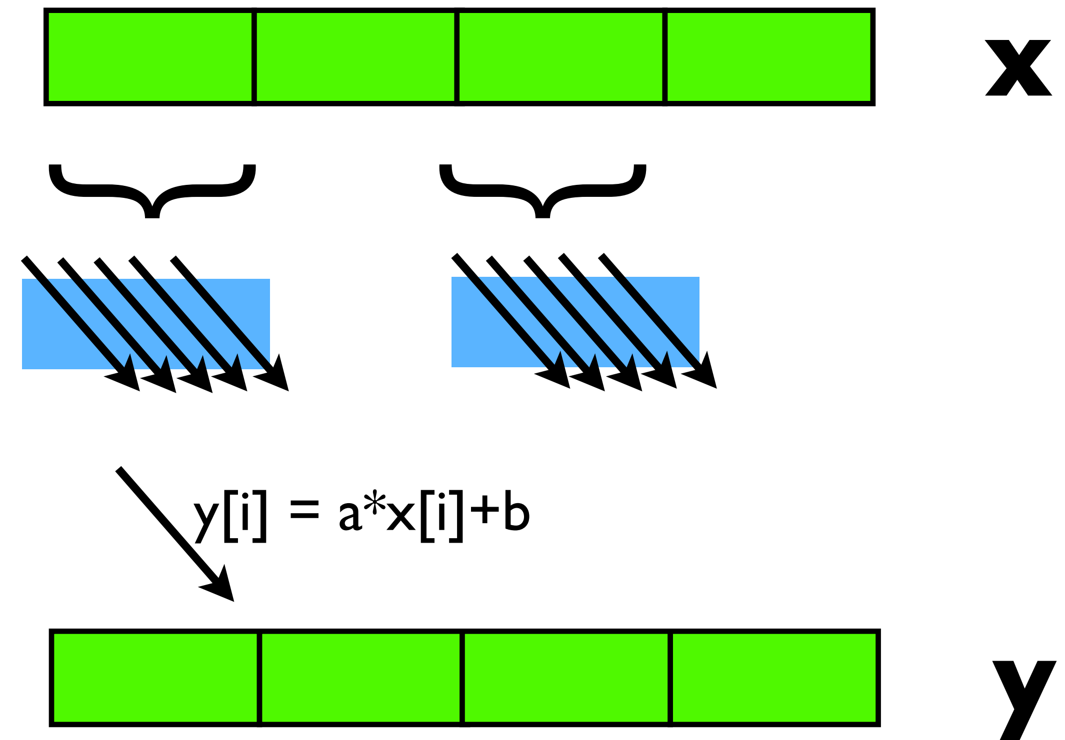
- Break input, output vectors into blocks
- Within each block, thread index specifies which item to work on
- Each thread does one update, puts results in y[i]



SciNet

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

SciNet

Hands on -- do multi-block saxpb

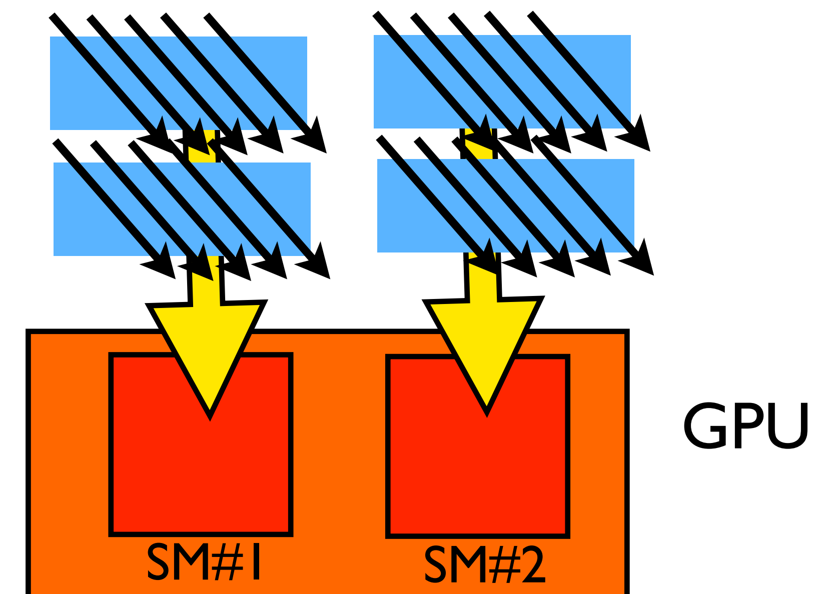
SciNet

More blocks → more SMs → more FLOPs

- 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
```

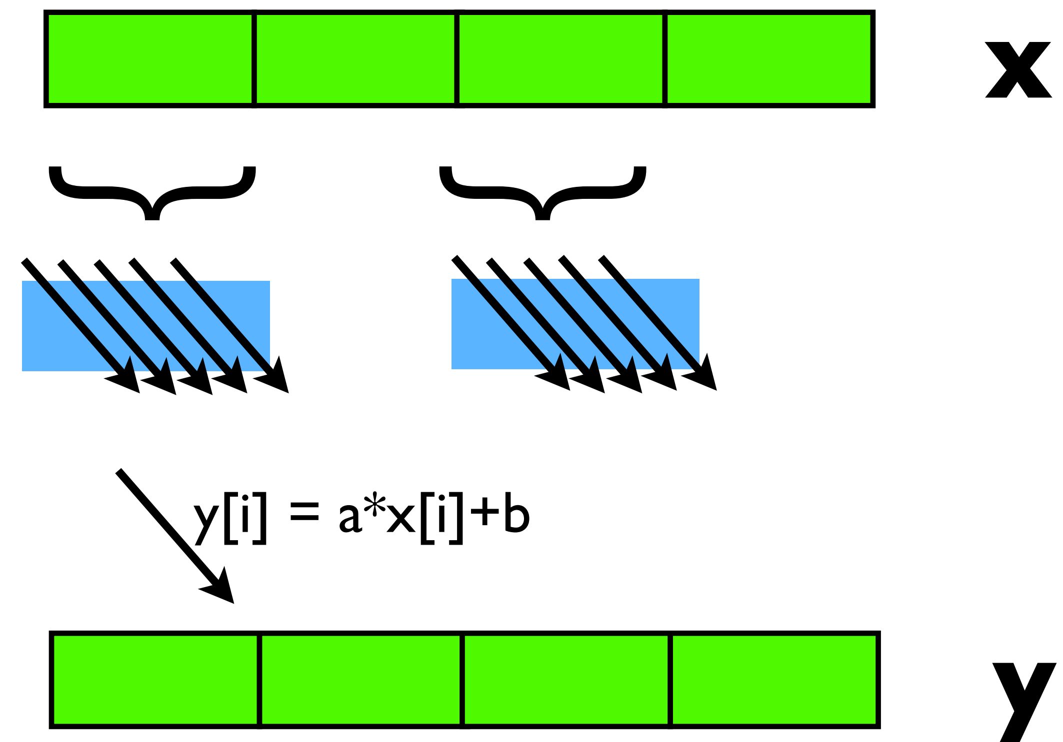


SciNet

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;
}
```

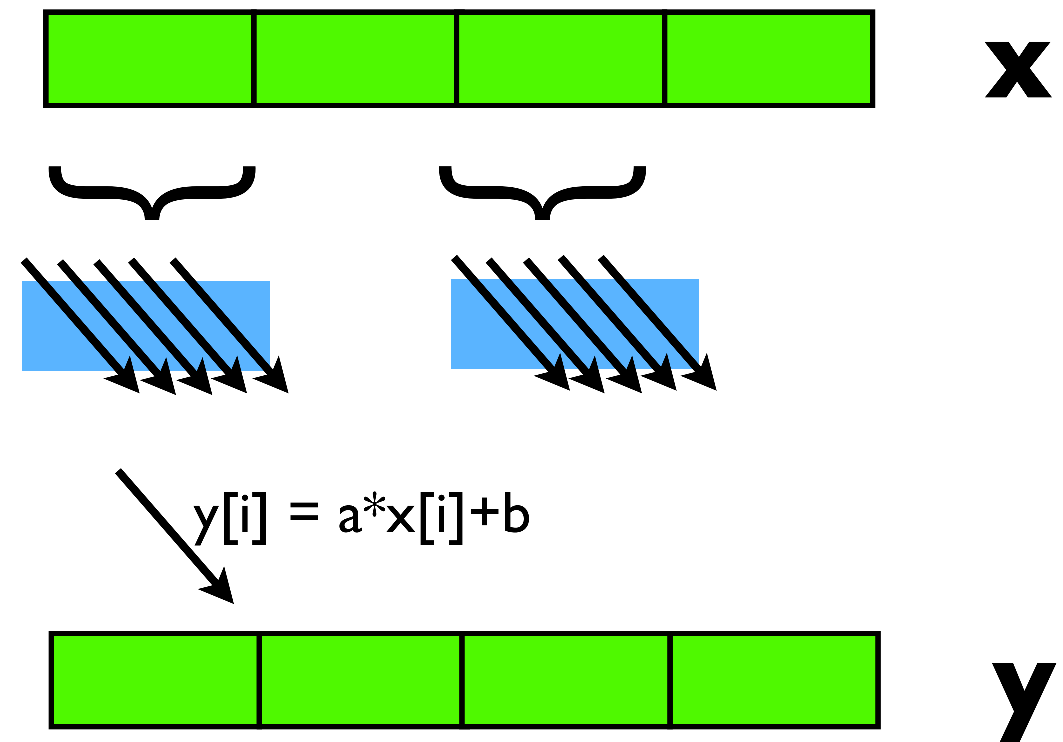
Index *within* block (0..blocksize-1)



SciNet

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;
}
```



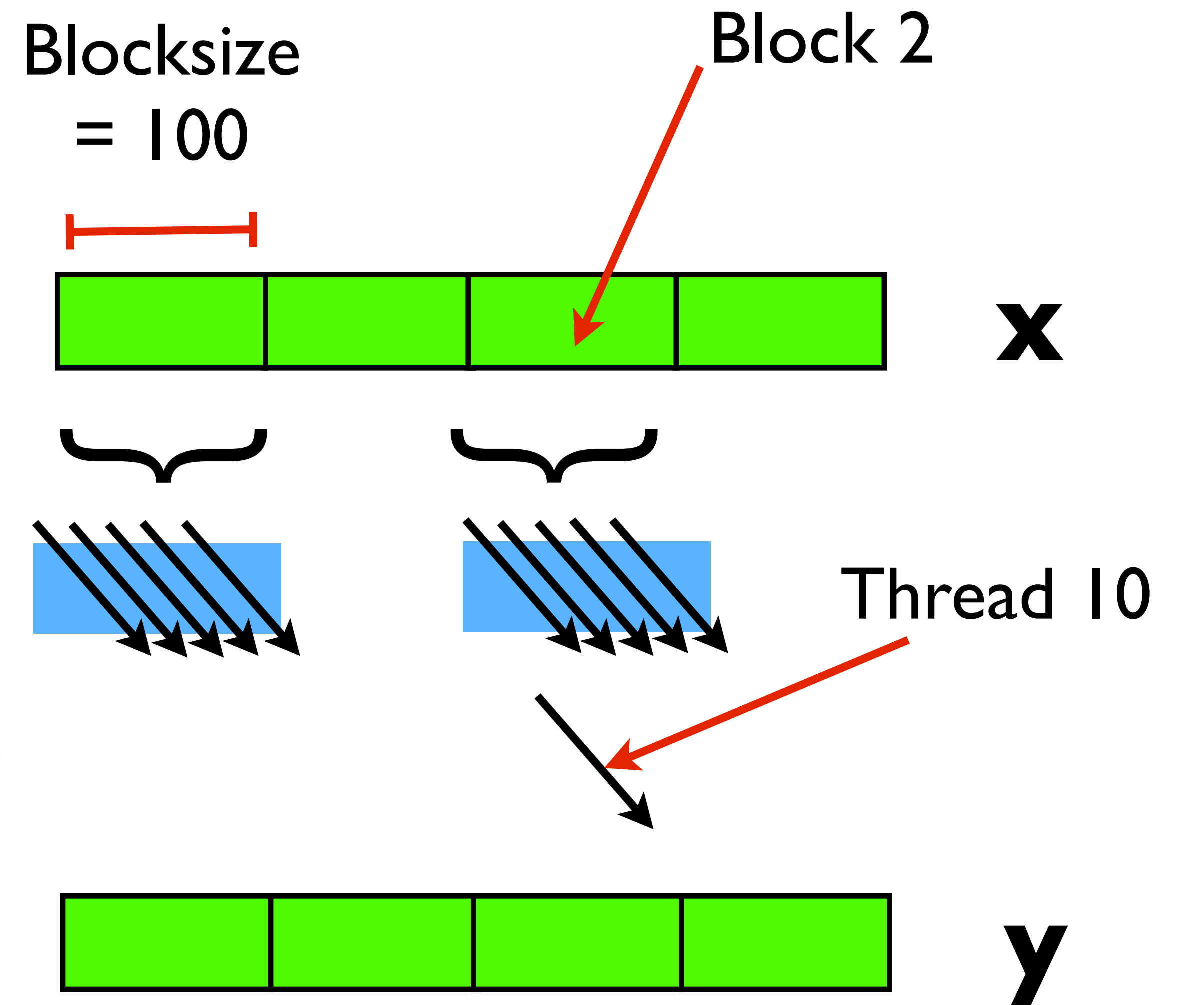
Index *of* block
(0..nblocks-1)

Size of block
(blocksize)

SciNet

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;
}
```

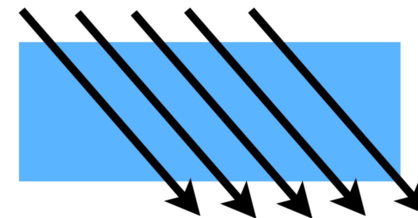



```
i = 10 + 2*100  = 210
yd[210] = a*xd[210] + b
```

SciNet

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:



SciNet

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

SciNet

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.

SciNet

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;
}
```

SciNet

Why the .xs?

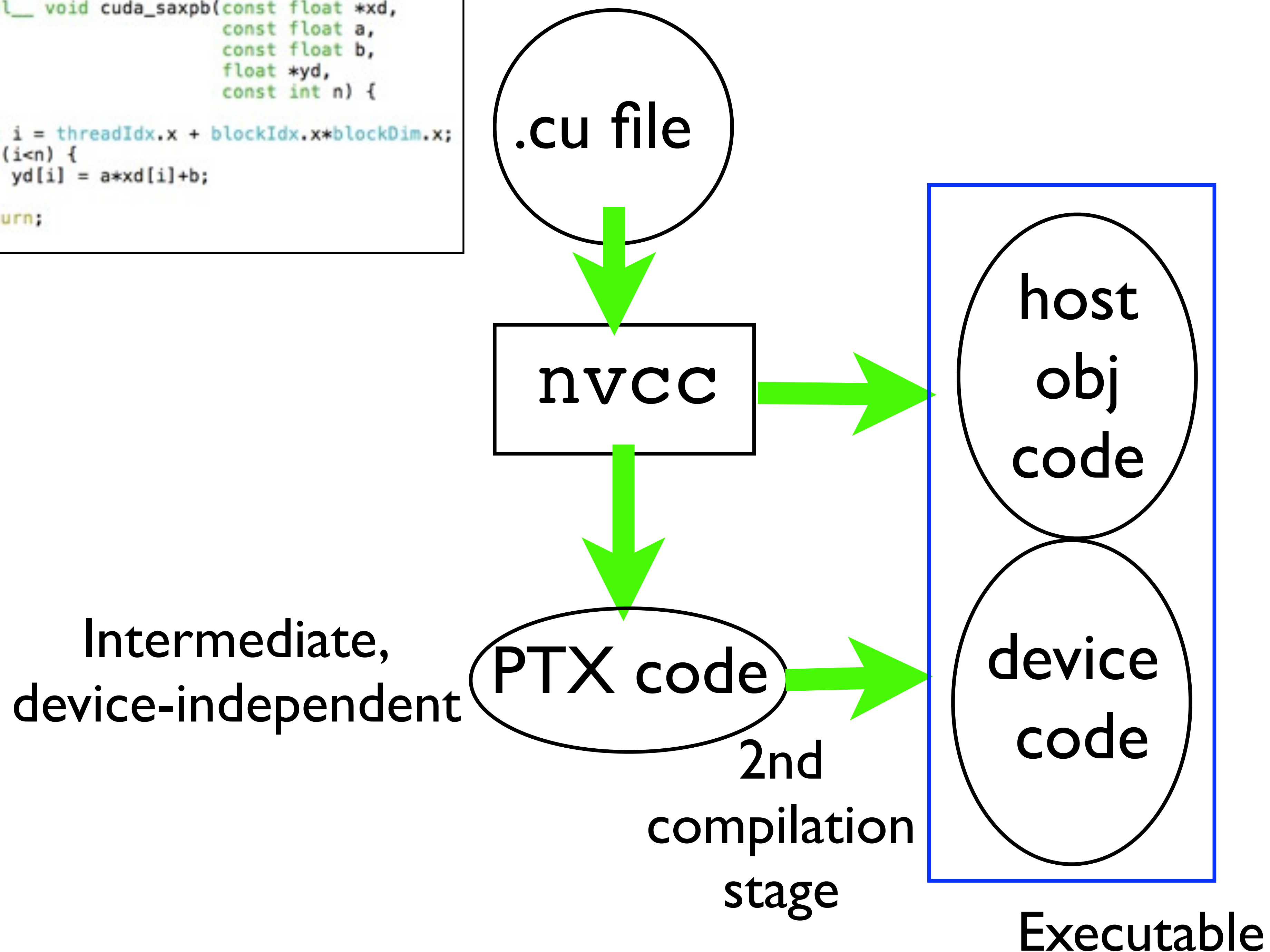
- `__global__` - device code that can be seen (invoked) from host.
- `__host__` - default. Not usually interesting.
- `__device__` - device code. Can be called only from other device code.
- `__host__ __device__` - compiled for both host and device.

```
__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;
}
```

SciNet

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;
}
```



SciNet

Restrictions

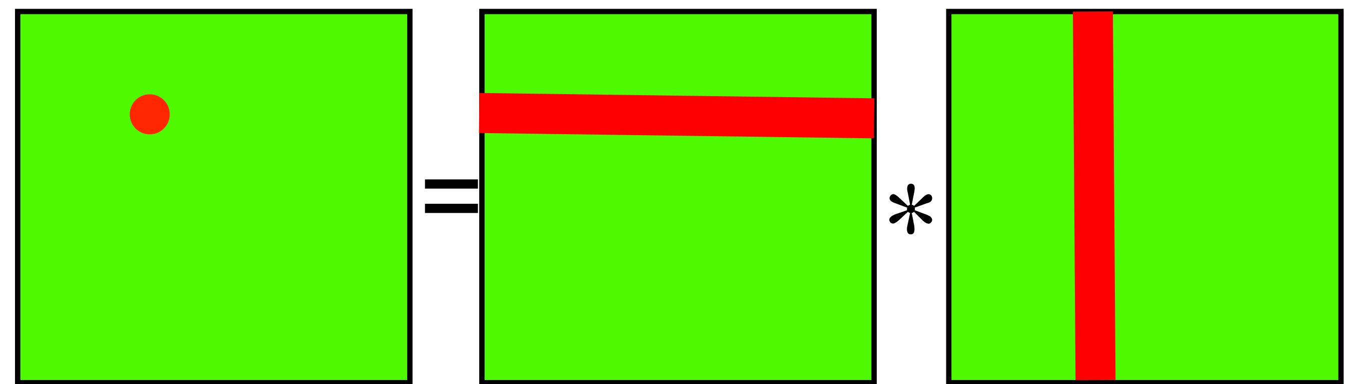
- __global__ functions can't recurse, neither can __device__ on non-Fermis
- No function pointers to __device__ functions on non-fermis, can't take address of __device__ function
- Can't have static variables in __global__, __device__ functions
- Can't use varargs with device code

```
__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;
}
```

SciNet

2-Dimensional Blocks

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



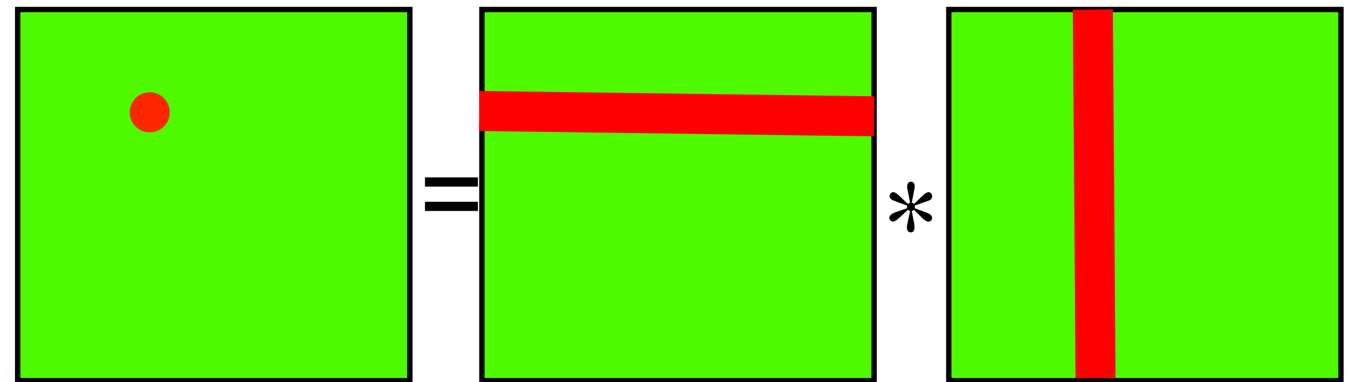
$$C_{i,j} = \sum_k A_{i,k} B_{k,j}$$

SciNet

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;
}
```

matmult.cu

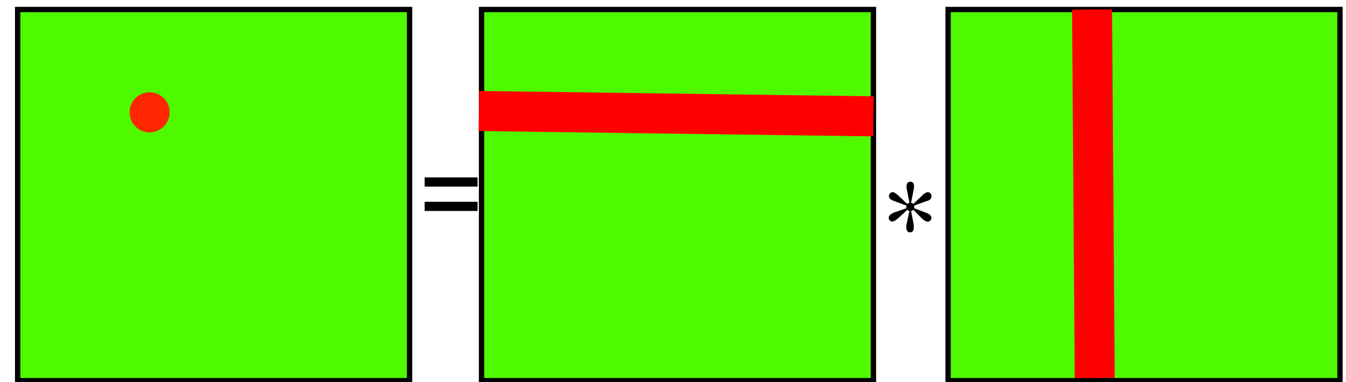


$$C_{i,j} = \sum_k A_{i,k} B_{k,j}$$

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;
}
```



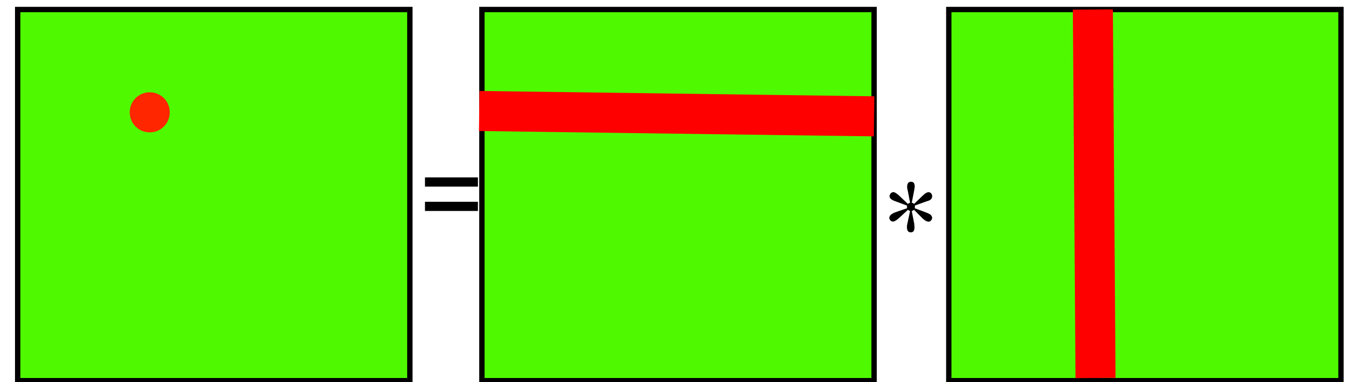
$$C_{i,j} = \sum_k A_{i,k} B_{k,j}$$

```
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);
```

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;
}
```



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

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

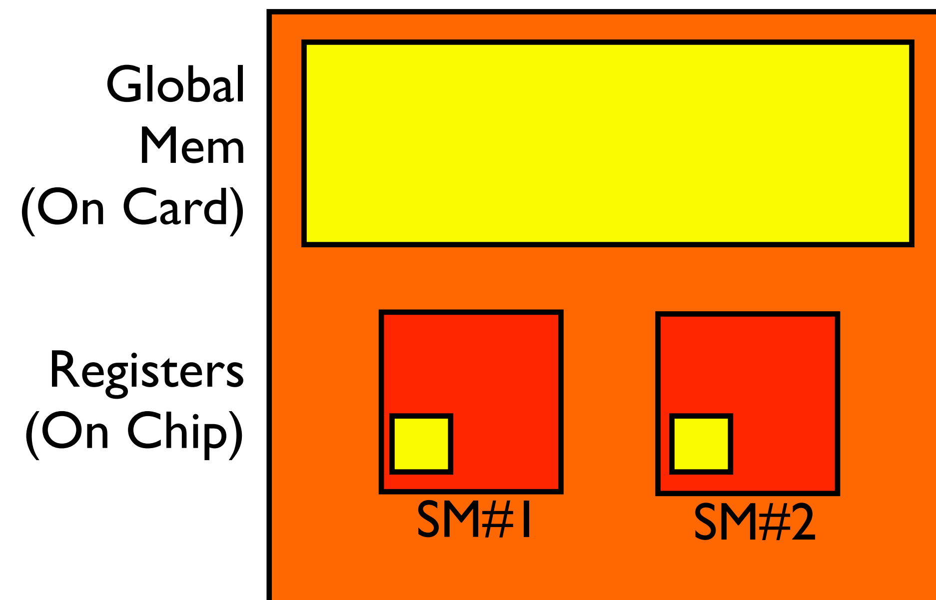
```
$ ./matmult --matsize=160 --nblocks=10
Matrix size = 160, Number of blocks = 10.
CPU time = 14.047 millisec.
GPU time = 2.219 millisec.
CUDA and CPU results differ by 0.000000
```

Faster, even with double precision sums - why?

SciNet

CUDA Memories

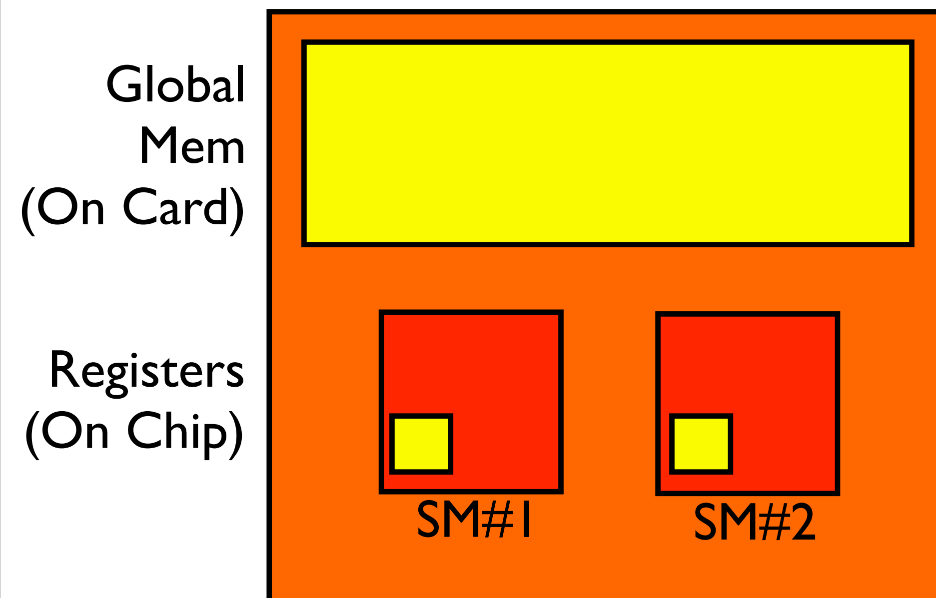
- All HPC, but especially GPU, all about planning memory access to be fast
- Global mem is off the GPU chip (but on the card); ~100 cycle latency
- Thread-local variables get put into registers on each SM - fast (~1 cycle) but small



SciNet

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

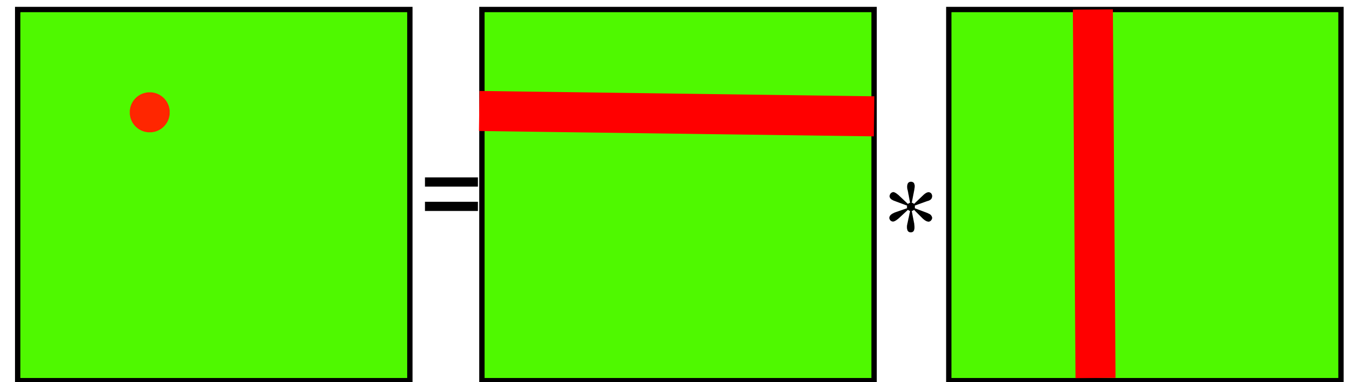


* if you run out of registers, will put 'local' mem in global.

SciNet

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

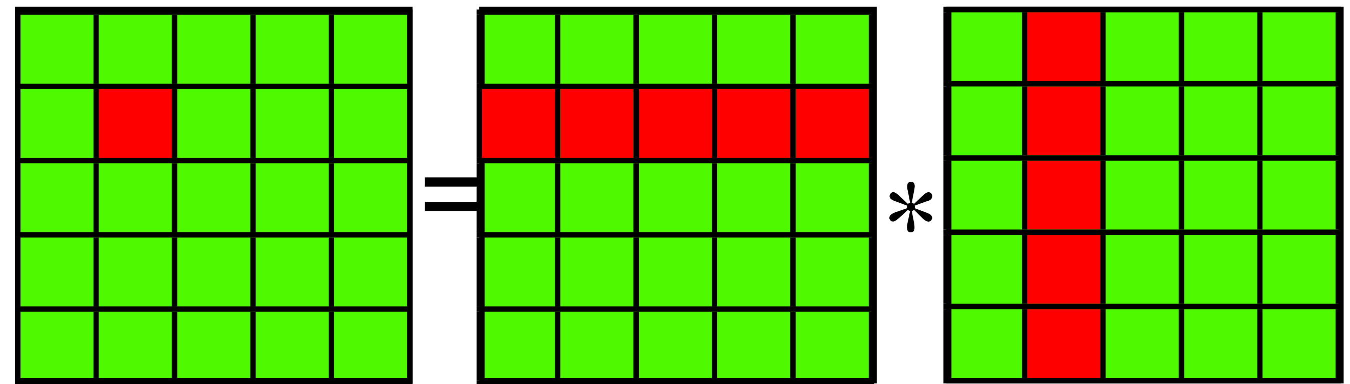


$$C_{i,j} = \sum_k A_{i,k} B_{k,j}$$

SciNet

Memory usage in SGEMM

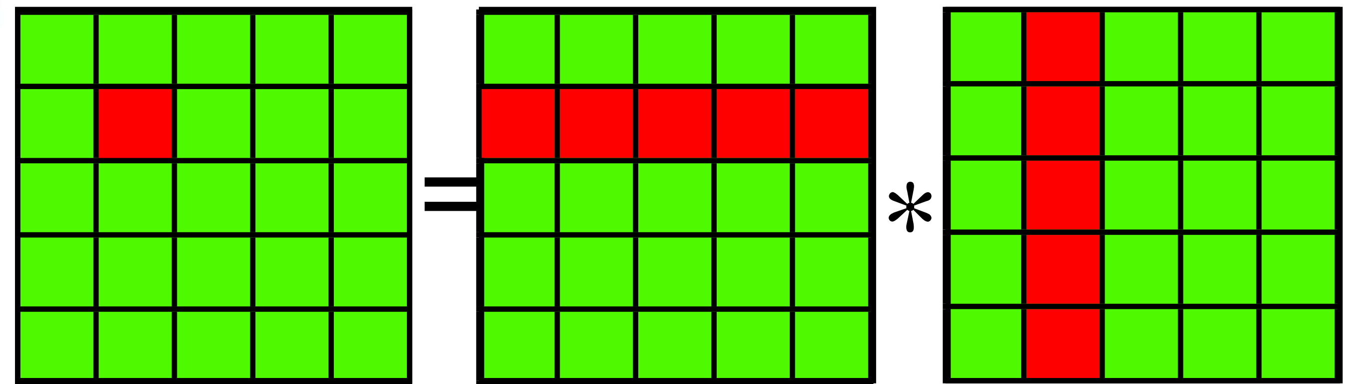
- One nice thing about matrix multiplication - same as block multiplication, each sub-block is a matrix mult
- Neighbouring threads within block all see nearby rows, columns
- Pull whole block in
- If *b* blocks in each dim, each data only pulled in *2b* times, not *2n* times



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

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;
```

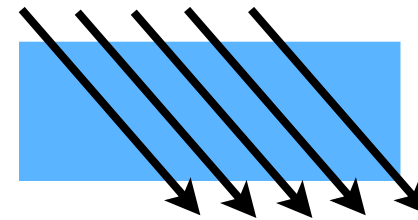


$$\mathrm{C}_{bi,bj} = \sum_k \mathrm{A}_{bi,bk}\mathrm{B}_{bk,bj}$$

SciNet

__syncthreads()

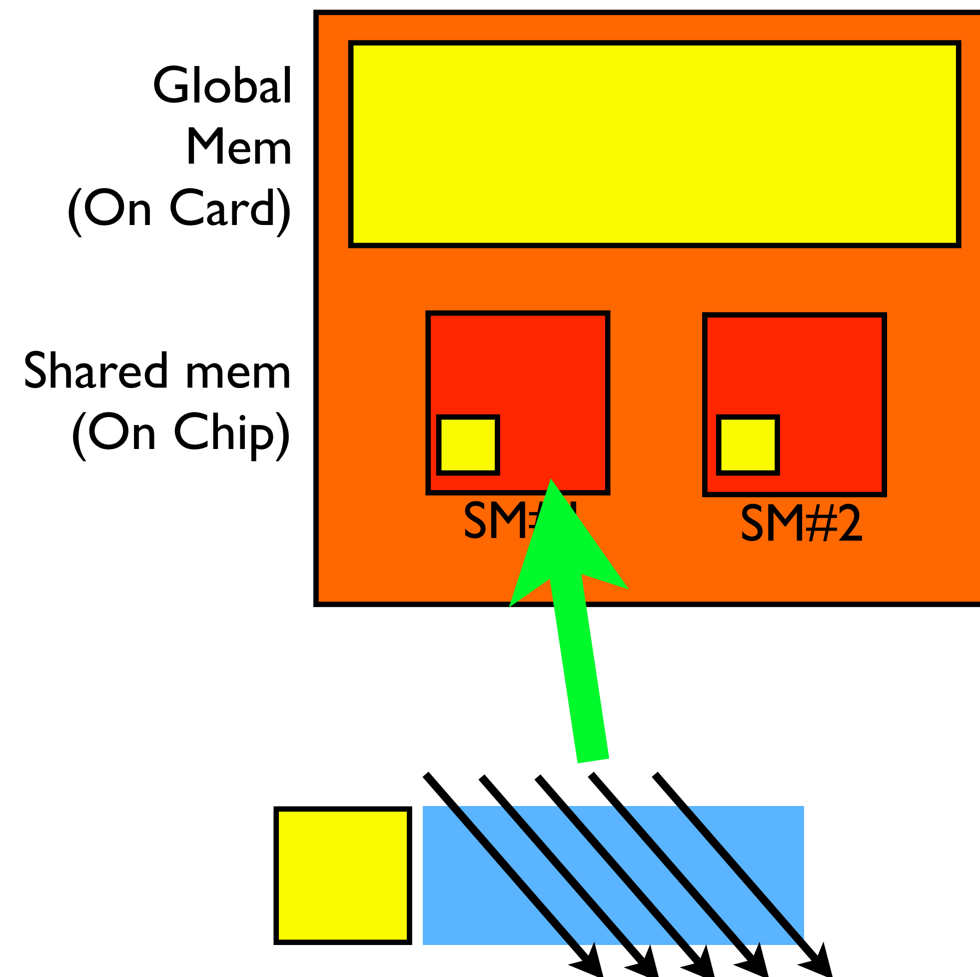
- Computation must wait until all threads have brought in their data
- Not all memory accesses may take same length of time
- __syncthreads() - waits until all threads in block are at same point.
- *No* equivalent between blocks
- Loop must similarly wait for computation



SciNet

__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



SciNet

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

SciNet

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

SciNet

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.

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
```

Shared

```
$ ./matmult --matsize=160 --nblocks=10
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
```

SciNet

Hands On

- Using matmult.cu as a template, look at laplacian.c implements 2d laplacian.
- Implement a CUDA version using shared memory, and make sure it gets same answer as CPU version.
- How would we do multiple iterations? Does entire memory have to be copied back each time?

SciNet

Making effective use of CUDA memories

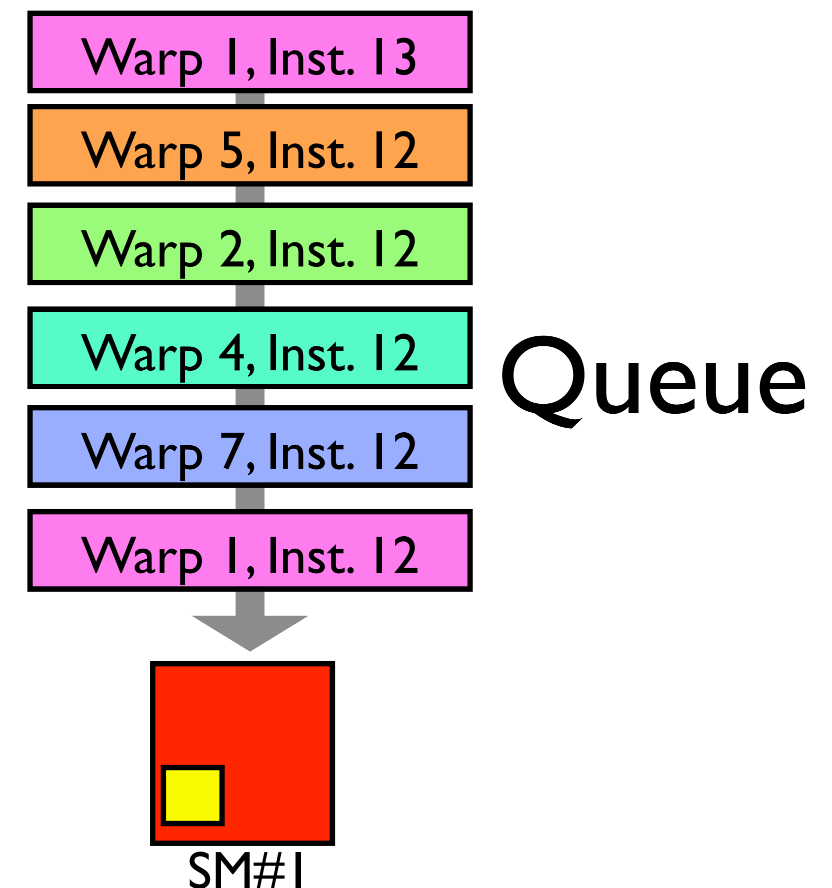
- Preload data wherever possible
- Global memory -
  - Coalesced access
  - Make use of 128B (or, maybe, 32B) at a time
- Profiler to see what's happening
- Shared memory
  - Bank conflicts

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

SciNet

Stalling on Memory Access

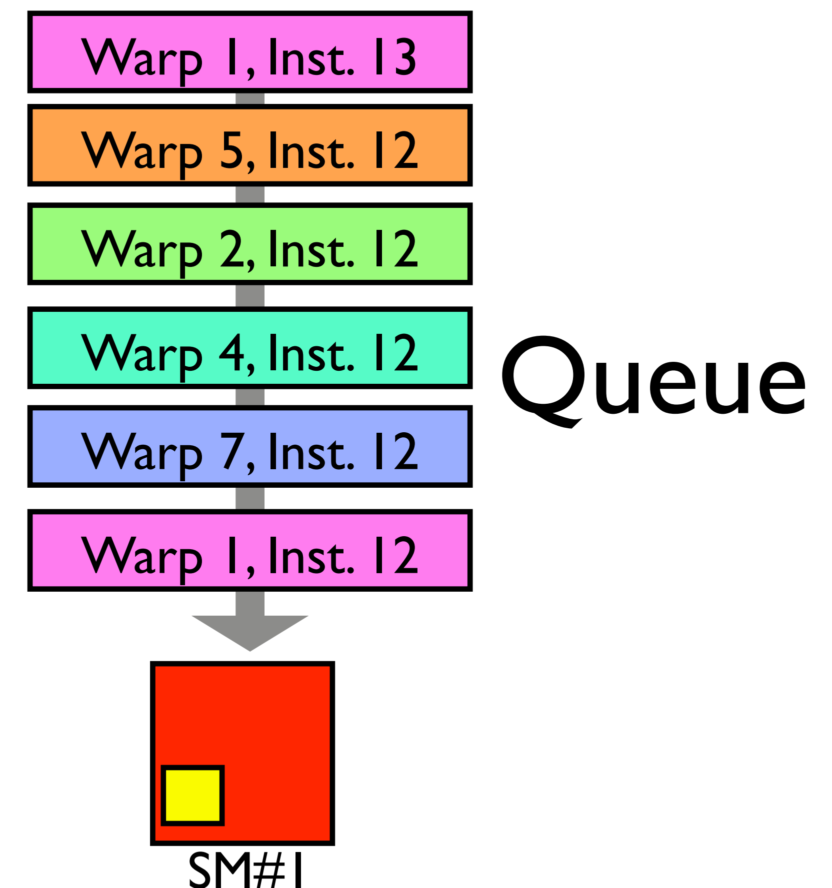
- Graphics card schedules by the warp on an SM
- All warps that are ready to execute get scheduled
- Not ready to execute - stalled on memory access
- Nothing ready - SM sits idle.



SciNet

Stalling on Memory Access

- Two ways to ensure no idle SM:
  - Lots of warps (=blocks*threads/32); hide latency with other threads.
  - Little or no stalling on memory access; hide latency within threads.
- Sometimes work to counter purposes! Must experiment to see what works best for your algorithm.



SciNet

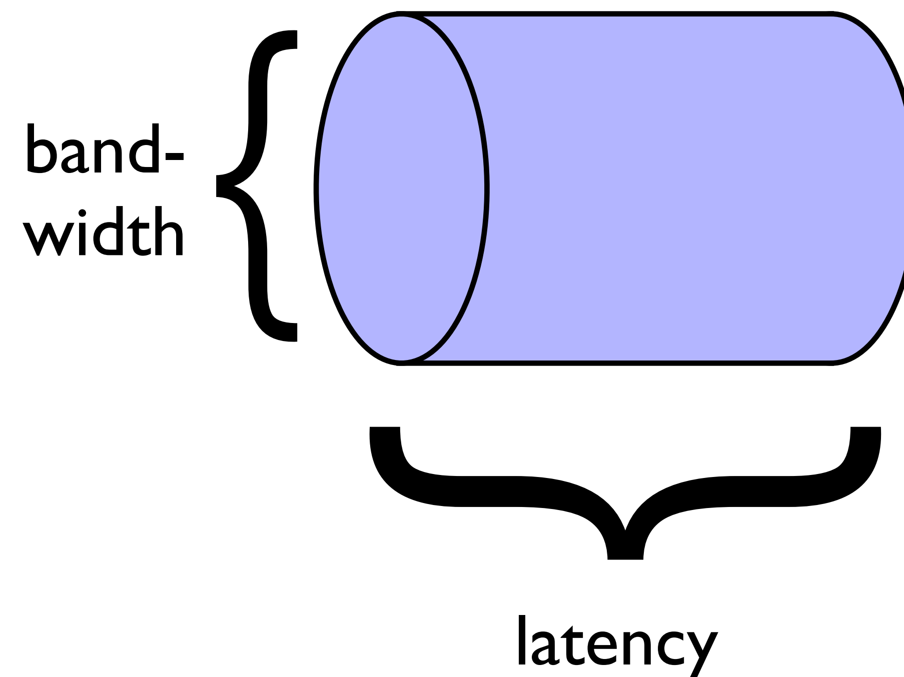
Stalling happens on *use*.

- Kernel does not stall on loading data
- Stalls when data not yet ready needs to be used
- Can "preload" data that you will need at beginning of kernel
- Hide latency by doing as much work as possible before need bulk of data.

```
__global__ mykernel(__device__ const float *ind,
                    __device__ float *outd) {

    float a;
    float b;    } register vars
    float c;

    a = ind[threadIdx.x];      <- ok
    b = ind[2*threadIdx.x];    <- ok

    c = a + b;                 <- stall

    /*.... */
}
```

Keep memory accesses going

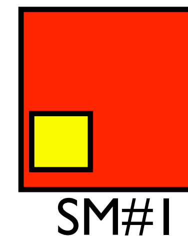
- Make maximum use of memory bandwidth hardware provides
- To fully use a pipe, must have bandwidth x latency memory accesses 'in flight'.
- Little's Law, Queueing theory - http://en.wikipedia.org/wiki/Little%27s_law

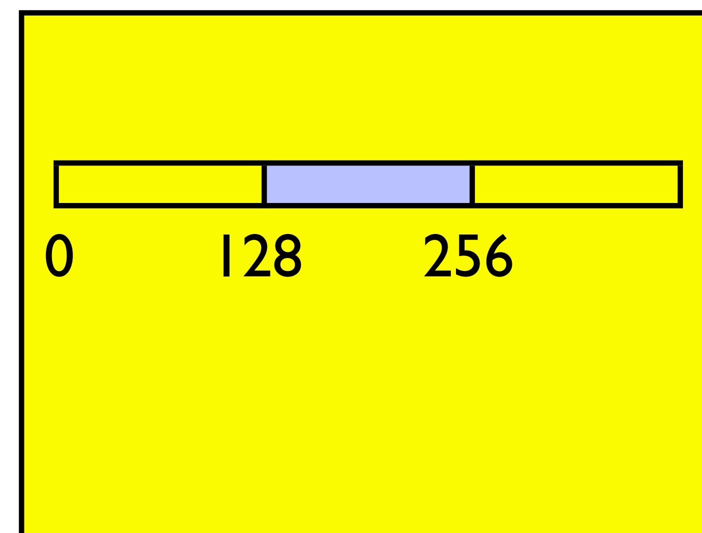


SciNet

Coalesced Memory Access

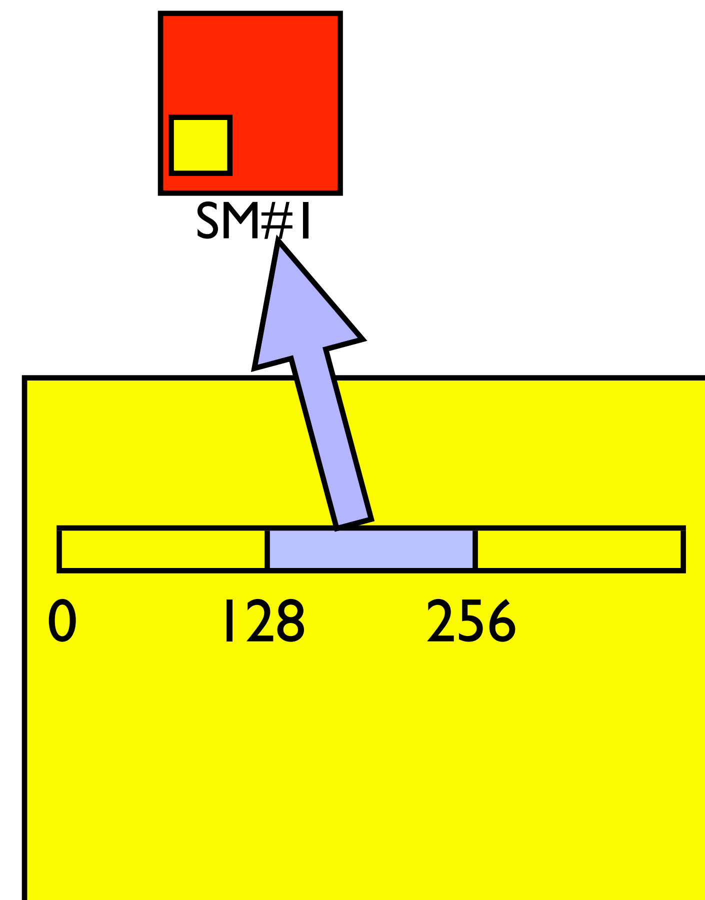
- Global memory is slow
- Get as much out of it per access as possible
- HW reads 128 byte lines from global memory (Fermi: can turn off caching and read 4x 32byte segments)
- Want to make the most of this





Coalesced Memory Access

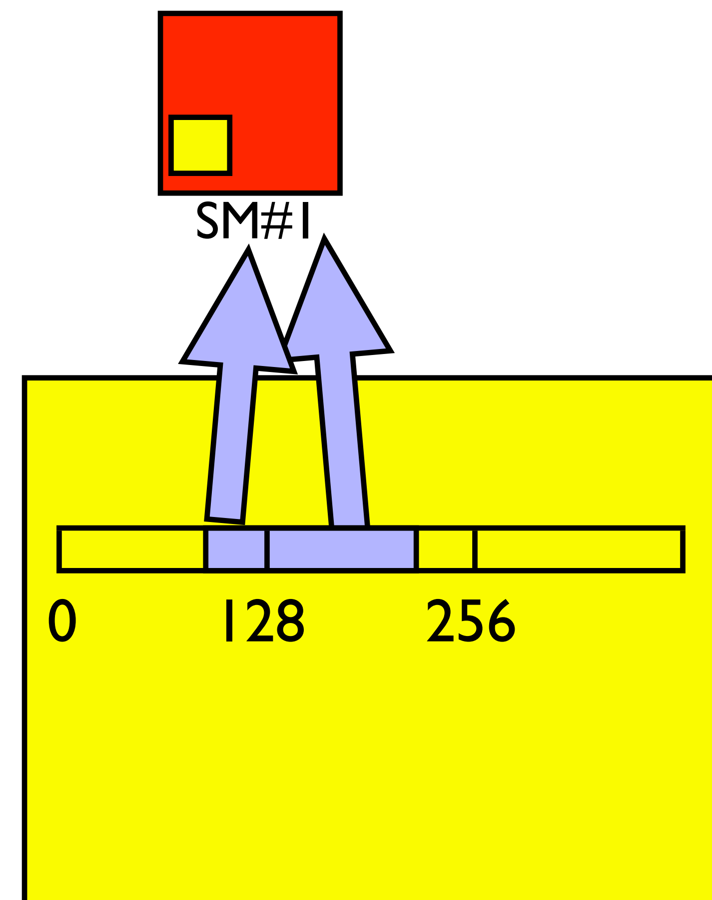
- Corresponds to 4B for each thread in a warp
- If each thread in warp reads consecutive float, aligned w/ boundary, can be coalesced into 1 read: high bandwidth
- Warp can continue after 1 global read cycle



SciNet

Coalesced Memory Access

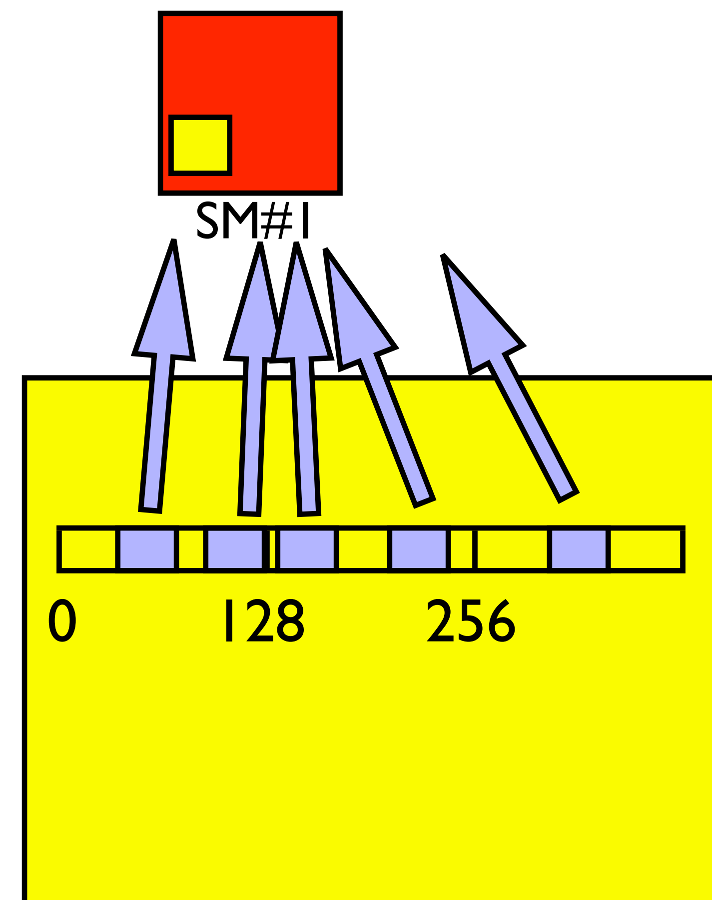
- If each thread in warp reads consecutive float, but offset, can be coalesced into 2 read: reduced bandwidth
- Warp can continue after 2 global read cycle (and 128B of bandwidth wasted)



SciNet

Coalesced Memory Access

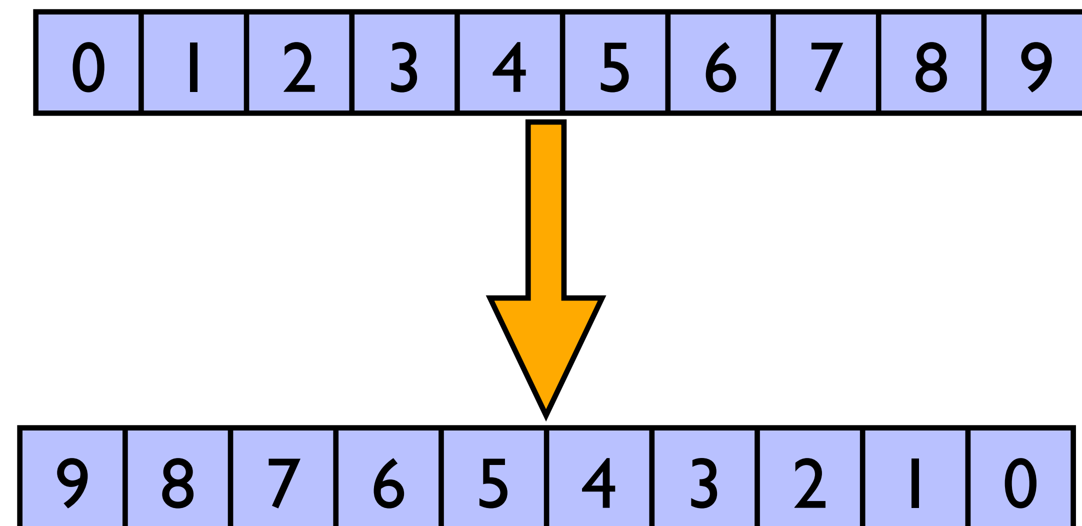
- Random access is a nightmare
- Can potentially take 32 times as long, wasting 97% of available global memory bandwidth



SciNet

List reversal

- Imagine having to reverse a list
- (Sounds dumb, but matrix transpose, partial pivoting, various graph algorithms require data reordering)
- Obvious way to do this, particularly on older (pre cc 1.2) hardware, doesn't work well:



SciNet

List reversal

```
__global__ void cuda_reverse(const float *xd,
                                   float *yd,
                             const int n) {

    int i = threadIdx.x + blockIdx.x*blockDim.x;
    if (i<n) {
        yd[n-(i+1)] = xd[i];
    }
    return;
}
```

List reversal

```
__global__ void cuda_reverse(const float *xd,
                                   float *yd,
                             const int n) {

    int i = threadIdx.x + blockIdx.x*blockDim.x;
    if (i<n) {
        yd[n-(i+1)] = xd[i];
    }
    return;
}
```

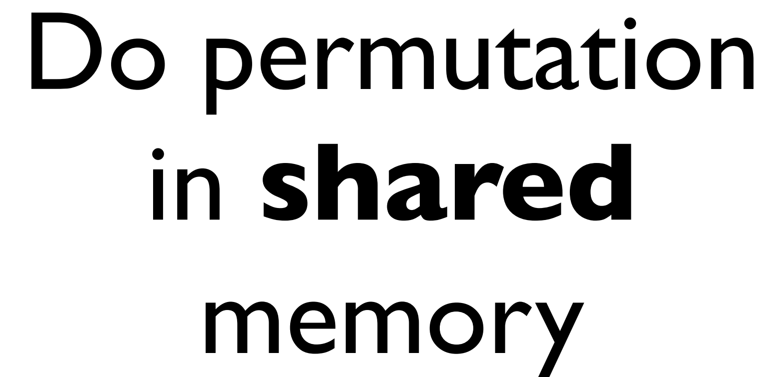
Read - coalesced

Write - reversed - possibly noncoalesced

SciNet

List reversal

```
__global__ void cuda_reverse_coalesced(const float *xd,
                                       float *yd,
                                 const int n) {

    extern __shared__ float blockdata[];
    int iin = threadIdx.x + blockIdx.x*blockDim.x;
    int outblock = gridDim.x - (blockIdx.x + 1);
    int iout = threadIdx.x + outblock*blockDim.x;

    if (iin<n) {
        blockdata[threadIdx.x] = xd[iin];
        __syncthreads();
        yd[iout] = blockdata[blockDim.x - (threadIdx.x+1)];
    }
    return;
}
```

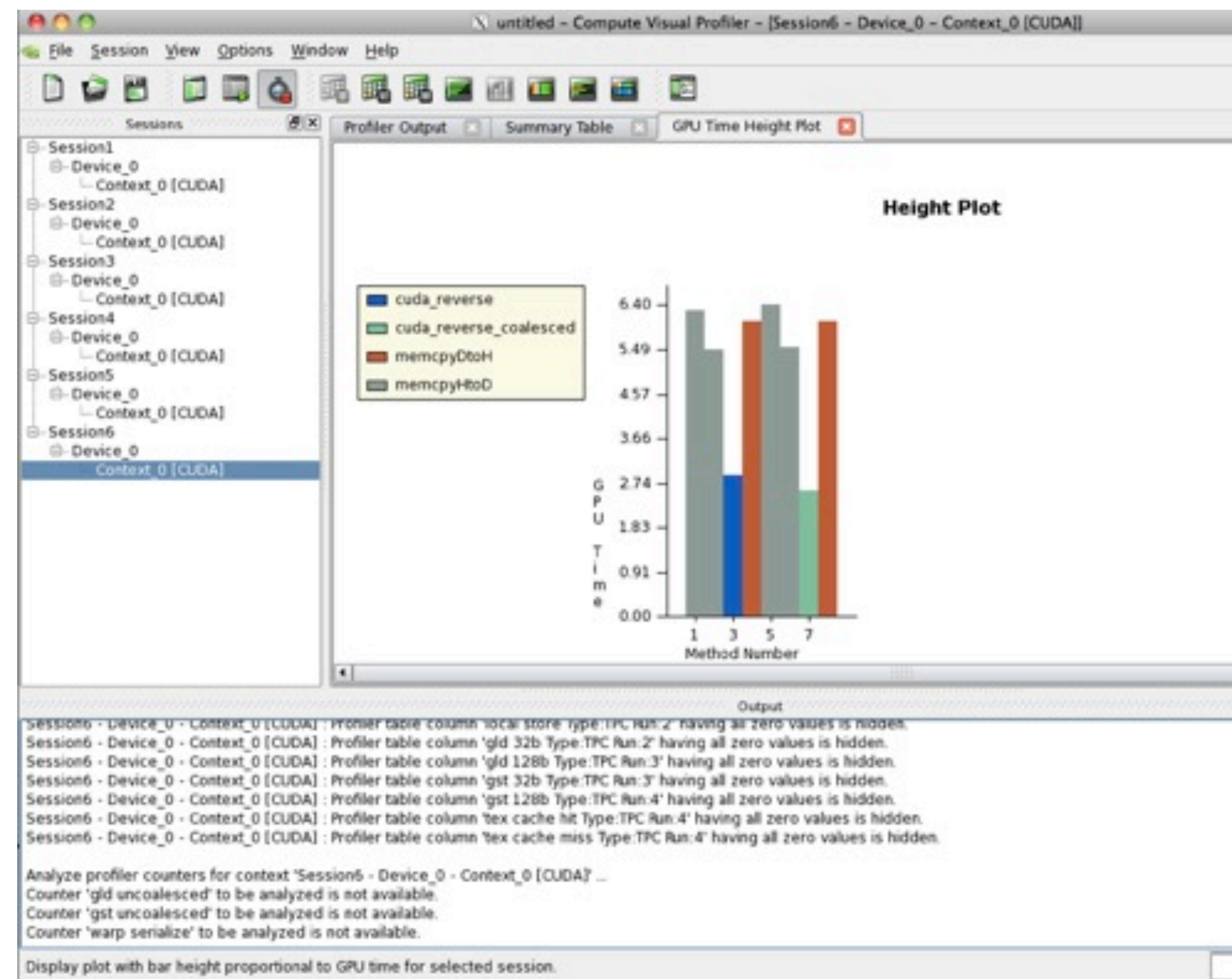



```
[ljdursi@tpb1 class4]$ ./reverse --nvals=960 --nblocks=
For run with n = 960, nblocks = 30, blocksize = 32,
iters=1,
CPU time  = 0.002 millisec.
GPU time  = 0.101 millisec, diff = 0.000000.
GPU2 time = 0.059 millisec, diff = 0.000000.
```

SciNet

Visual Profiler

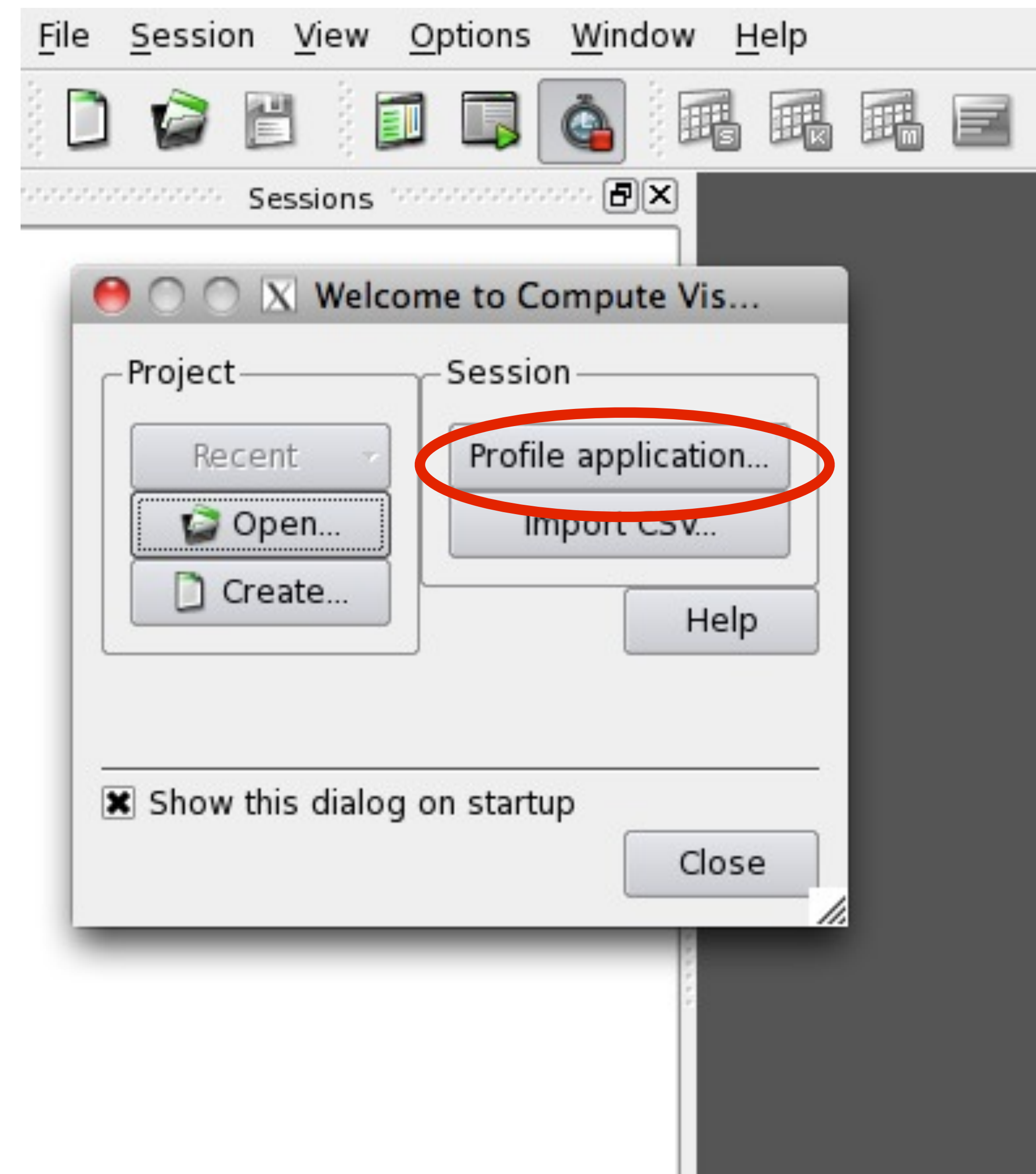
- Sometimes we'd like to see more detail than just integrated timings
- Cuda/OpenCL profiler comes with NVidia SDK
- run with `computeprof` (/scinet/arc/cuda-3.2/computeprof)
- From there, you can run an application and look at timings



SciNet

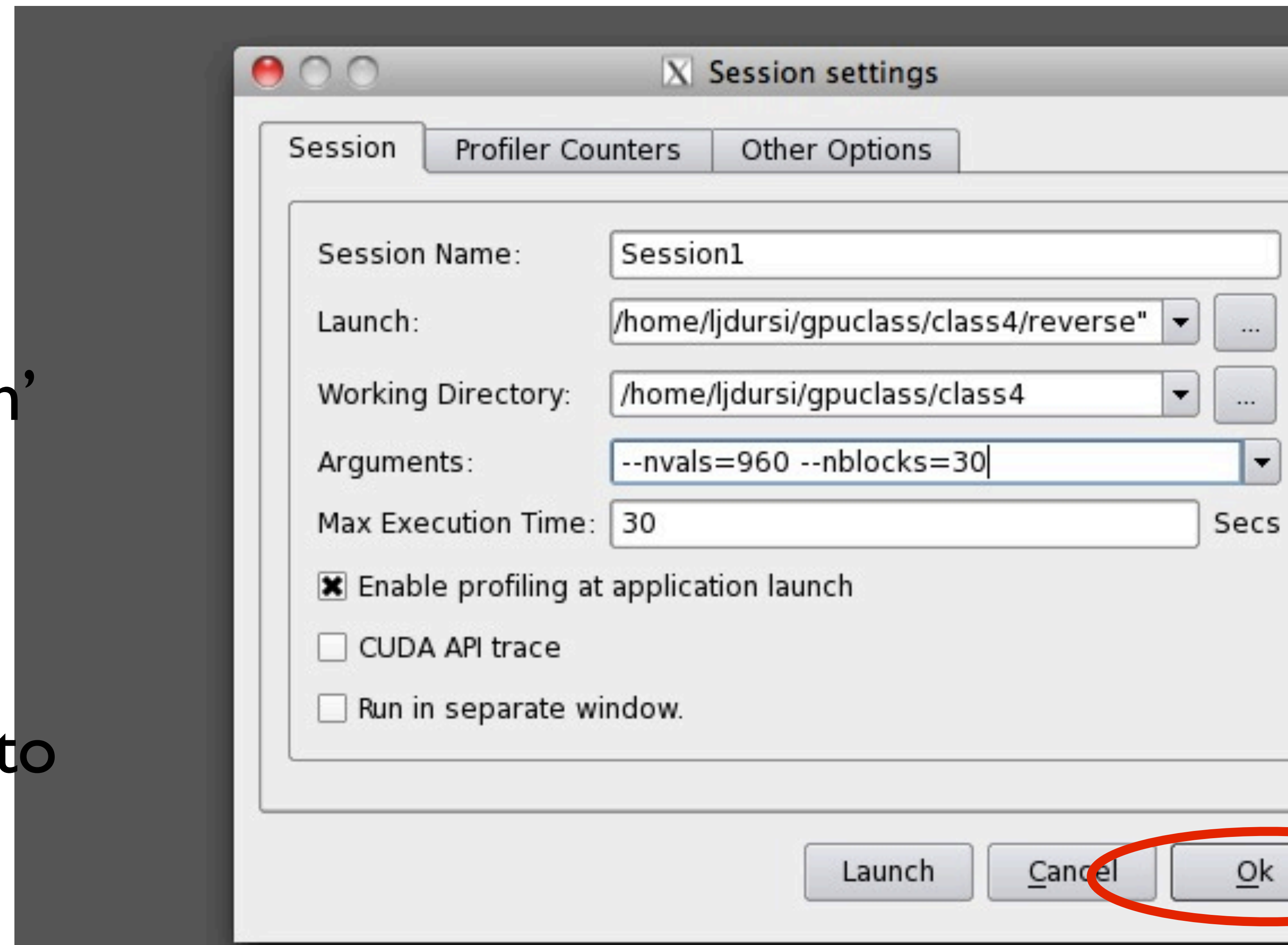
Visual Profiler

- Click 'Profile application' to begin getting data,



Visual Profiler

- Click 'Profile application' to begin getting data,
- Enter directory, executable, and arguments of program to profile,



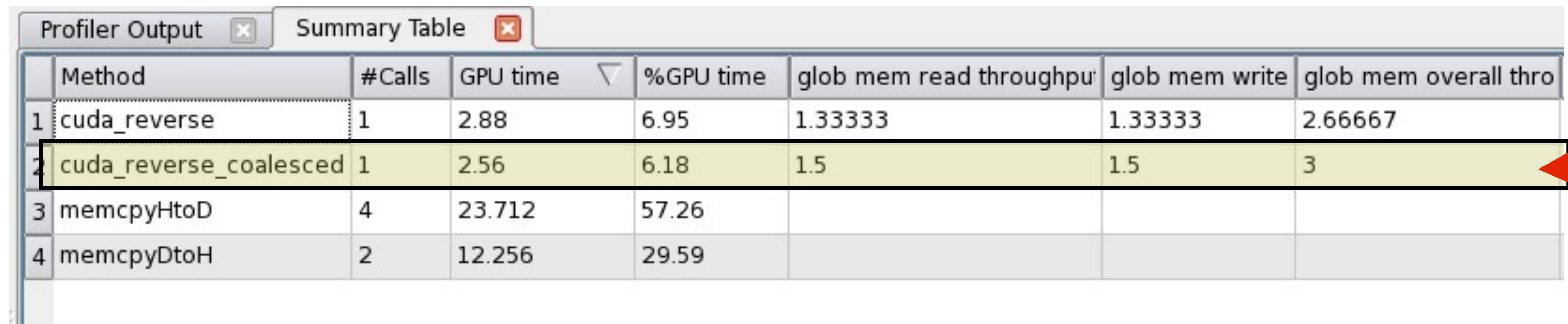
SciNet

Visual Profiler

- Click 'Profile application' to begin getting data,
- Enter directory, executable, and arguments of program to profile,
- and then run the program.   Program runs several times to get all counter information.

Visual Profiler

- Summary table shows lots of good stuff
- Here we see overall *kernel* time is about 12% faster, presumably because of roughly ~12% better global memory throughput.

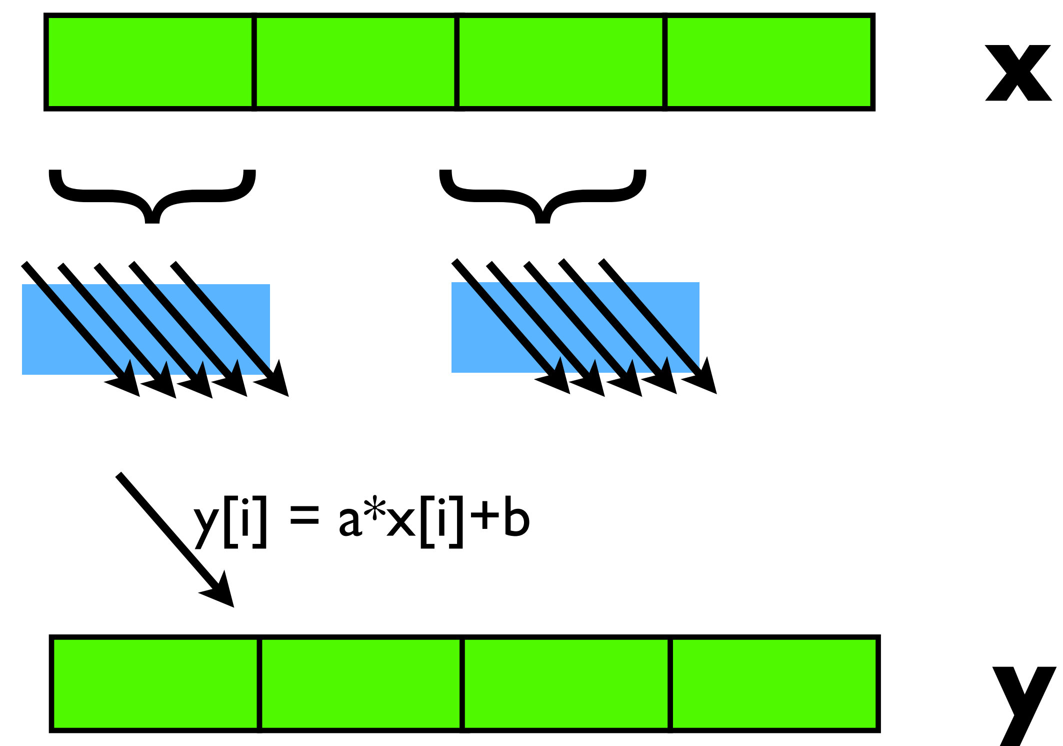


	Method	#Calls	GPU time	%GPU time	glob mem read throughpu	glob mem write	glob mem overall thro
1	cuda_reverse	1	2.88	6.95	1.33333	1.33333	2.66667
2	cuda_reverse_coalesced	1	2.56	6.18	1.5	1.5	3
3	memcpyHtoD	4	23.712	57.26			
4	memcpyDtoH	2	12.256	29.59			

SciNet

Another Example: Multi-block y=ax+b

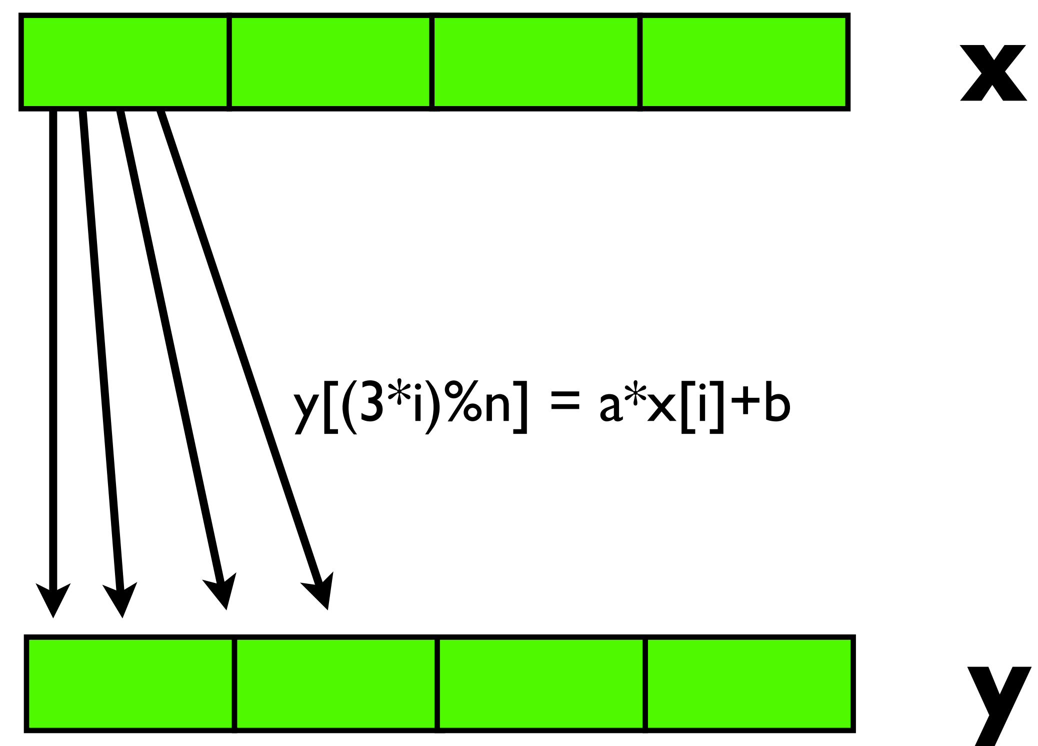
- Break input, output vectors into blocks
- Within each block, thread index specifies which item to work on
- Each thread does one update, puts results in y[i]



SciNet

Another Example: 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]
- But now with a stride:
- Can coalesce reads, writes, but not both.



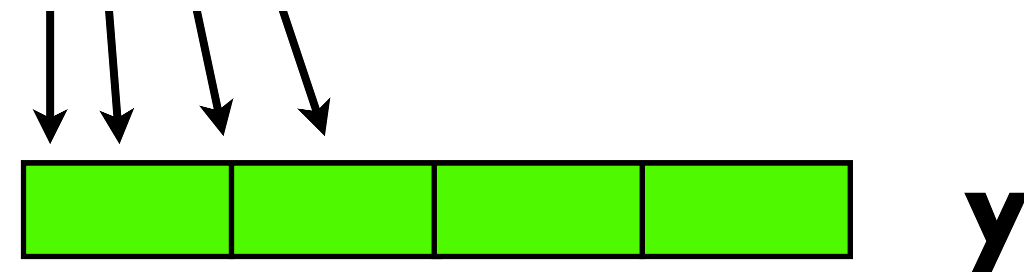
SciNet

Another Example: Multi-block y=ax+b

- Break input, output vectors into blocks

	Method	#Calls	GPU time	%GPU time	glob mem read throughput	glob mem write	glob mem overall	gld efficiency	gst efficiency	instr
1	cuda_saxpb_strided	1	4.608	7.61	18.6806	18.6806	37.3611	0.307692	0.307692	0.14
2	cuda_saxpb	1	3.008	4.97	4.78723	4.78723	9.57447	1	1	0.04
3	memcpyHtoD	4	37.088	61.32						
4	memcpyDtoH	2	15.776	26.08						

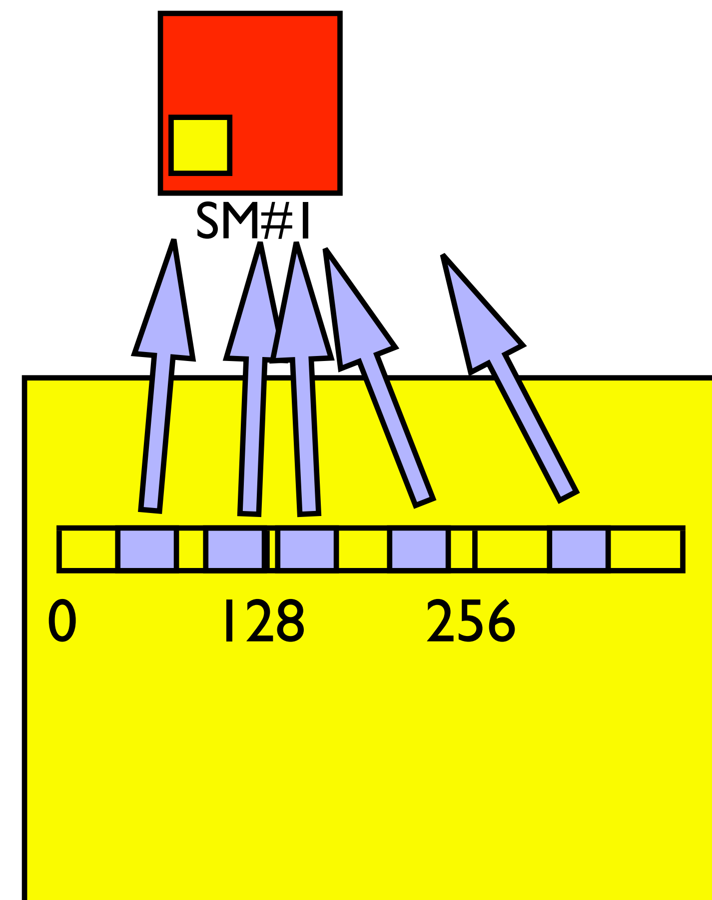
- Each thread does one update, puts results in y[i]
- But now with a stride:
- Can coalesce reads, writes, but not both.



SciNet

Coalesced Memory Access

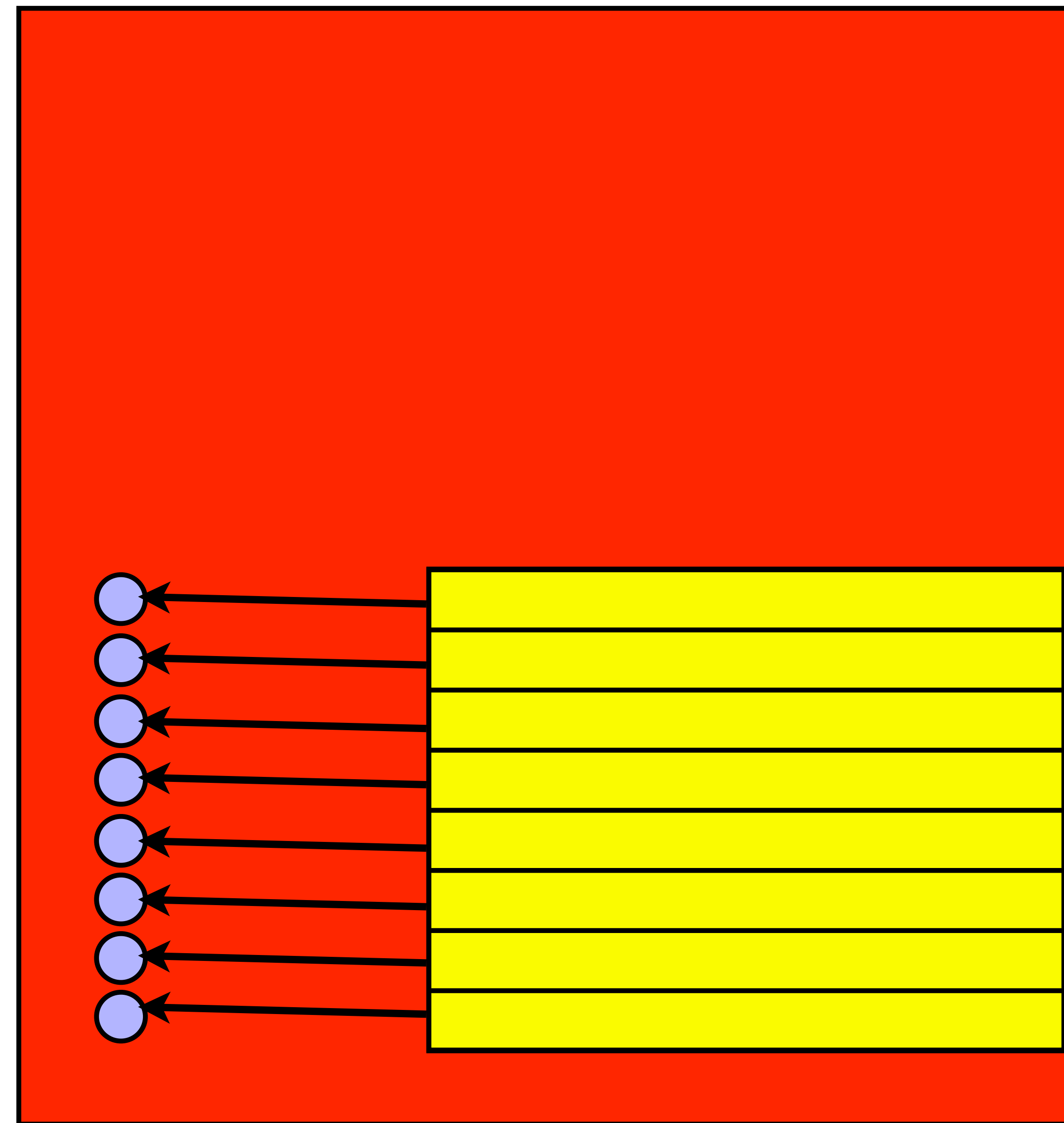
- Rewriting algorithm to ensure coalesced memory access probably most important optimization.
- Try to rearrange data before transfer to device to be in order needed;
- Reorder in shared mem if necessary.



SciNet

Shared Memory Bank Conflicts

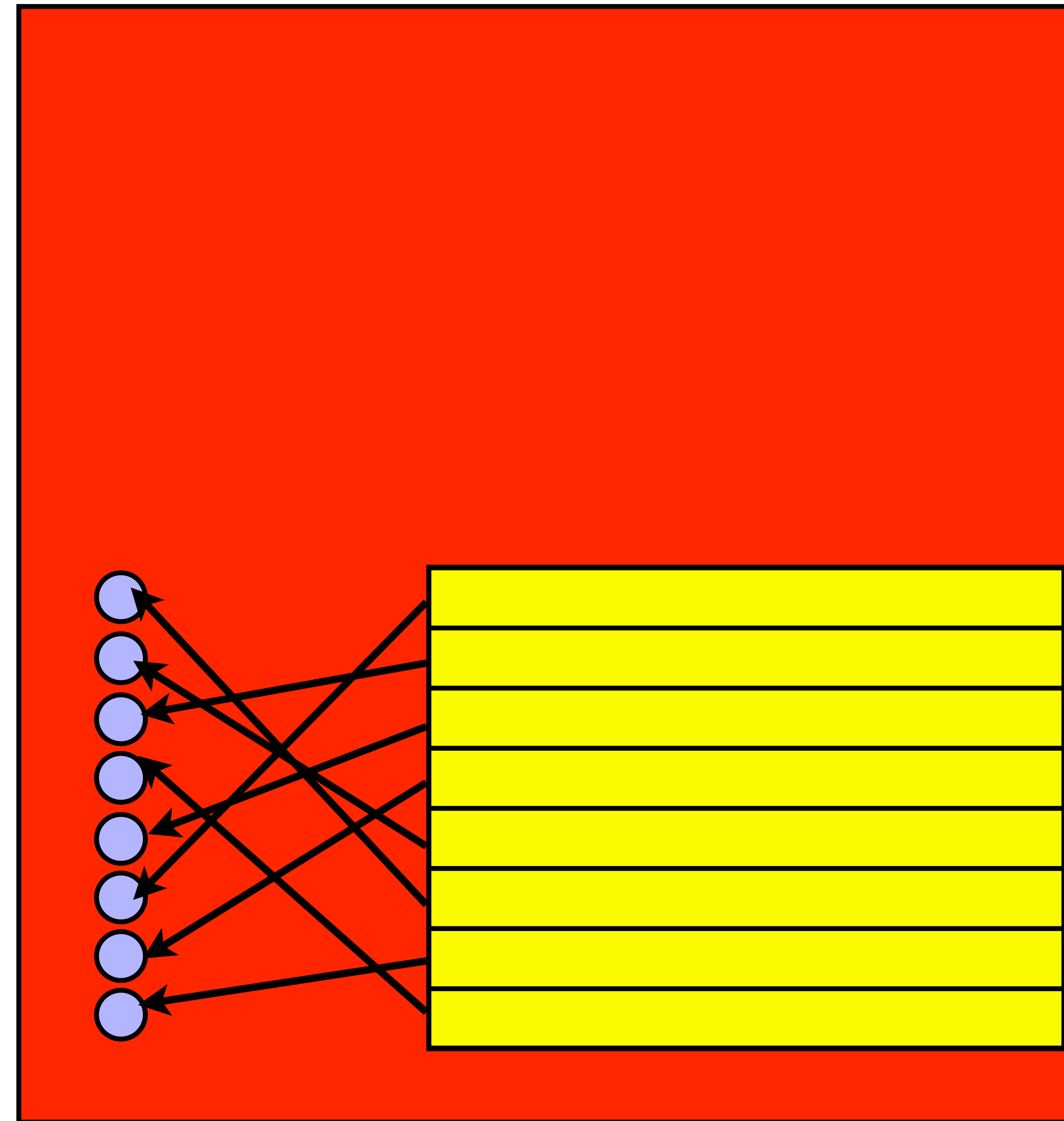
- Each thread in warp accesses different bank: no problem.



SciNet

Shared Memory Bank Conflicts

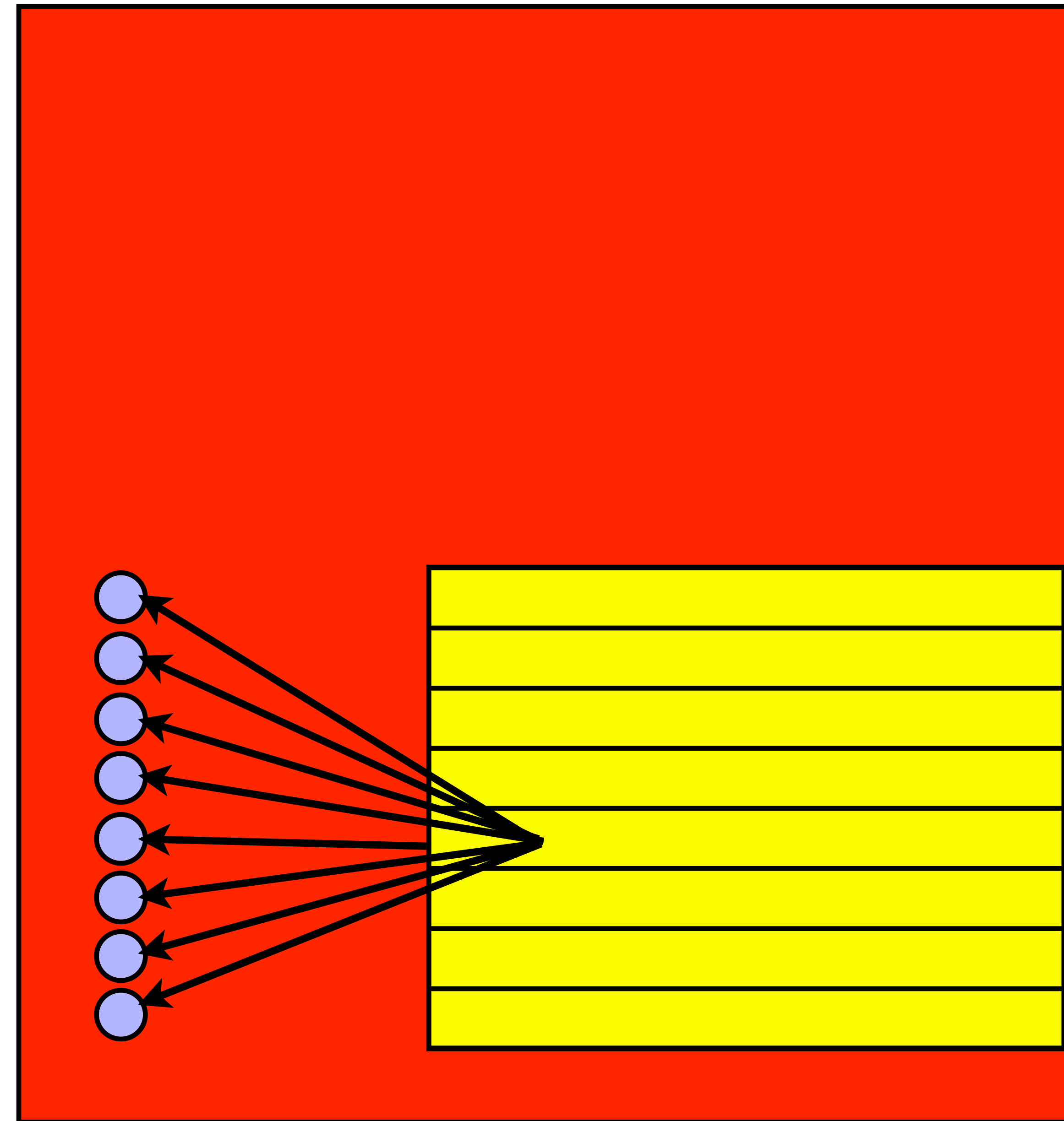
- Each thread in warp accesses different bank: no problem.



SM#1

Shared Memory Bank Conflicts

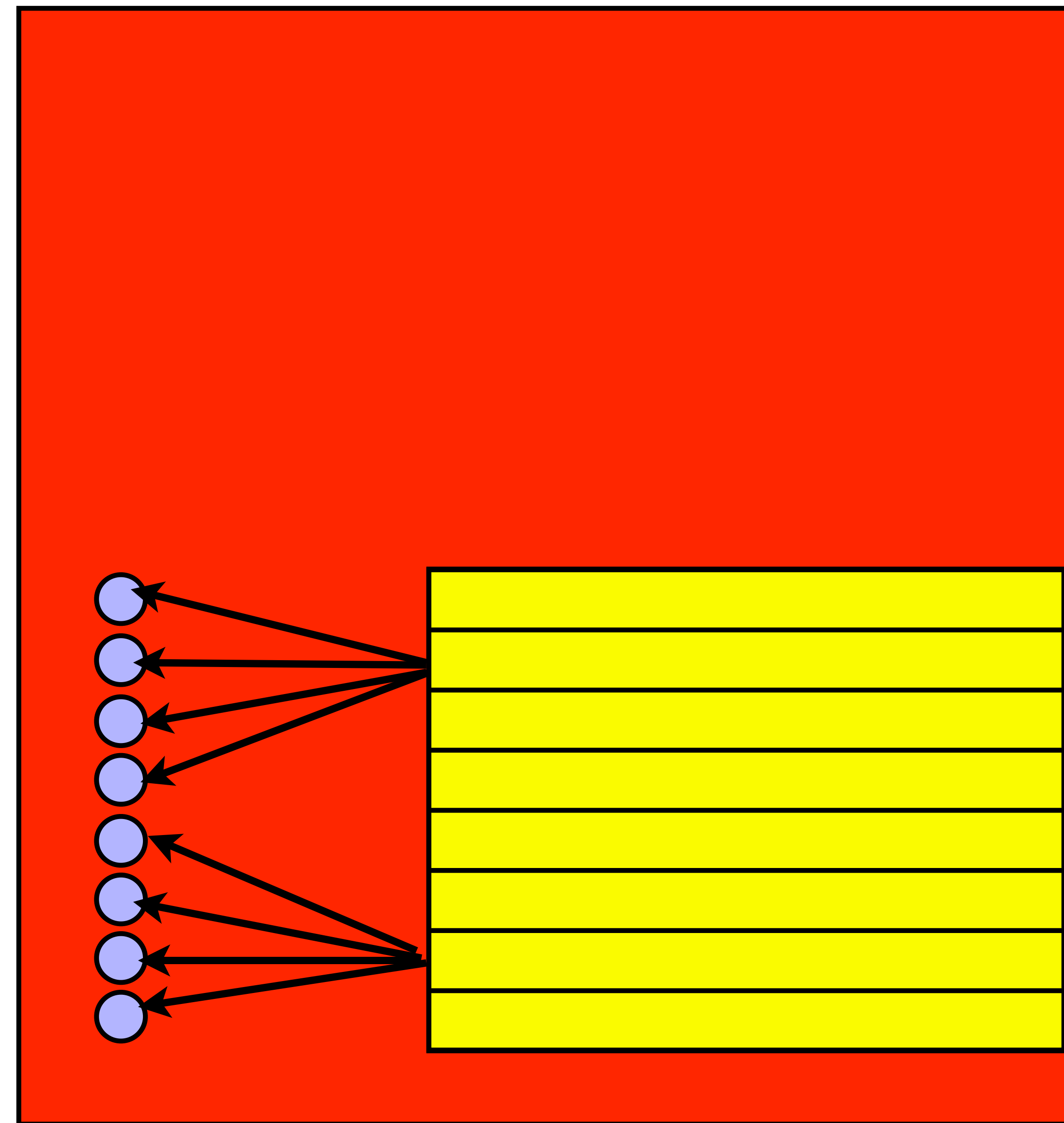
- Each thread in warp accesses different bank: no problem.
- Each thread accesses same one value: ‘broadcast’, no problem.



SM#1

Shared Memory Bank Conflicts

- Each thread in warp accesses different bank: no problem.
- Each thread accesses same one value: 'broadcast', no problem.
- Multiple threads need data from same bank: conflict. Accesses are serialized.



SciNet

Shared Memory Bank Conflicts

- Imagine 8 banks, and working on an 8xN matrix

Bank 0	Bank 1	Bank 2	Bank 3	Bank 4	Bank 5	Bank 6	Bank 7
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

SciNet

Shared Memory Bank Conflicts

- Imagine 8 banks, and working on an 8xN matrix
- Row operations are great

Bank 0	Bank 1	Bank 2	Bank 3	Bank 4	Bank 5	Bank 6	Bank 7
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

SciNet

Shared Memory Bank Conflicts

- Imagine 8 banks, and working on an 8xN matrix
- Row operations are great
- Column operations maximally bad

Bank 0	Bank 1	Bank 2	Bank 3	Bank 4	Bank 5	Bank 6	Bank 7
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

SciNet

Shared Memory Bank Conflicts

- Imagine 8 banks, and working on an 8xN matrix
- Row operations are great
- Column operations maximally bad
- Solutions
  - Row ops if possible

Bank 0	Bank 1	Bank 2	Bank 3	Bank 4	Bank 5	Bank 6	Bank 7
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

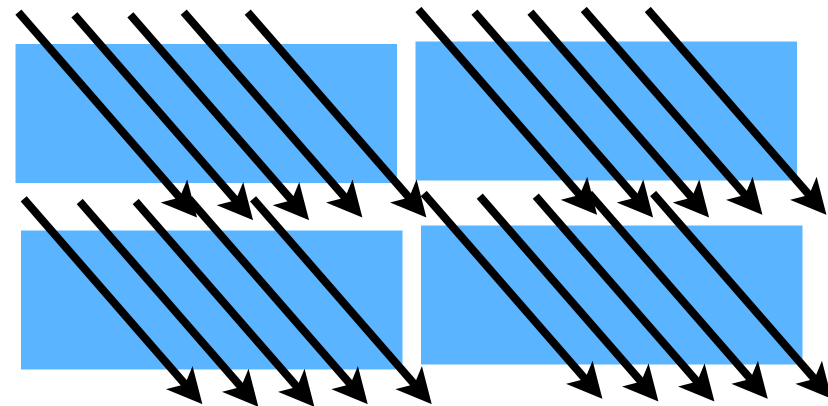
Shared Memory Bank Conflicts

- Imagine 8 banks, and working on an 8xN matrix
- Row operations are great
- Column operations maximally bad
- Solutions
  - Row ops if possible
  - Pad matrix with extra column to stride across banks

Bank 0	Bank 1	Bank 2	Bank 3	Bank 4	Bank 5	Bank 6	Bank 7
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Warps in multi-d blocks

- Easy to see how warps are assigned in 1-d block:
  - First 32 = warp0
  - Next 32 = warp1..
- How done in 2d block?
- C ordering: x first, then y
- blockDim.x = 32:
  - warp 0 : blockDim.y = 0
  - warp 1: blockDim.y = 1..



SciNet

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

    extern __shared__ float shared_data[];

    int loci = threadIdx.x;
    int locj = threadIdx.y;
    int tilesize = blockDim.x;
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    int j = threadIdx.y + blockIdx.y*blockDim.y;
    int k;
    int blockk;

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

    double sum;
    if (i<n && j<n) {
        sum = 0.;
        for (blockk=0; blockk<gridDim.x; blockk++) {
            /* read in shared data */
            atile[loci*tilesize + locj] = ad[(tilesize*bx+loci)*n + (tilesize*blockk+locj)];
            btile[loci*tilesize + locj] = bd[(tilesize*blockk+loci)*n + (tilesize*by+locj)];
            __syncthreads();
            for (k=0; k<tilesize; k++)
                sum += atile[loci*tilesize + k]*btile[k*tilesize + locj];
            __syncthreads();
        }
        cd[i*n + j] = sum;
    }
    return;
}
```

Striding through matrix w/ slow moving index; Massive bank conflicts if blocksize = warpsize

matmult.cu

SciNet

File Session View Options Window Help

Sessions

- Session1
  - Device_0
    - Context_0 [CUDA]
- Session2
  - Device_0
    - Context_0 [CUDA]
- Session3
  - Device_0

Profiler Output | Summary Table

	Method	#Calls	GPU time	%GPU time	warp serialize
1	cuda_sgemm_shared	1	112289	63.09	58021046
2	cuda_sgemm_shared_transpose	1	53739.4	30.19	0
3	memcpyHtoD	4	6673.89	3.74	
4	memcpyDtoH	2	5268.99	2.96	

blocksize = 32
= warpsize

```
marten$ ./matmult --matsize=1536 --nblocks=48
Matrix size = 1536, Number of blocks = 48.
CPU  time = 29466.5 millisec, GFLOPS=0.245966
GPU  time = 522.71 millisec, GFLOPS=13.865733, diff = 0.000000.
GPU2 time = 128.905 millisec, GFLOPS=56.225572, diff = 0.000000.
```

4x performance

SciNet

Memory structure informs block sizes:

- By choosing block size in such a way to maximize global, shared memory bandwidth and preloading data into shared, can extract significant performance
- Get your code working first, then use these considerations to get them working fast

```
$ ./matmult --matsize=1536 --nblocks=24
Matrix size = 1536, Number of blocks = 24.
CPU  time = 29467.4 millisec, GFLOPS=0.245958
GPU  time = 8.203 millisec, GFLOPS=883.549593, diff = 0.000000.
GPU2 time = 8.122 millisec, GFLOPS=892.361156, diff = 0.000000.
```

- Use tuned code where available (this is still much slower than CUBLAS, MAGMA!)

SciNet