An Introduction to GPGPU with CUDA

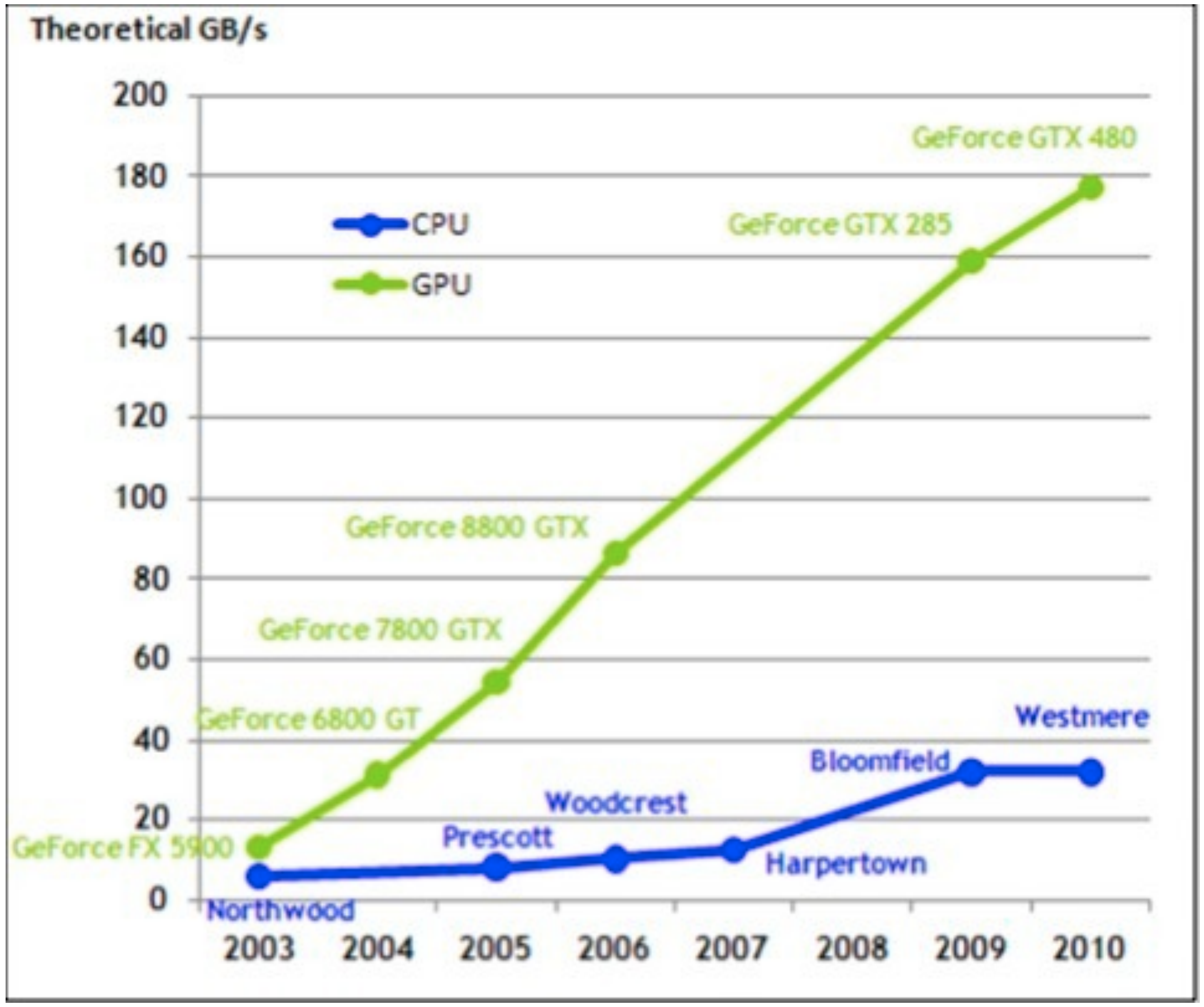
Aug 2011

Upcoming GPU events:

- GPGPU Research Workshop - TB
- Monthly cross-campus GPGPU meetings - TBA
- ECE Graduate GPGPU course - Spring 2012
- Astronomy/Physics GPGPU minicourse/modular course - Spring 2012
- https://support.scinet.utoronto.ca/courses
- https://support.scinet.utoronto.ca/mailman/listinfo/scinet-gpgpu

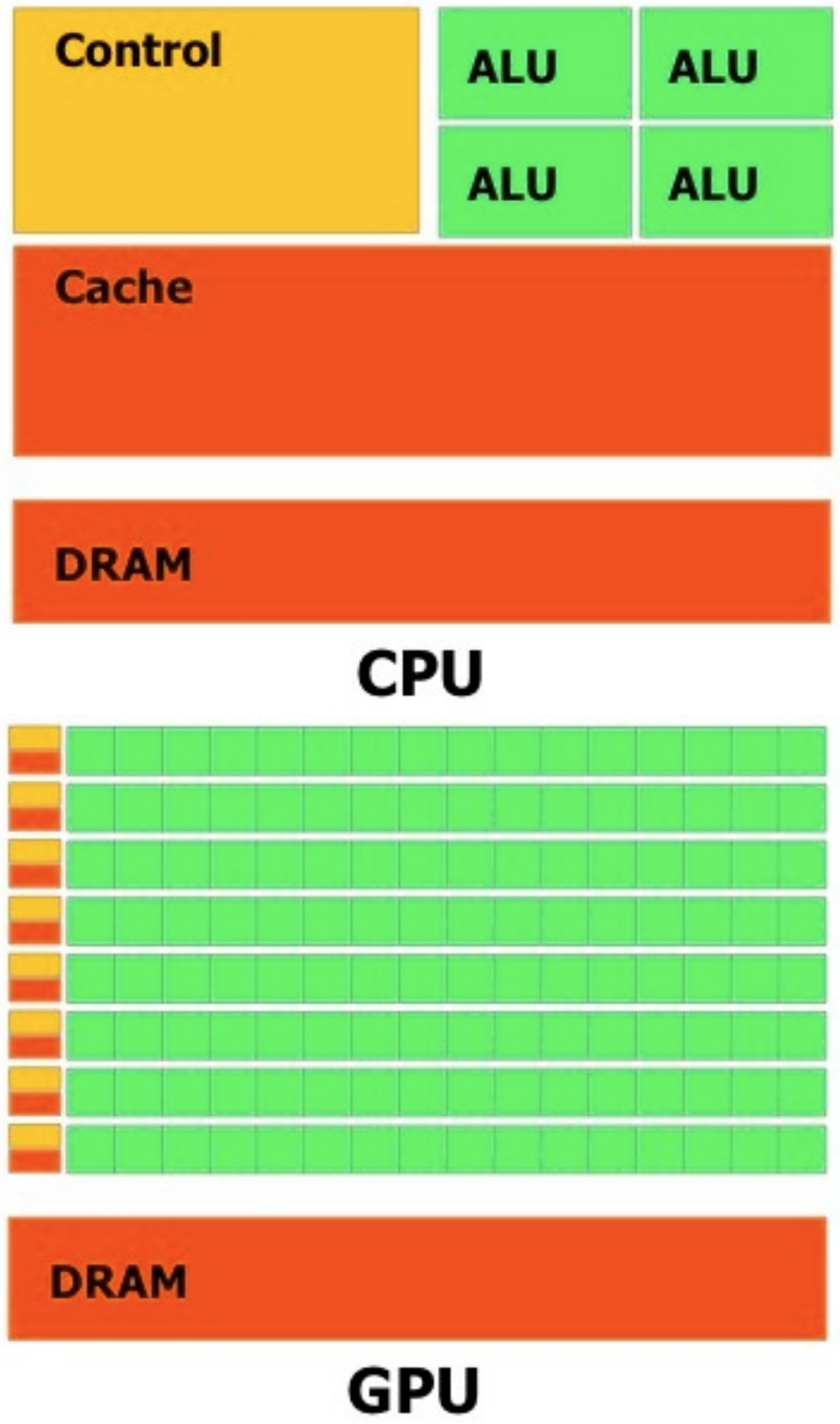
Your graphics card is probably faster than your computer.

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



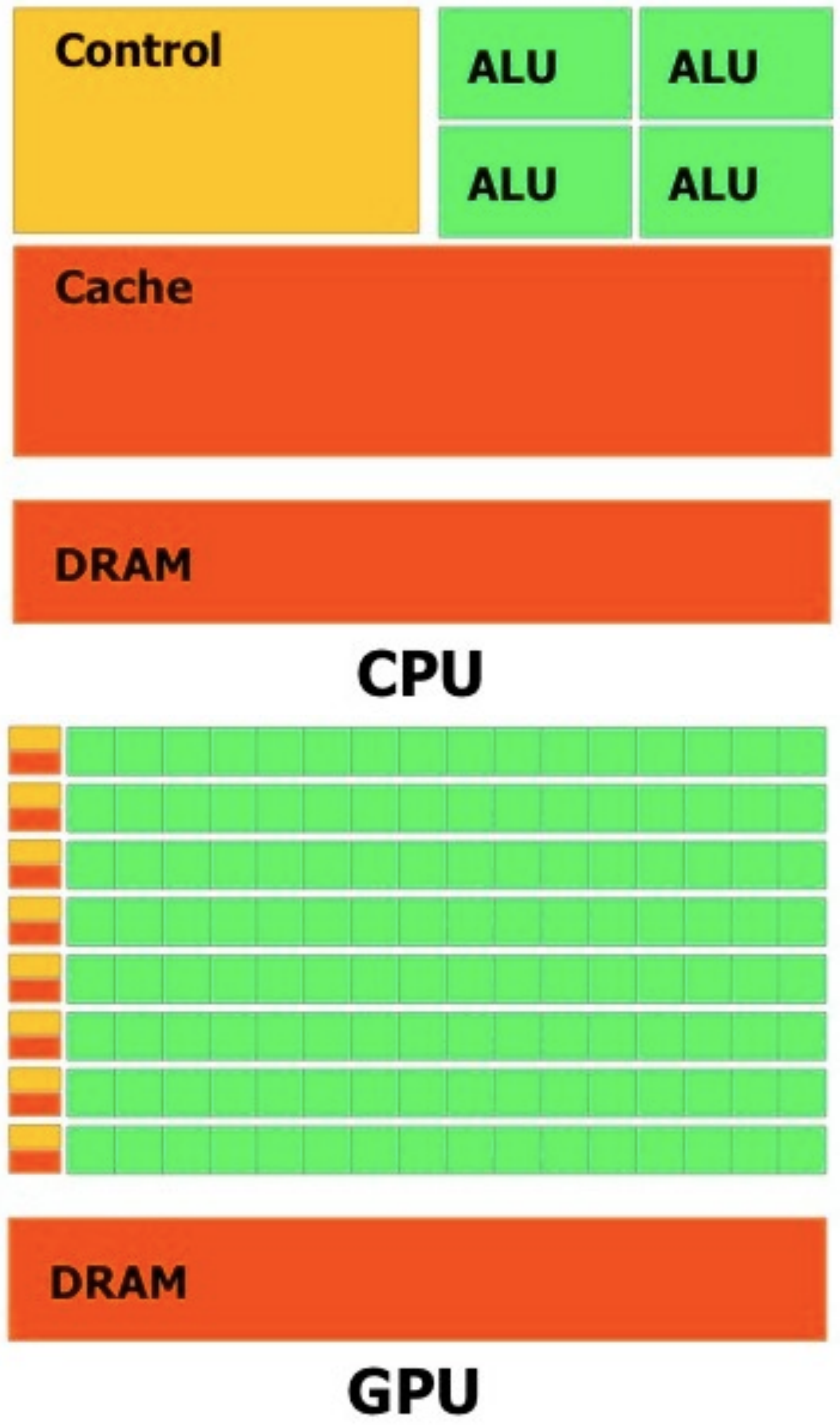
...but it's not magic

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



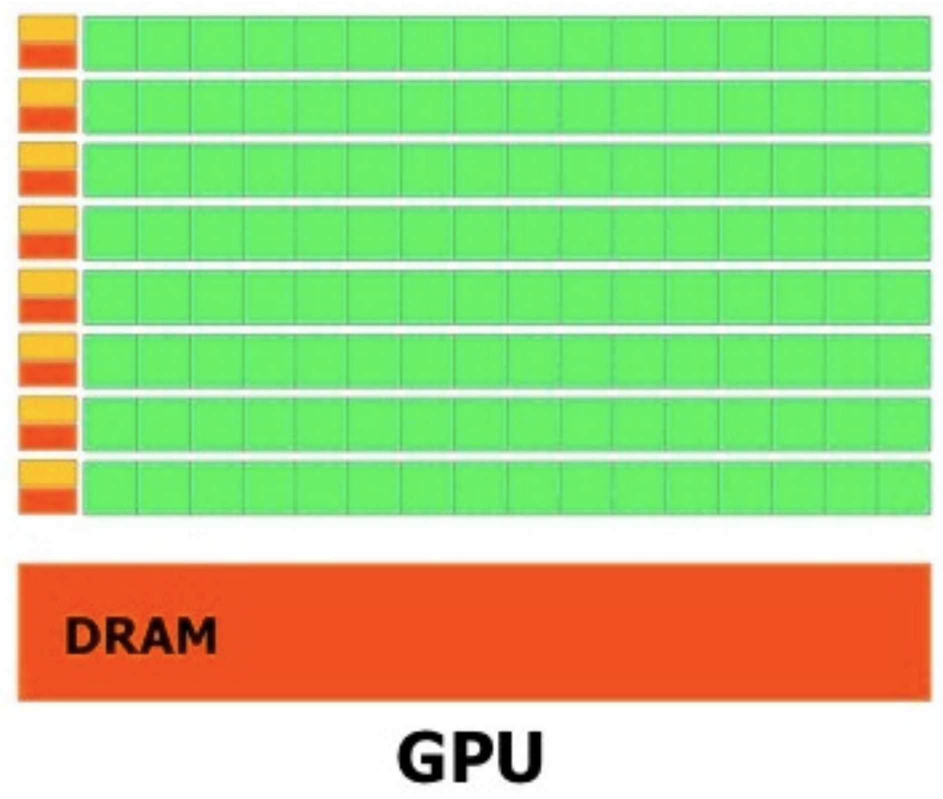
...but it's not magic

- All cores in a "multiprocessor unit" have same control, cache
- Act in lock step
- Do same computations on different data
- "Data parallel"
- Very small cache (48KB/SM)



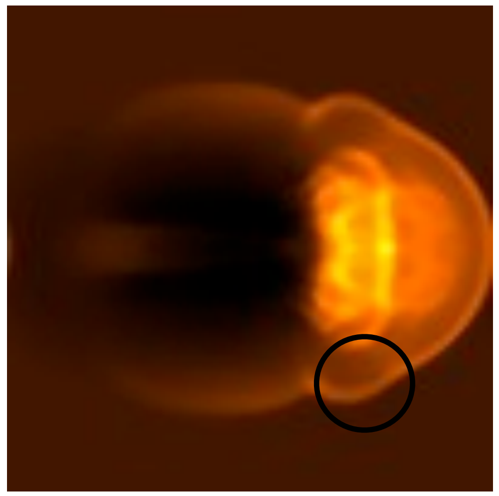
If it works, it's great..

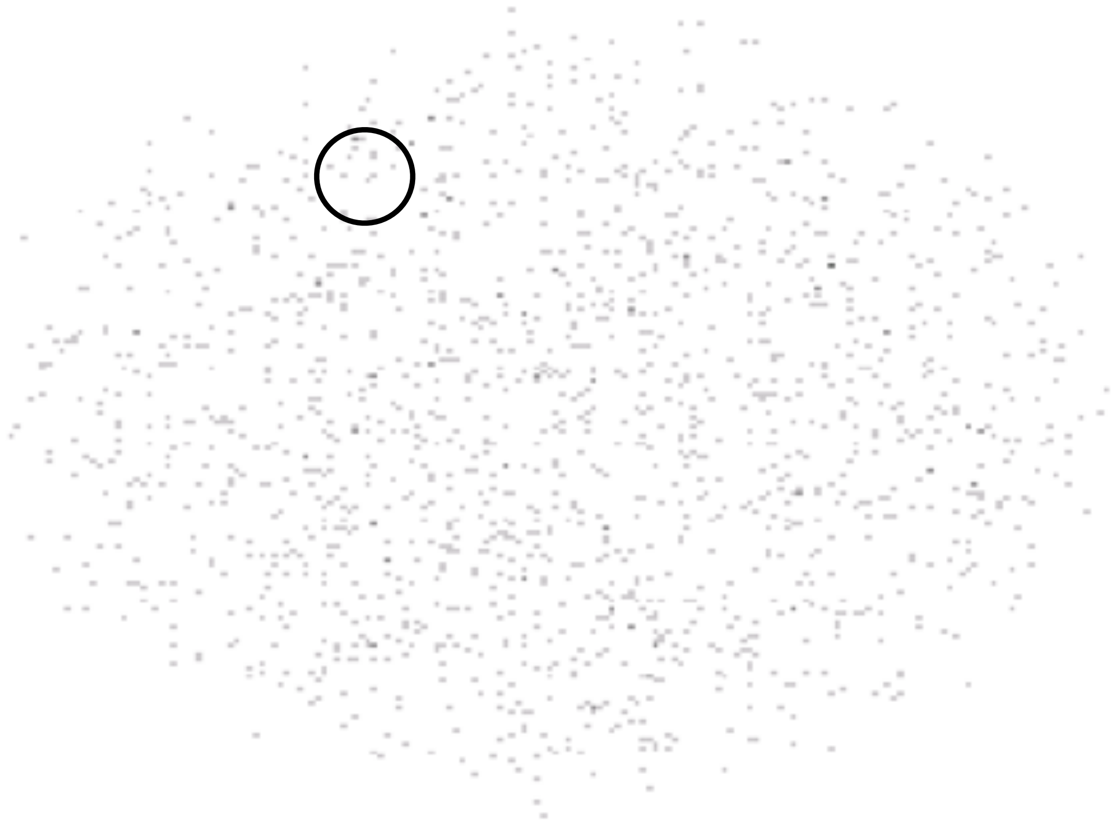
- GPU: ~448 compute cores, into ~14 streaming multiprocessors (SM)
- ~32 threads operate at once



..and it often does.

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



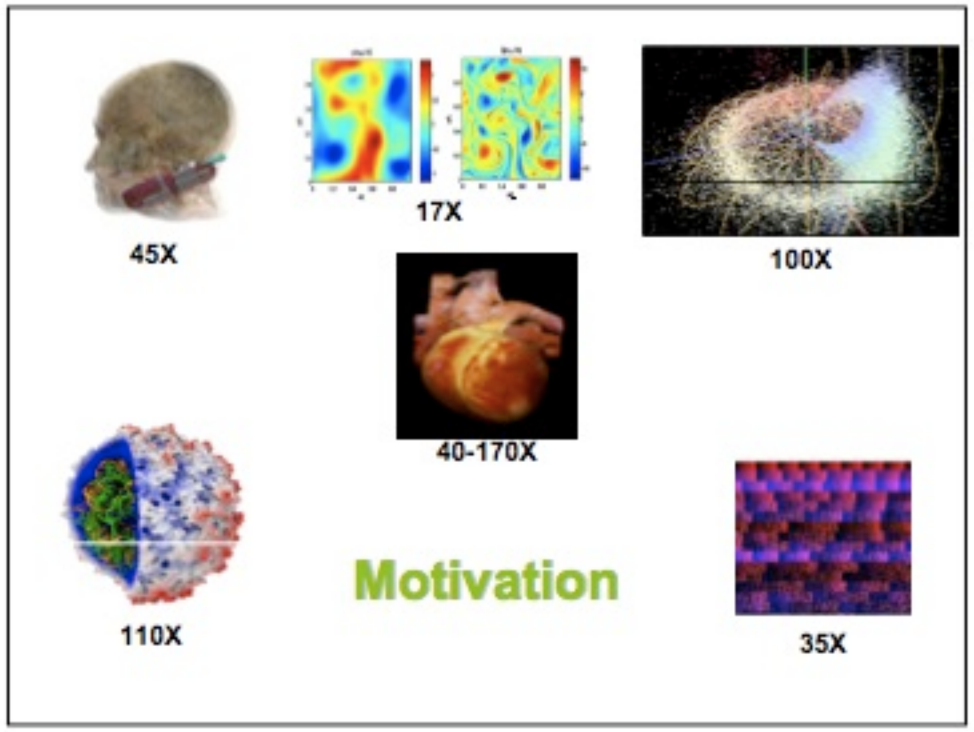


What we'll be covering

- Plan - have you leave being able to start developing simple (single GPU) codes in CUDA
- Know where to look for libraries, development tools
- Know what to think about for more advanced applications

Why CUDA?

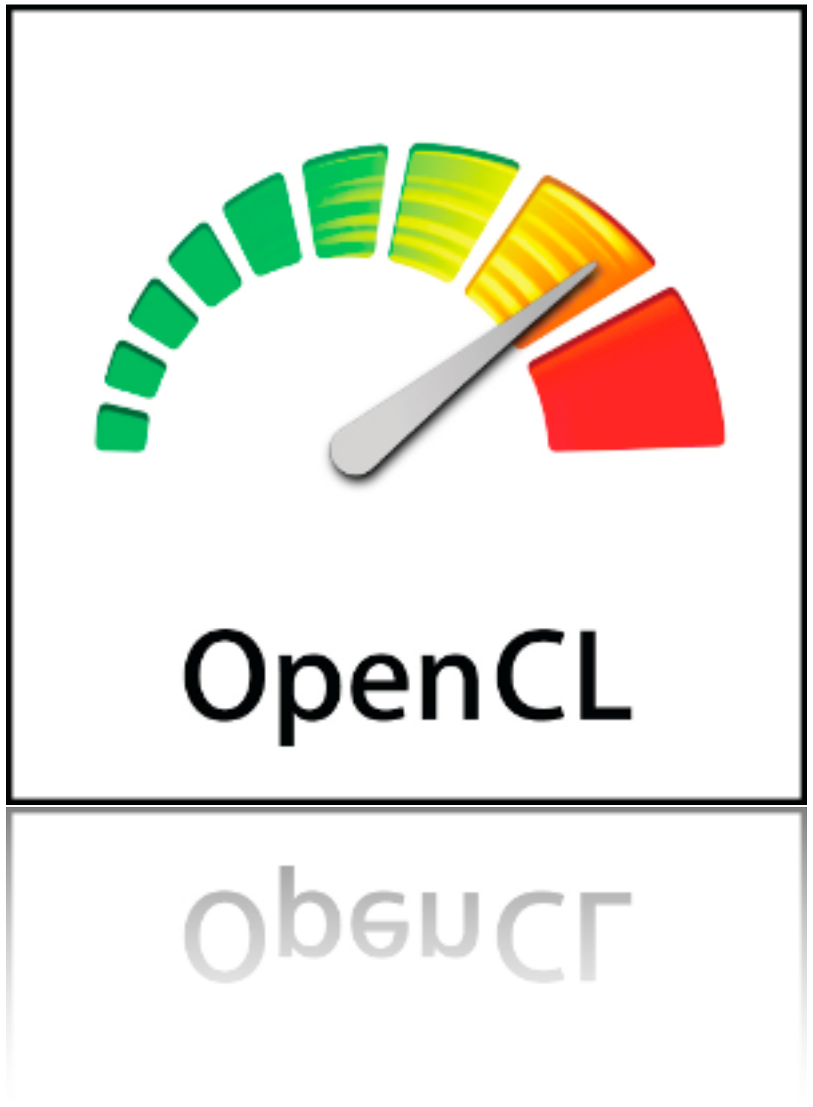
- GPGPU used to be pretty bad; put array in as 'textures', have each point in your grid be a vertex that maps the texture...
- Much better now: CUDA (NVidia), OpenCL (NVidia, Apple,AMD)...



NVidia SC2007 tutorial slides

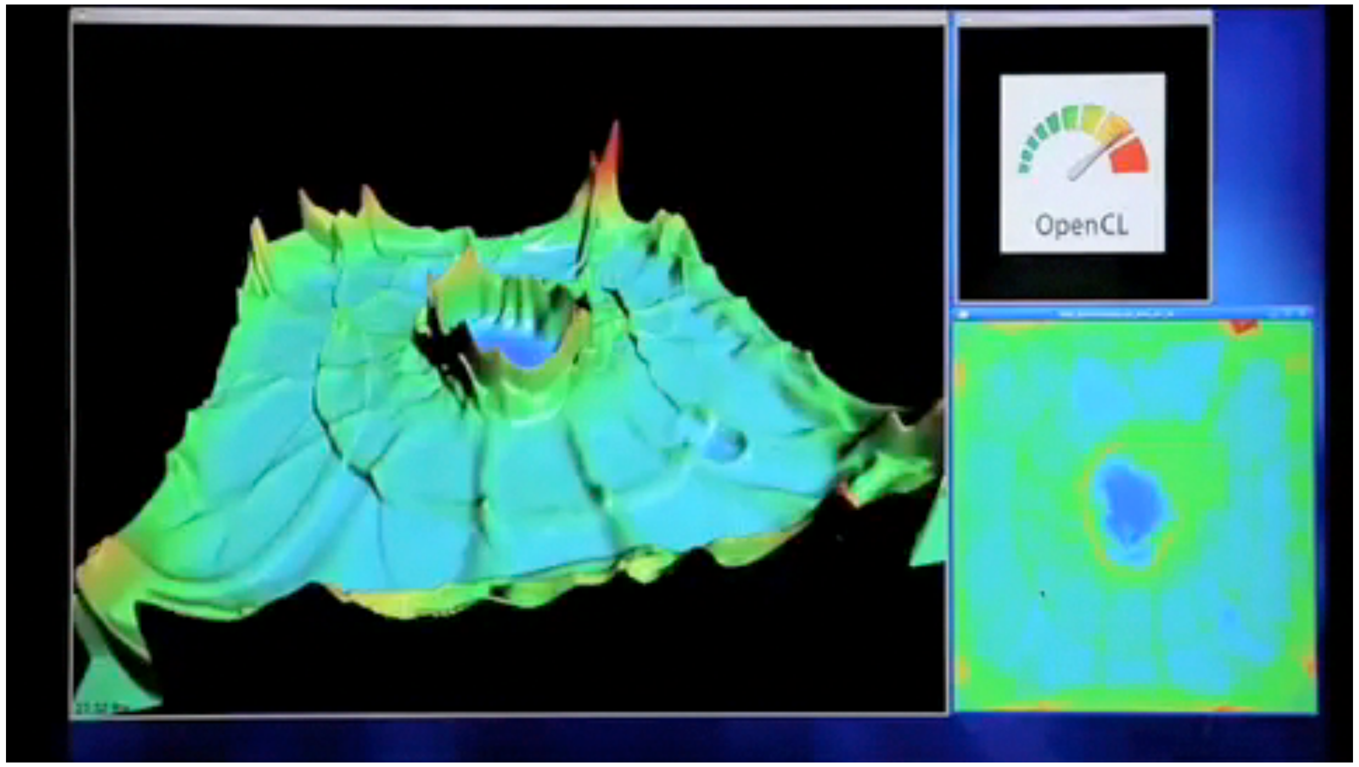
Open standard

- Driven by Apple (comes standard in Snow Leopard, Lion)
- NVIDIA, AMD, Intel, IBM (Cell)
- Exposes a consistent, GPU-like interface to any multicore system



Heterogeneous, Open

- Can work with various hardware
- IBM Cell, AMD processors, ATI cards, NVidia cards, Intel processors
- Multi- and Many- core
- SC09 demo: parallel CFD running on all of the above at once in *same program*, using MPI to tie them together



```
__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;
}
```

CUDA

```
__kernel void opencl_saxpb(__global const float *x,
                                   const float a, const float b,
                           __global       float *y)
{
     int i = get_global_id(0);
     if (i < get_global_size(0) )
         y[i] = a*x[i] + b;
}
```

OpenCL

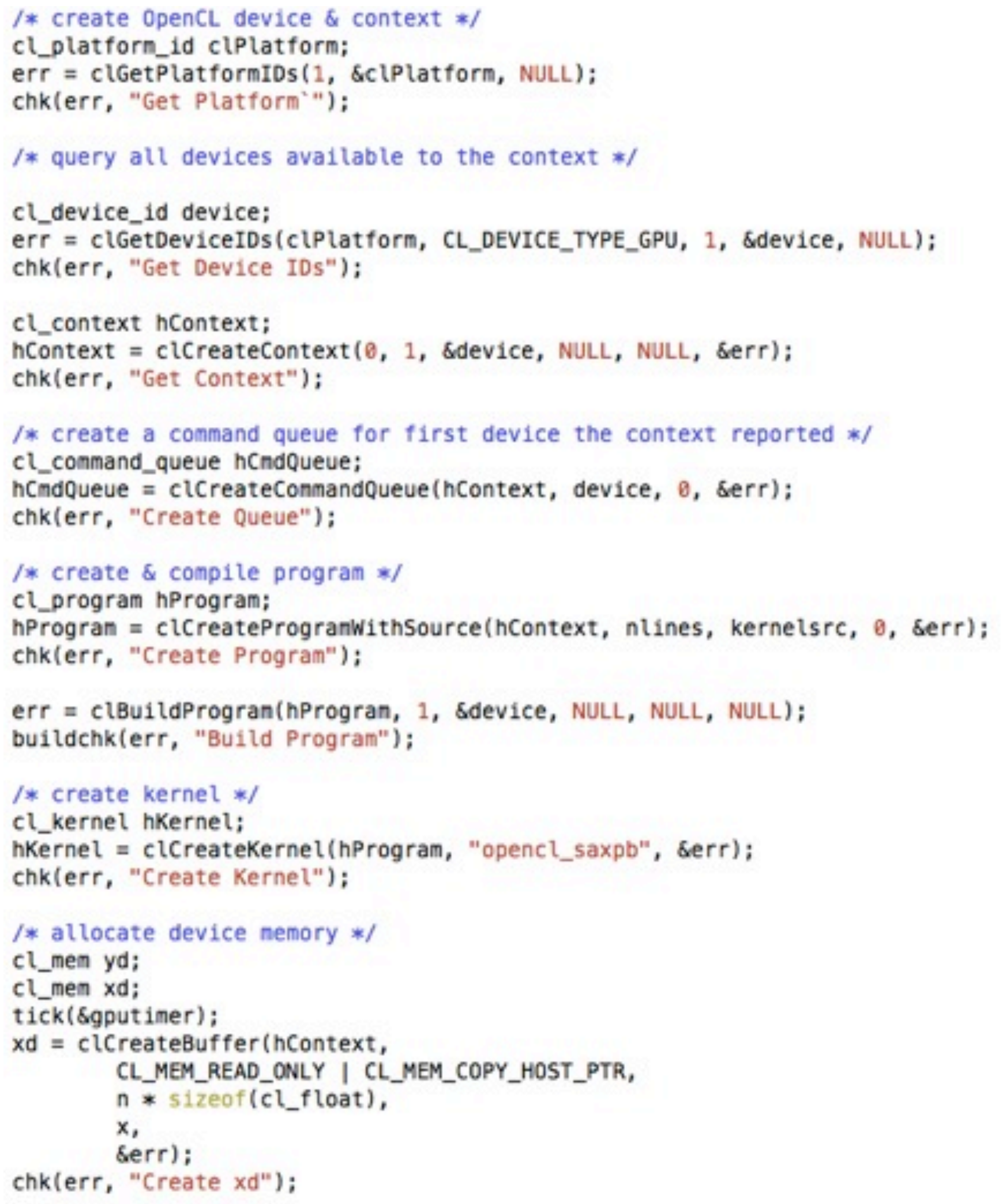
CUDA vs OpenCL kernel code

- Since maps to similar hardware, basic concepts the same
- Some terminology changes; some better, some worse.
- Kernels not really that different.

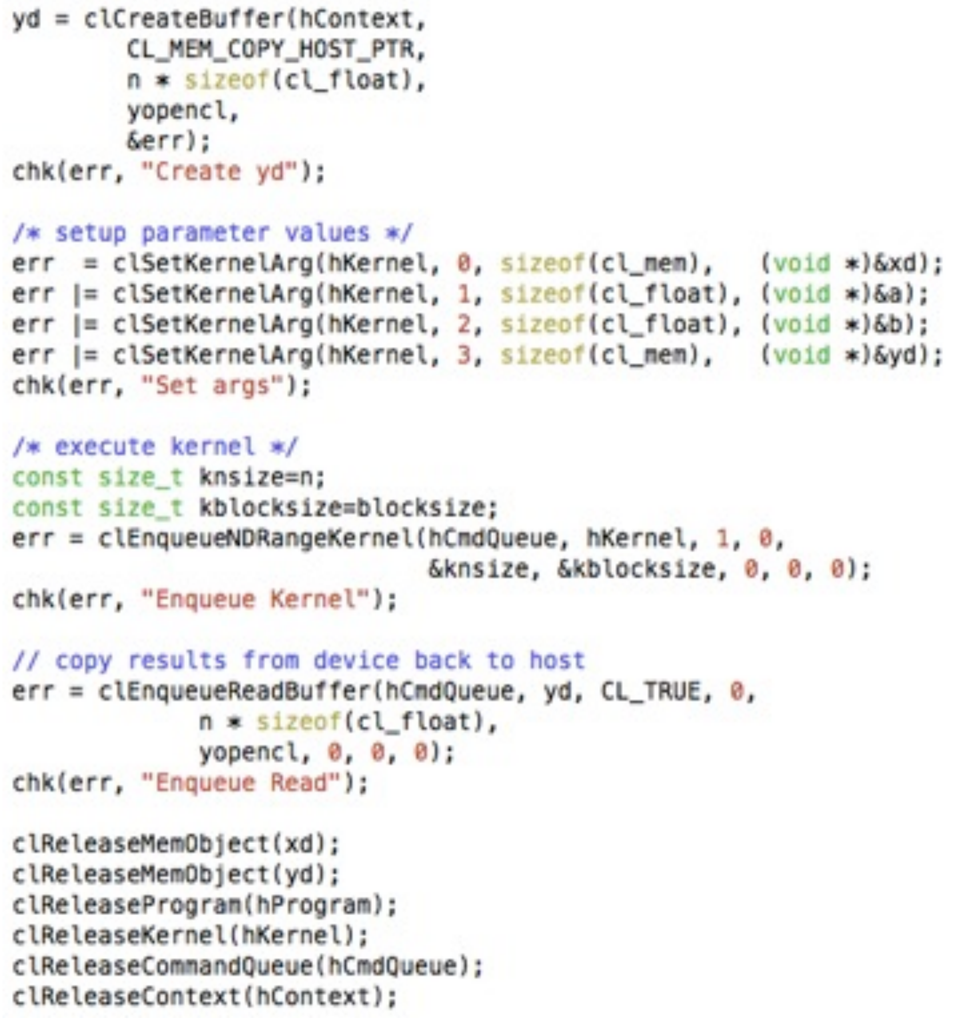
CUDA	OpenCL
__global__	__kernel
__device__ (function)	
__constant__	__constant
__device__ (mem)	__global
__shared__	__local
Local Mem	Private Mem
__syncthreads()	barrier()

CUDA

```
/* 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) );

blocksize = (n+nblocks-1)/nblocks;
for (i=0; i<niters; i++) {
    cuda_saxpb<<<nblocks, blocksize>>>(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) );
```



```c
/* create OpenCL device & context */
cl_platform_id clPlatform;
err = clGetPlatformIDs(1, &clPlatform, NULL);
chk(err, "Get Platform`");

/* query all devices available to the context */

cl_device_id device;
err = clGetDeviceIDs(clPlatform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
chk(err, "Get Device IDs");

cl_context hContext;
hContext = clCreateContext(0, 1, &device, NULL, NULL, &err);
chk(err, "Get Context");

/* create a command queue for first device the context reported */
cl_command_queue hCmdQueue;
hCmdQueue = clCreateCommandQueue(hContext, device, 0, &err);
chk(err, "Create Queue");

/* create & compile program */
cl_program hProgram;
hProgram = clCreateProgramWithSource(hContext, nlines, kernelsrc, 0, &err);
chk(err, "Create Program");

err = clBuildProgram(hProgram, 1, &device, NULL, NULL, NULL);
buildchk(err, "Build Program");

/* create kernel */
cl_kernel hKernel;
hKernel = clCreateKernel(hProgram, "opencl_saxpb", &err);
chk(err, "Create Kernel");

/* allocate device memory */
cl_mem yd;
cl_mem xd;
tick(&gputimer);
xd = clCreateBuffer(hContext,
        CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
        n * sizeof(cl_float),
        x,
        &err);
chk(err, "Create xd");
```

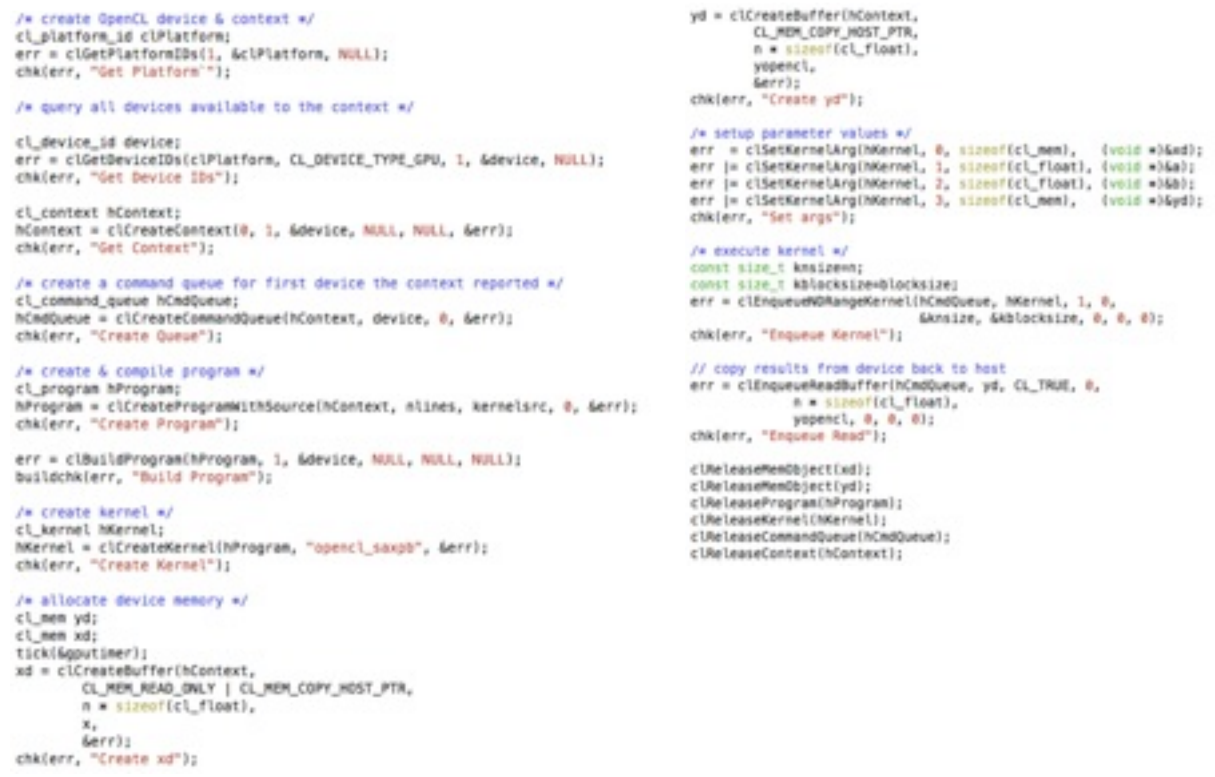


```c
yd = clCreateBuffer(hContext,
        CL_MEM_COPY_HOST_PTR,
        n * sizeof(cl_float),
        yopencl,
        &err);
chk(err, "Create yd");

/* setup parameter values */
err  = clSetKernelArg(hKernel, 0, sizeof(cl_mem),   (void *)&xd);
err |= clSetKernelArg(hKernel, 1, sizeof(cl_float), (void *)&a);
err |= clSetKernelArg(hKernel, 2, sizeof(cl_float), (void *)&b);
err |= clSetKernelArg(hKernel, 3, sizeof(cl_mem),   (void *)&yd);
chk(err, "Set args");

/* execute kernel */
const size_t knsize=n;
const size_t kblocksize=blocksize;
err = clEnqueueNDRangeKernel(hCmdQueue, hKernel, 1, 0,
                            &knsize, &kblocksize, 0, 0, 0);
chk(err, "Enqueue Kernel");

// copy results from device back to host
err = clEnqueueReadBuffer(hCmdQueue, yd, CL_TRUE, 0,
            n * sizeof(cl_float),
            yopencl, 0, 0, 0);
chk(err, "Enqueue Read");

clReleaseMemObject(xd);
clReleaseMemObject(yd);
clReleaseProgram(hProgram);
clReleaseKernel(hKernel);
clReleaseCommandQueue(hCmdQueue);
clReleaseContext(hContext);
```

OpenCL

Why CUDA?

- Doesn't really make a difference.
- Kernels (where all the hard work goes) are almost identical.
- Boilerplate, which is straightforward (copy memory, launch kernel) is different but not all that important
- CUDA makes easy things easy, so we'll use that.
- Both are about the same for more complicated situations (multi-GPU, etc)



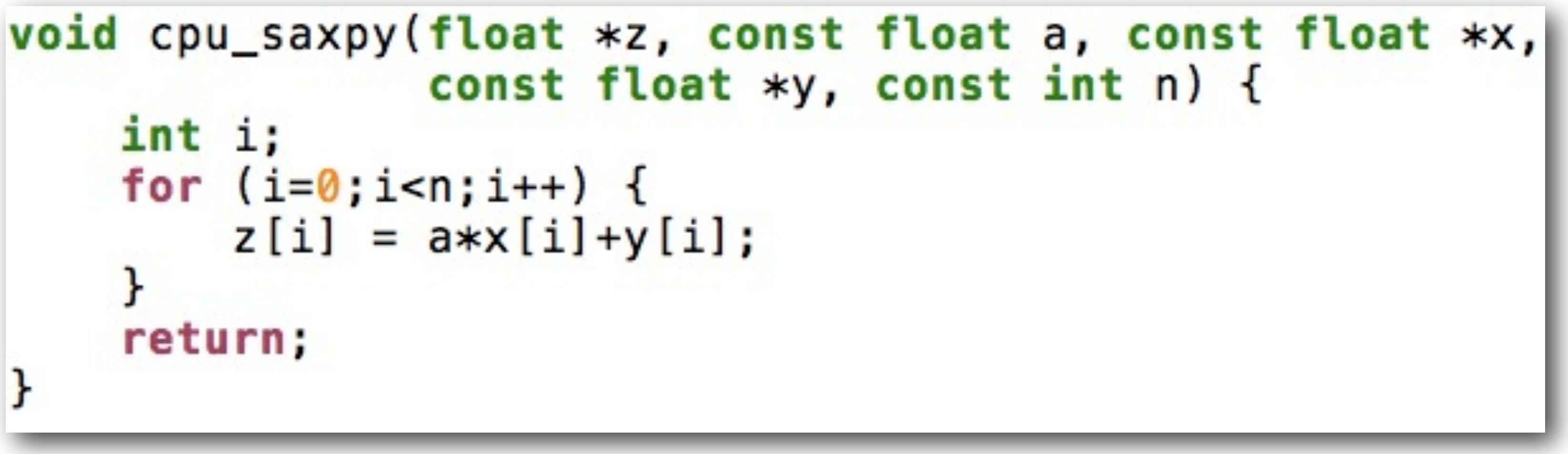
```
/* create OpenCL device & context */
cl_platform_id clPlatform;
err = clGetPlatformIDs(1, &clPlatform, NULL);
chk(err, "Get Platform`");

/* query all devices available to the context */

cl_device_id device;
err = clGetDeviceIDs(clPlatform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
chk(err, "Get Device IDs");

cl_context hContext;
hContext = clCreateContext(0, 1, &device, NULL, NULL, &err);
chk(err, "Get Context");

/* create a command queue for first device the context reported */
cl_command_queue hCmdQueue;
hCmdQueue = clCreateCommandQueue(hContext, device, 0, &err);
chk(err, "Create Queue");

/* create & compile program */
cl_program hProgram;
hProgram = clCreateProgramWithSource(hContext, nlines, kernelsrc, 0, &err);
chk(err, "Create Program");

err = clBuildProgram(hProgram, 1, &device, NULL, NULL, NULL);
buildchk(err, "Build Program");

/* create kernel */
cl_kernel hKernel;
hKernel = clCreateKernel(hProgram, "opencl_saxpb", &err);
chk(err, "Create Kernel");

/* allocate device memory */
cl_mem yd;
cl_mem xd;
tick(&gputimer);
xd = clCreateBuffer(hContext,
        CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
        n * sizeof(cl_float),
        x,
        &err);
chk(err, "Create xd");
```

```
yd = clCreateBuffer(hContext,
        CL_MEM_COPY_HOST_PTR,
        n * sizeof(cl_float),
        yopencl,
        &err);
chk(err, "Create yd");

/* setup parameter values */
err  = clSetKernelArg(hKernel, 0, sizeof(cl_mem),   (void *)&xd);
err |= clSetKernelArg(hKernel, 1, sizeof(cl_float), (void *)&a);
err |= clSetKernelArg(hKernel, 2, sizeof(cl_float), (void *)&b);
err |= clSetKernelArg(hKernel, 3, sizeof(cl_mem),   (void *)&yd);
chk(err, "Set args");

/* execute kernel */
const size_t knsize=n;
const size_t kblocksize=blocksize;
err = clEnqueueNDRangeKernel(hCmdQueue, hKernel, 1, 0,
                             &knsize, &kblocksize, 0, 0, 0);
chk(err, "Enqueue Kernel");

// copy results from device back to host
err = clEnqueueReadBuffer(hCmdQueue, yd, CL_TRUE, 0,
            n * sizeof(cl_float),
            yopencl, 0, 0, 0);
chk(err, "Enqueue Read");

clReleaseMemObject(xd);
clReleaseMemObject(yd);
clReleaseProgram(hProgram);
clReleaseKernel(hKernel);
clReleaseCommandQueue(hCmdQueue);
clReleaseContext(hContext);
```

Let's get straight to it

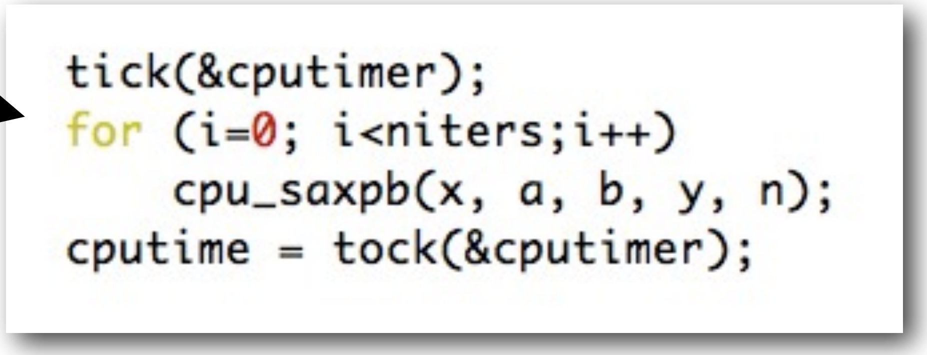
- From login node, ssh to arc01 (devel node of accelerator research cluster)
- `cp -r /scinet/course/intro-gpu/ . ; cd intro-gpu`
- `source setup`
- `cd saxpy`
- `make clean all`
- `./saxpy --help`
- `./saxpy`

$$\vec{z} = \alpha\vec{x} + \vec{y}$$



```
void cpu_saxpy(float *z, const float a, const float *x,
               const float *y, const int n) {
    int i;
    for (i=0;i<n;i++) {
        z[i] = a*x[i]+y[i];
    }
    return;
}
```

(run several times for timing)

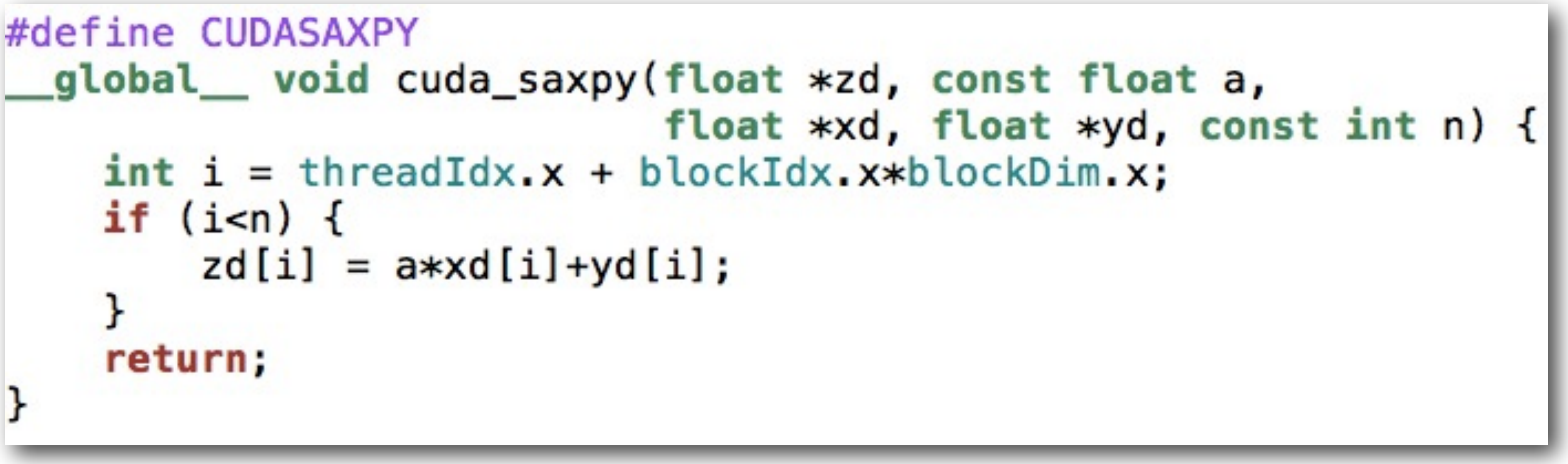


```
tick(&cputimer);
for (i=0; i<niters;i++)
    cpu_saxpb(x, a, b, y, n);
cputime = tock(&cputimer);
```

saxpy.cu

Question: How would we OpenMP this? MPI this?

$\vec{z} = \alpha\vec{x} + \vec{y}$



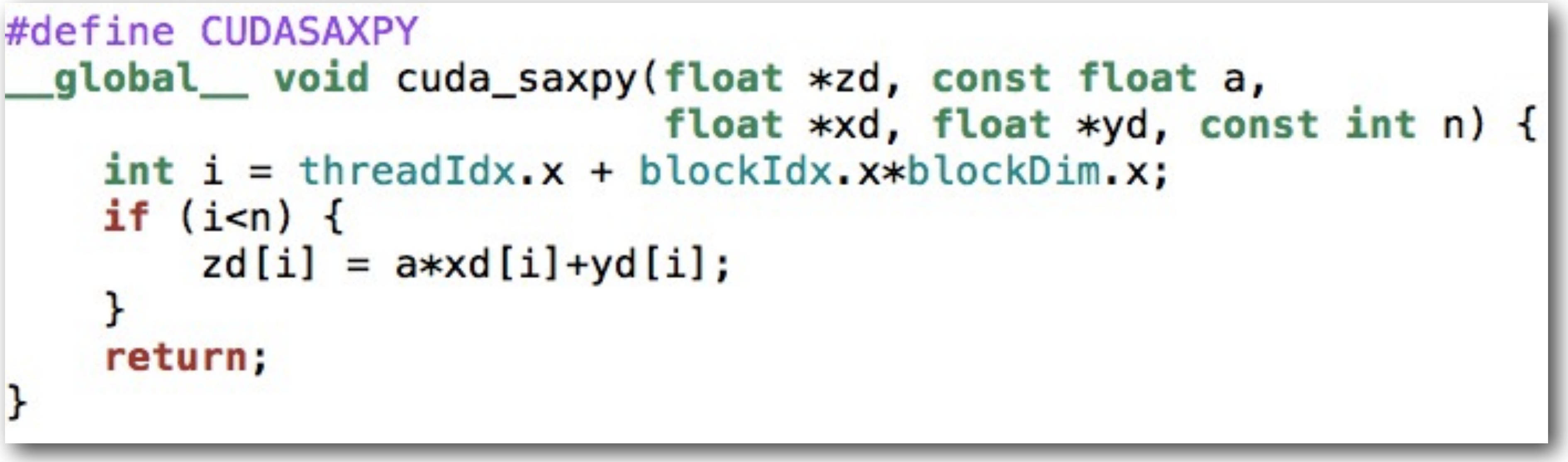
```
#define CUDASAXPY
__global__ void cuda_saxpy(float *zd, const float a,
                           float *xd, float *yd, const int n) {
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    if (i<n) {
        zd[i] = a*xd[i]+yd[i];
    }
    return;
}
```

saxpy.cu

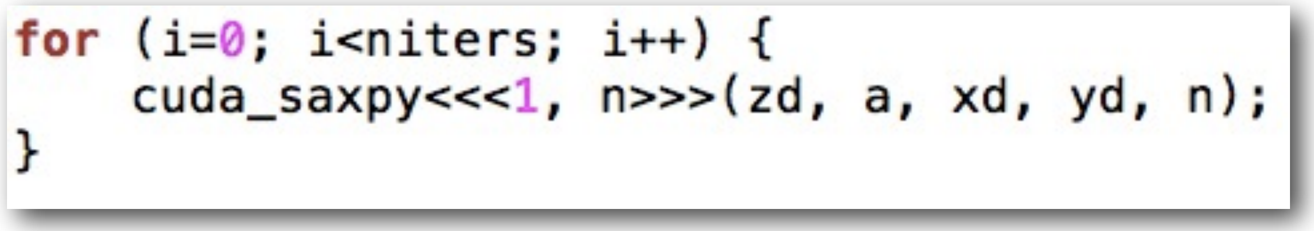
Very fine-grained parallelism.
Each core does one (or few) tasks.

Type "make", and "./saxpy"

$$\vec{z} = \alpha\vec{x} + \vec{y}$$



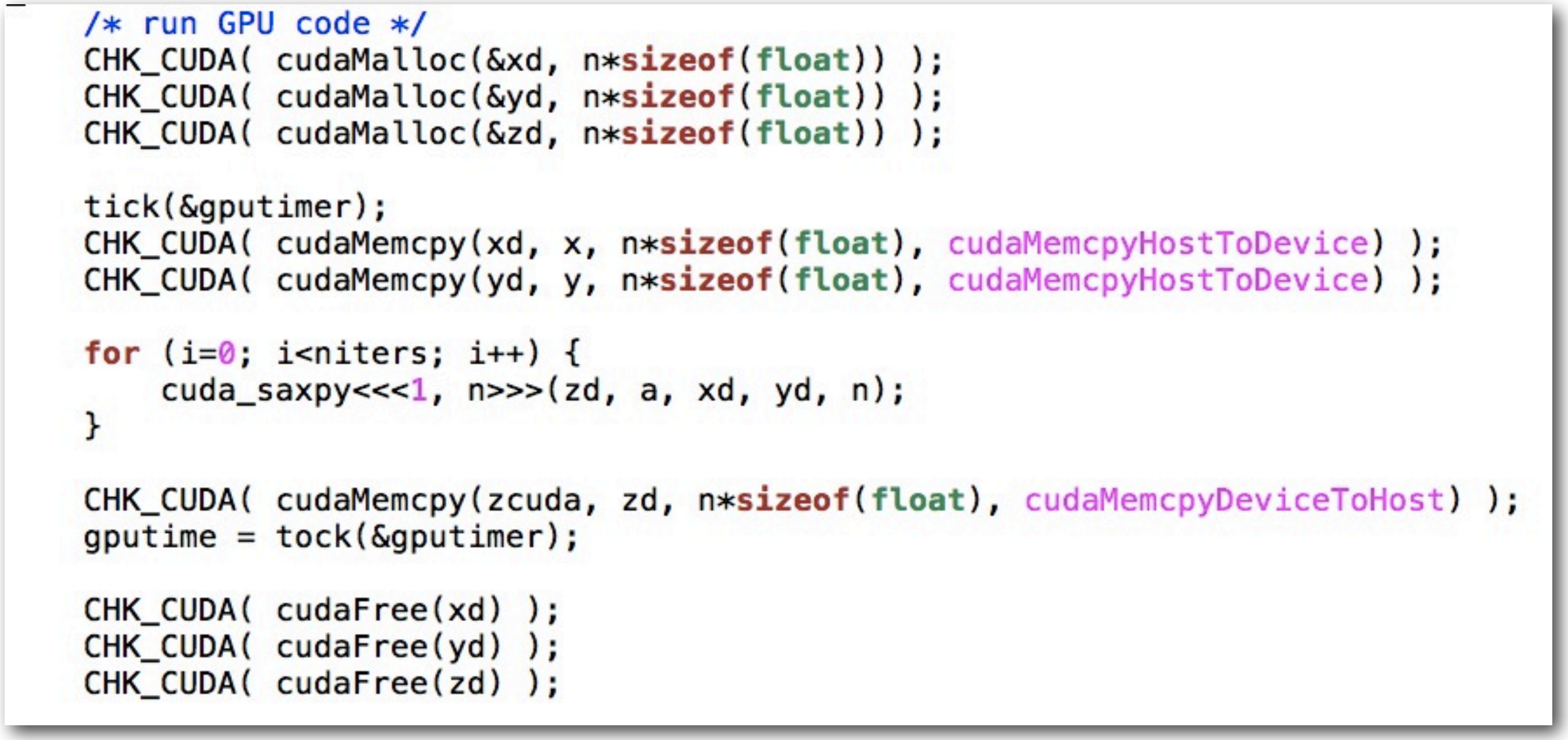
```
#define CUDASAXPY
__global__ void cuda_saxpy(float *zd, const float a,
                           float *xd, float *yd, const int n) {
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    if (i<n) {
        zd[i] = a*xd[i]+yd[i];
    }
    return;
}
```



```
for (i=0; i<niters; i++) {
    cuda_saxpy<<<1, n>>>(zd, a, xd, yd, n);
}
```

saxpy.cu

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



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

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

for (i=0; i<niters; i++) {
    cuda_saxpy<<<1, n>>>(zd, a, xd, yd, n);
}

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

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

saxpy.cu

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

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

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

    for (i=0; i<niters; i++) {
        cuda_saxpy<<<1, n>>>(zd, a, xd, yd, n);
    }

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

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

saxpy.cu

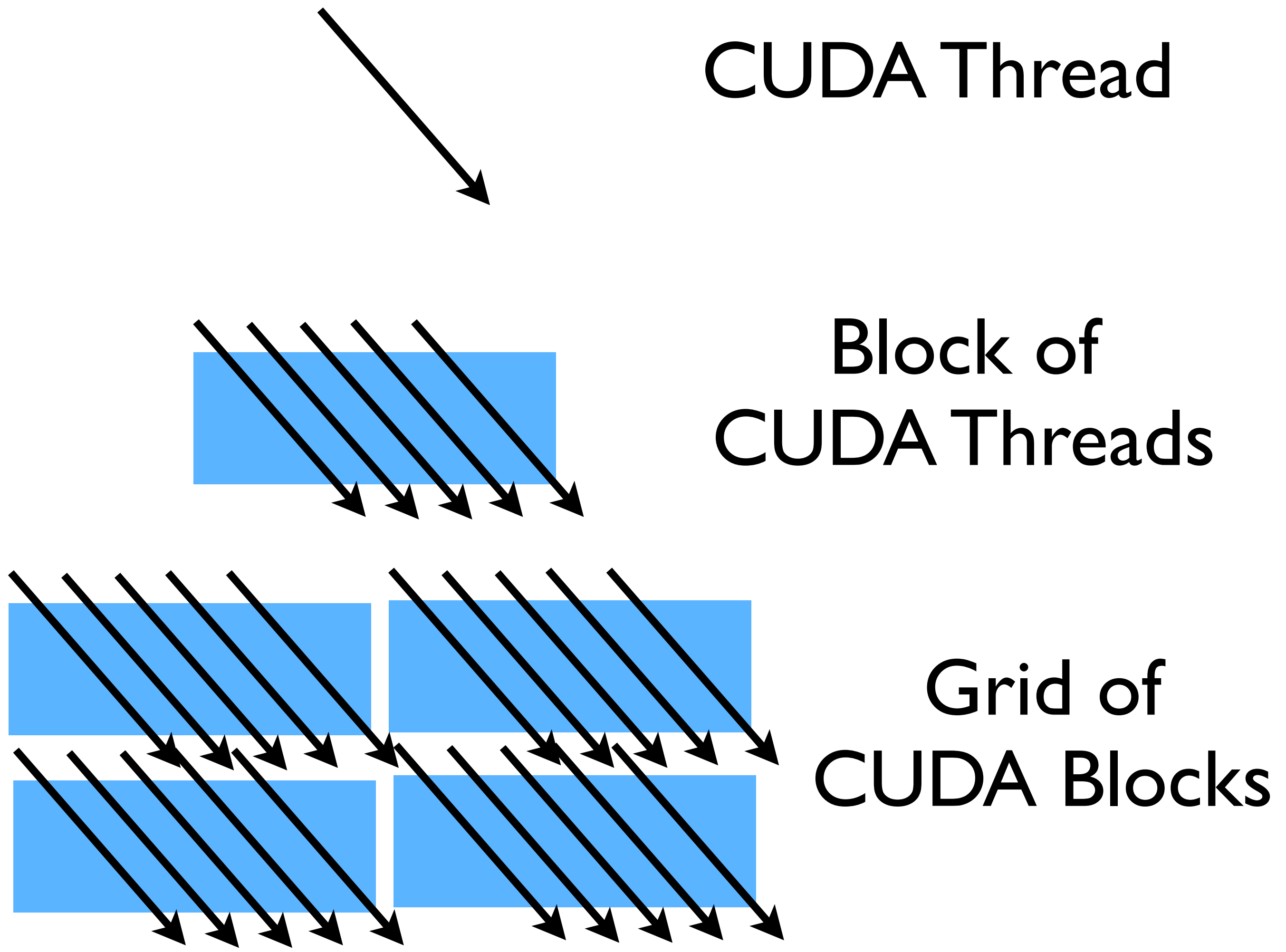
```
__global__ void cuda_saxpy(float *zd, const float a,
                           float *xd, float *yd, const int n) {
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    if (i<n) {
        zd[i] = a*xd[i]+yd[i];
    }
    return;
}
```

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 ./saxpy --nvals=200 ? ./saxpy --nvals=2048 ?

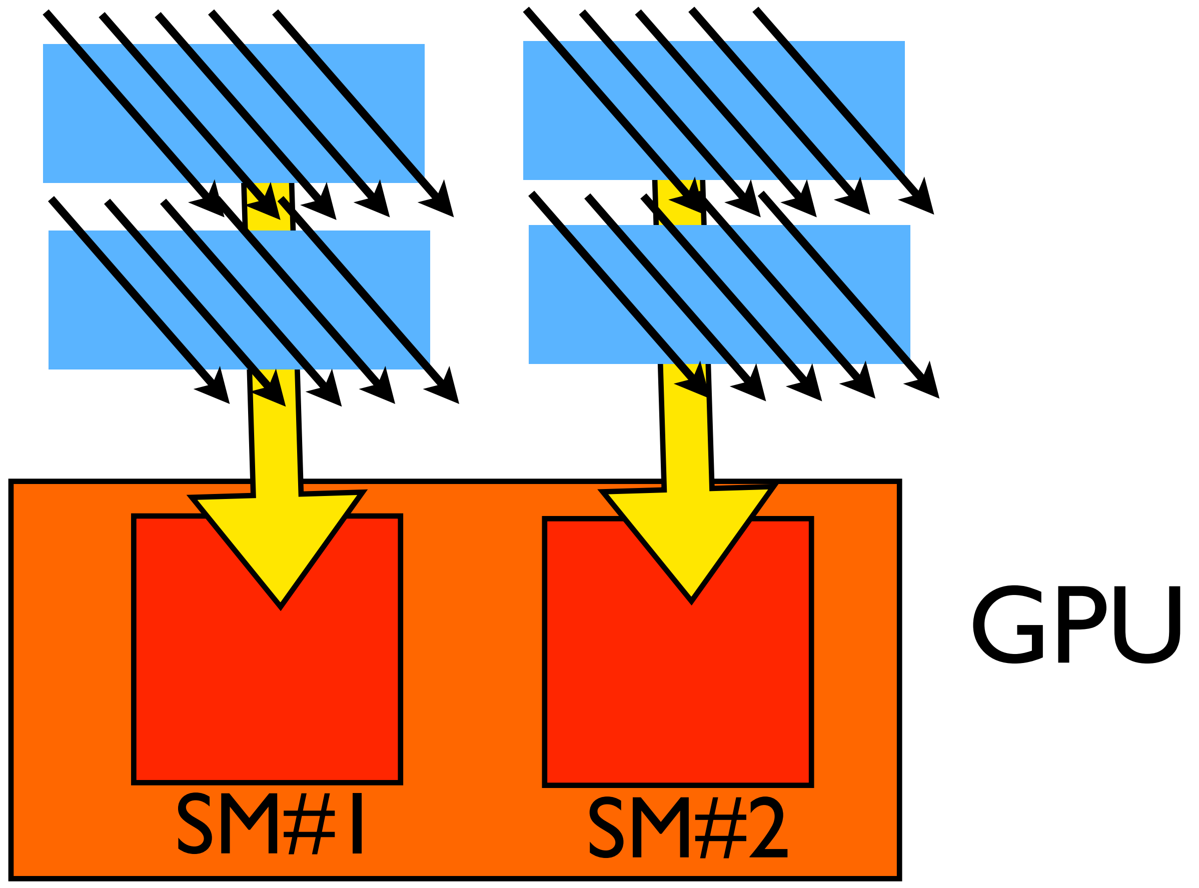
Threads, Blocks, Grids

- 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



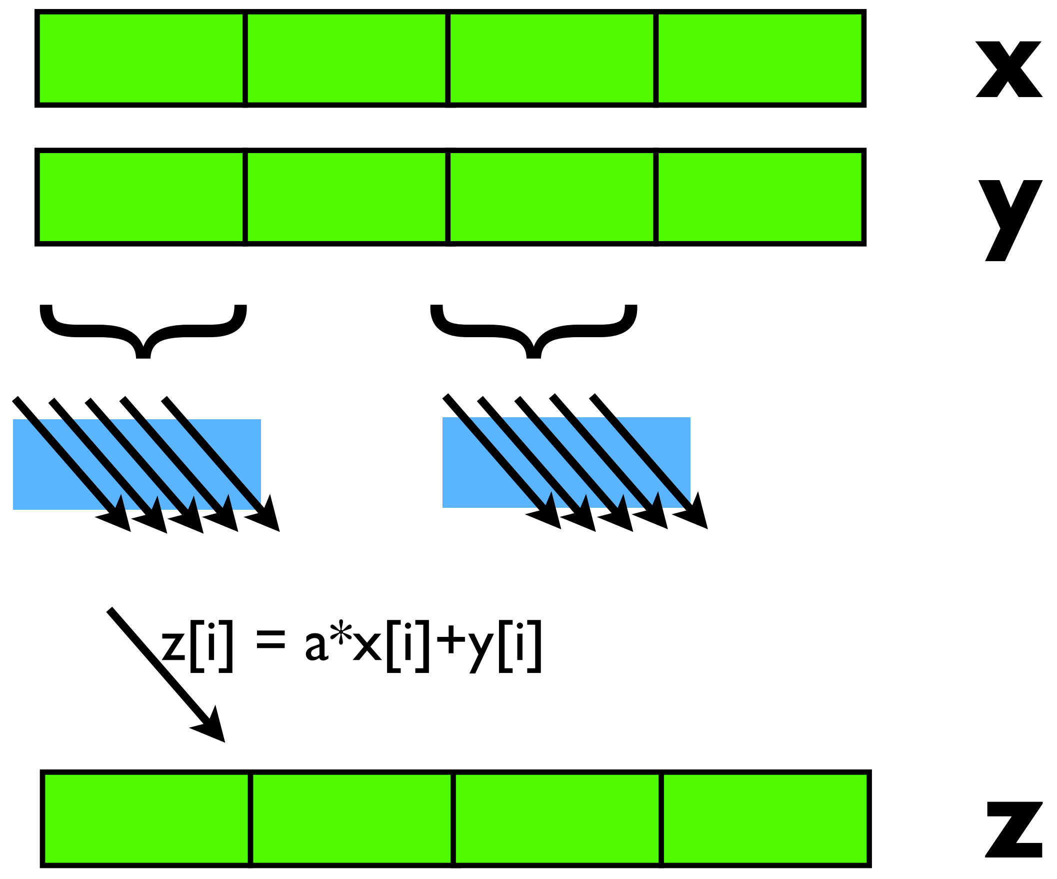
CUDA - H/W mapping

- 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



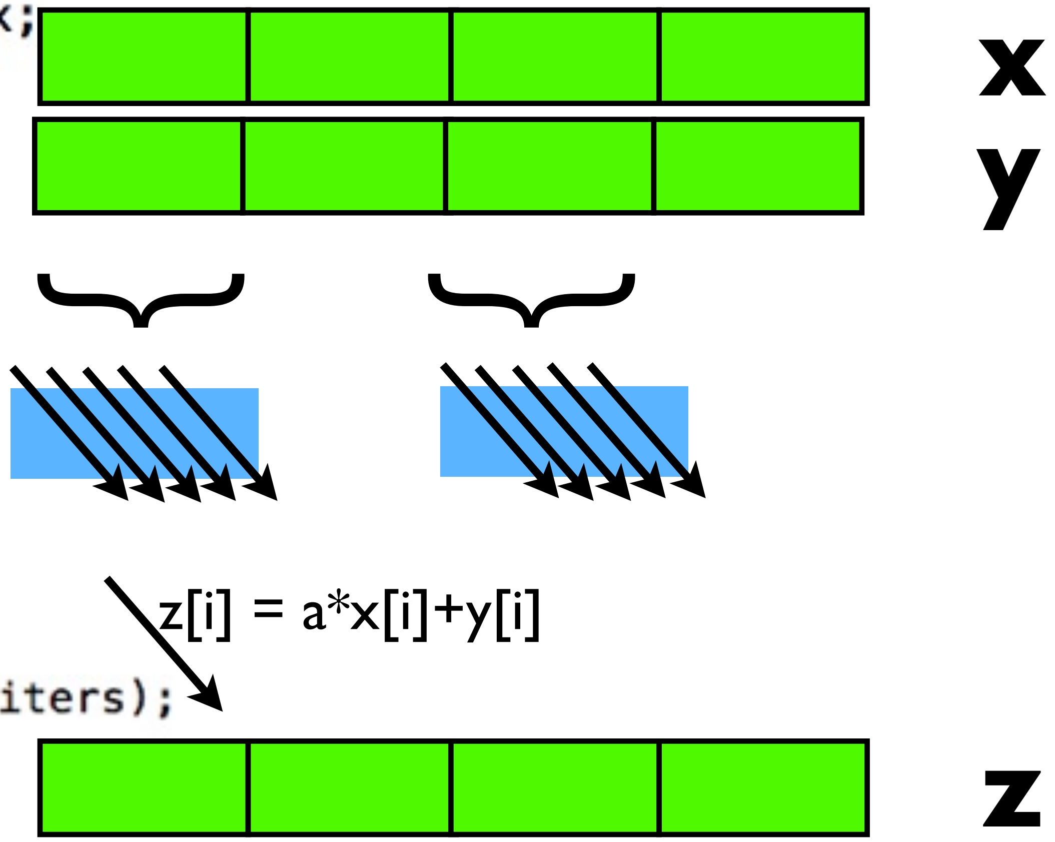
Multi-block z=ax+y

- 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 z[i]



Multi-block z=ax+y

```
__global__ void cuda_saxpy(float *zd, const float a,
                           float *xd, float *yd, const int n) {
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    if (i<n) {
        zd[i] = a*xd[i]+yd[i];
    }
    return;
}
```

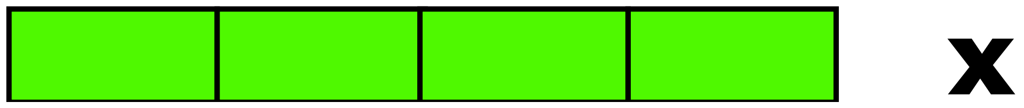


```
err = get_options(argc, argv, &n, &nblocks, &a, &niters);

/* ... */

int blocksize = /*..*/;

for (i=0; i<niters; i++) {
    cuda_saxpy<<<nblocks, blocksize>>>(zd, a, xd, yd, n);
}
```

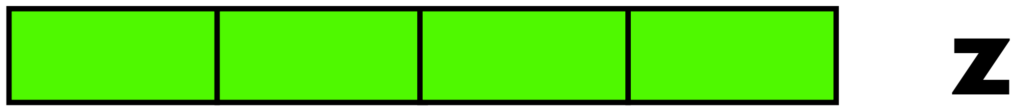
Hands on -- do multi-block saxpy

Enable use of multiple blocks (== multiple SMs!)

Multi-block z=ax+y



```
#define CUDASAXPY
__global__ void cuda_saxpy(float *zd, const float a, float *xd, float *yd, const i

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

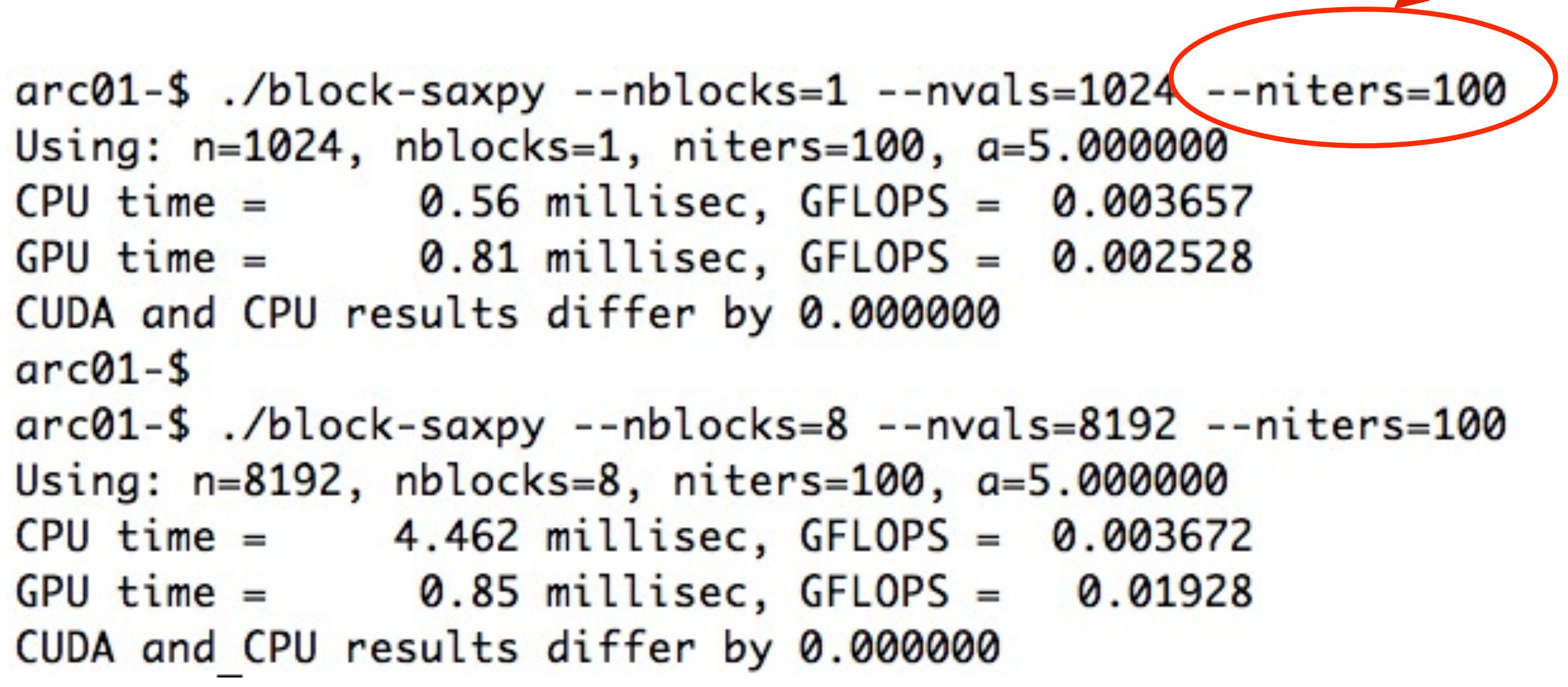


```
blocksize = (n+nblocks-1)/nblocks;
for (i=0; i<niters; i++) {
    cuda_saxpy<<<nblocks, blocksize>>>(zd, a, xd, yd, n);
}
```

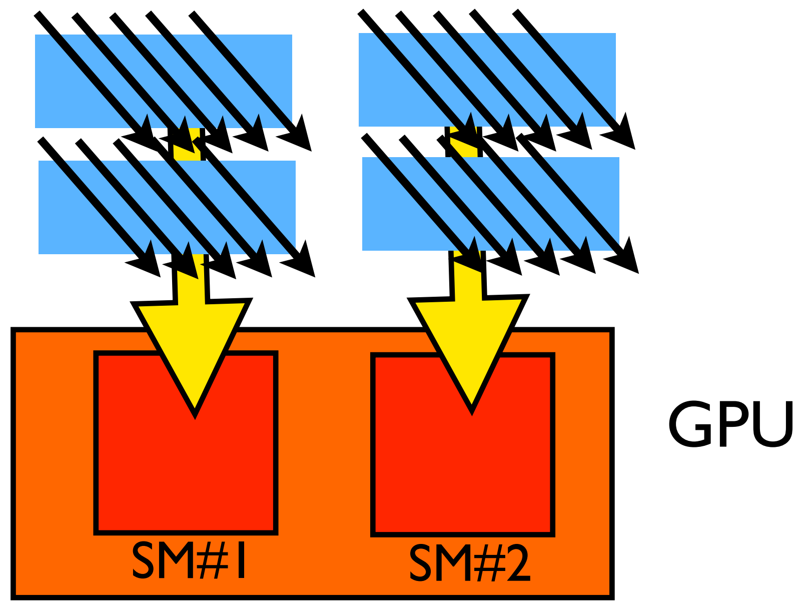
More blocks → more SMs → more FLOPs

- We can use 1024 threads/block:

Multiple calcs, so timing not dominated by memory copy



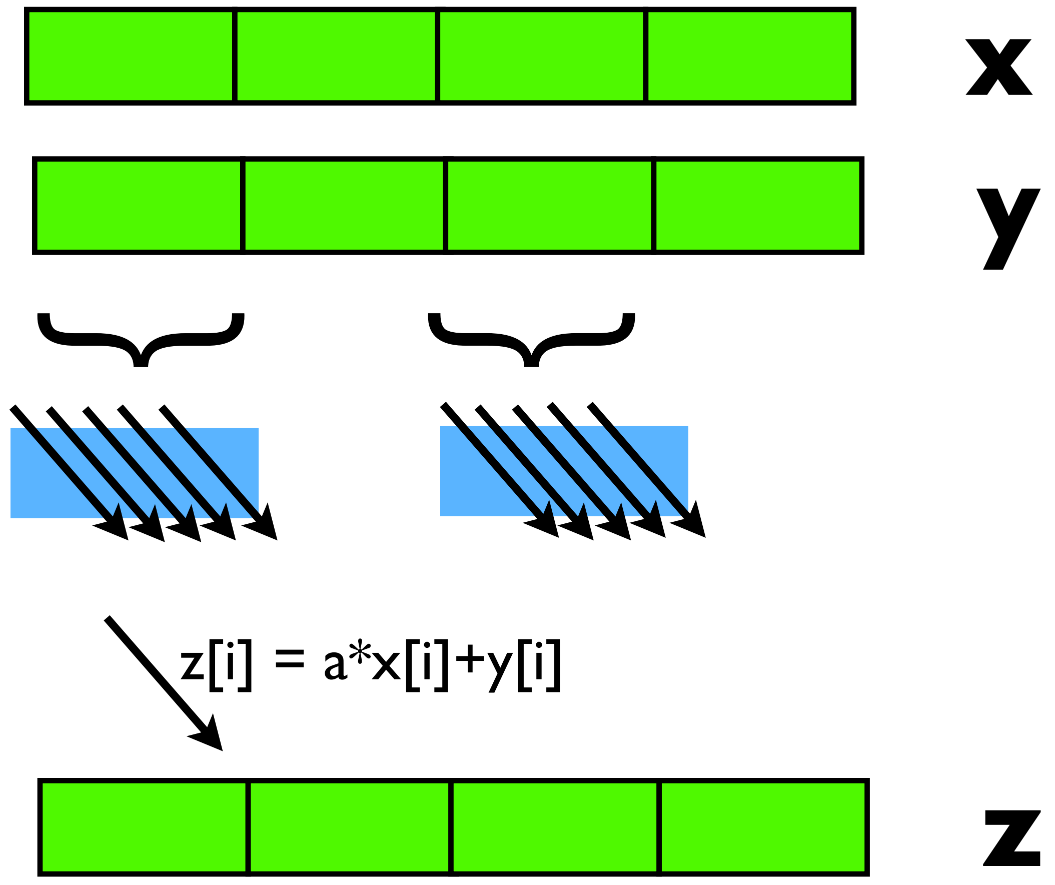
```
arc01-$ ./block-saxpy --nblocks=1 --nvals=1024 --niters=100
Using: n=1024, nblocks=1, niters=100, a=5.000000
CPU time =      0.56 millisec, GFLOPS =  0.003657
GPU time =      0.81 millisec, GFLOPS =  0.002528
CUDA and CPU results differ by 0.000000
arc01-$
arc01-$ ./block-saxpy --nblocks=8 --nvals=8192 --niters=100
Using: n=8192, nblocks=8, niters=100, a=5.000000
CPU time =     4.462 millisec, GFLOPS =  0.003672
GPU time =      0.85 millisec, GFLOPS =   0.01928
CUDA and_CPU results differ by 0.000000
```



Multi-block z=ax+y

```
__global__ void cuda_saxpy(float *zd,
                           const float a,
                           float *xd,
                           float *yd,
                           const int n) {

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

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

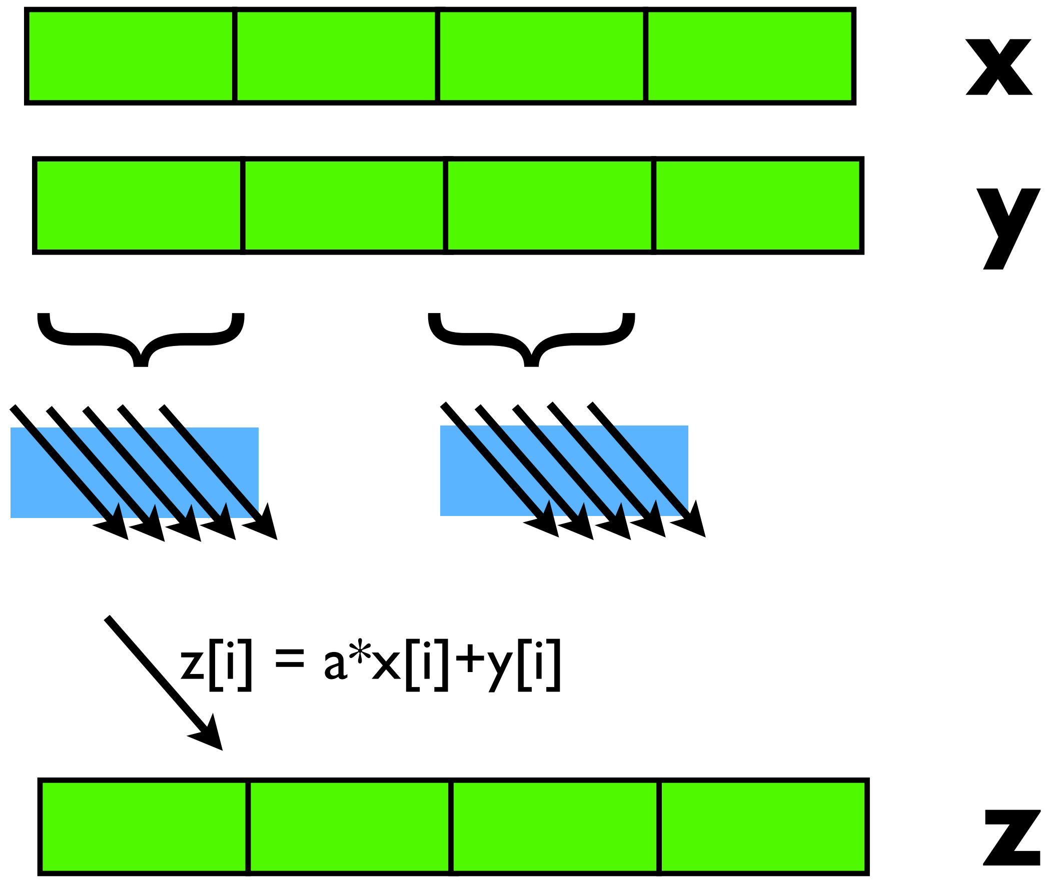


Multi-block z=ax+y

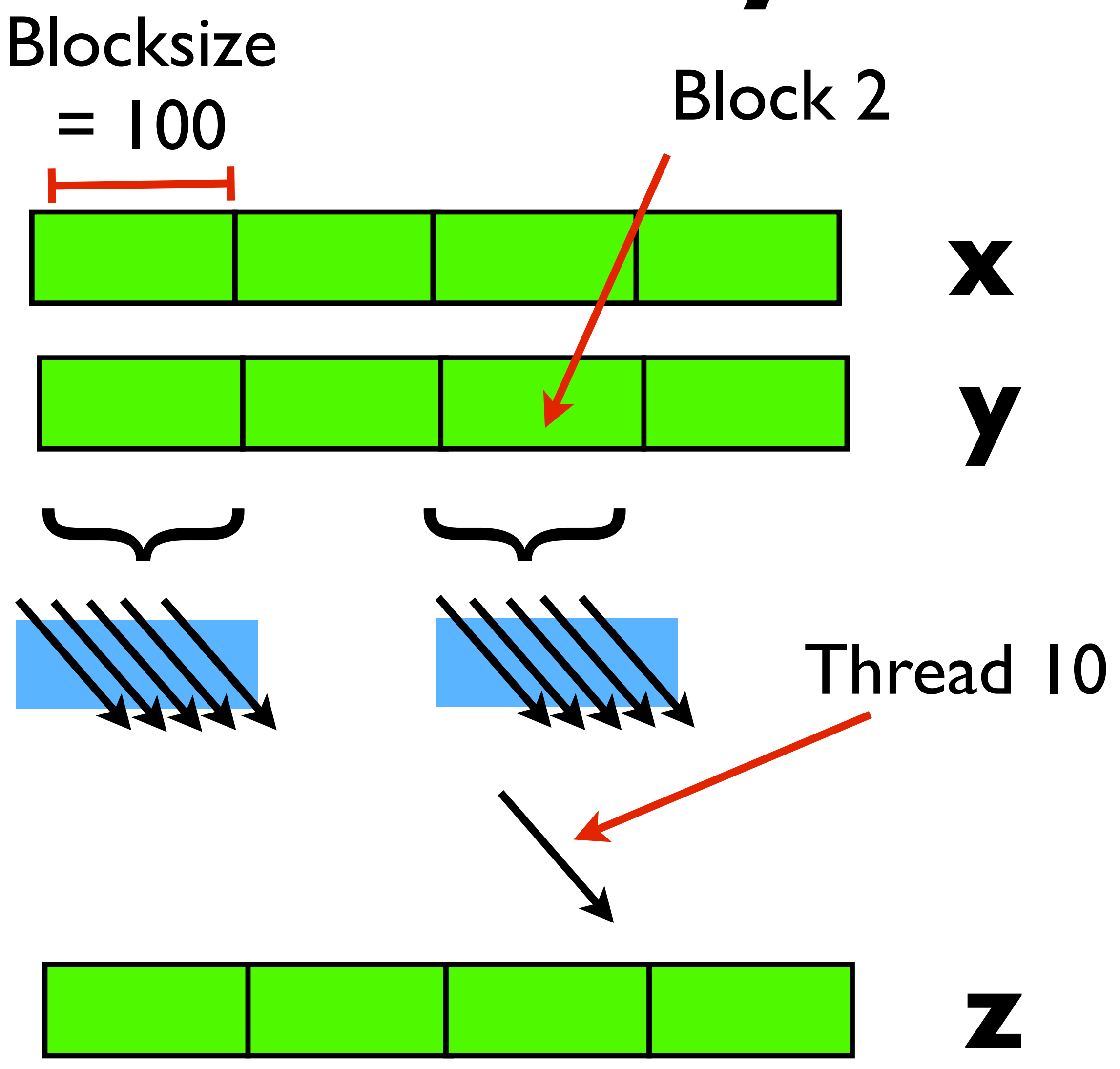
```
__global__ void cuda_saxpy(float *zd,
                           const float a,
                           float *xd,
                           float *yd,
                           const int n) {

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

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

Size of block (blocksize)



Multi-block z=ax+y



```
__global__ void cuda_saxpy(float *zd,
                           const float a,
                           float *xd,
                           float *yd,
                           const int n) {

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

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

SciNet

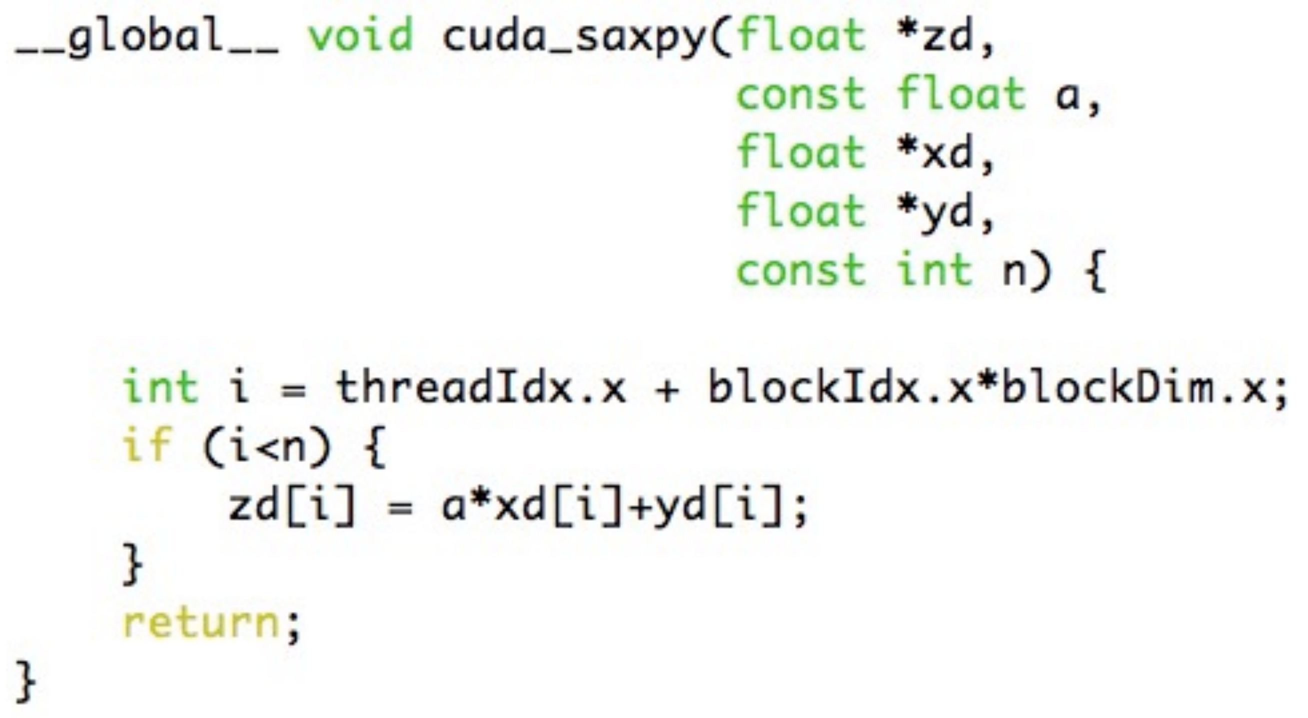
Multi-block z=ax+y

- Now the "if" makes sense:
- Number of work items may not be evenly divided by block size
- Make sure we don't "go off the end"
- What happens in the if statement?
- Thread divergence

```
__global__ void cuda_saxpy(float *zd,
                           const float a,
                           float *xd,
                           float *yd,
                           const int n) {

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

Multi-block z=ax+y

- All threads in a thread block go through kernel in same order.
- Threads in a warp go through in lock step.
- All threads go through if clauses (and else), even if they don't need results
  - (Don't get stored)
- Can be very wasteful!
- Highly "branchy" code not very good for GPUs

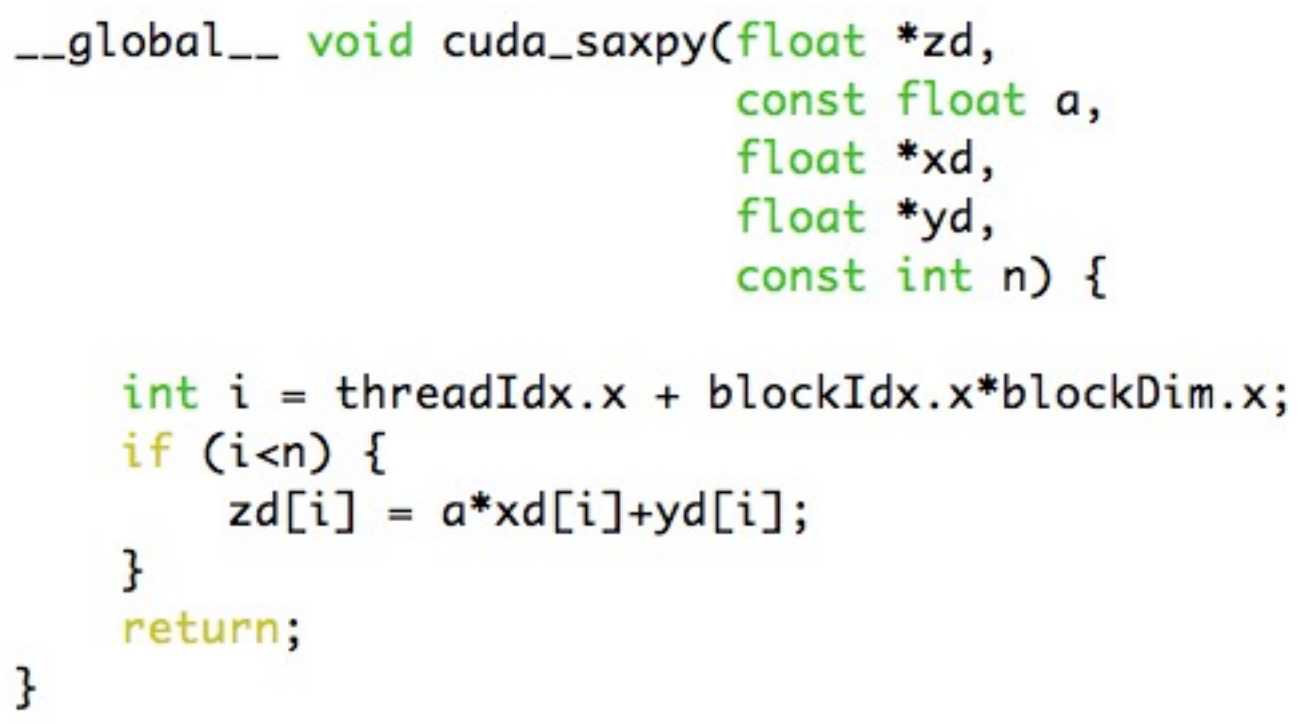


```
__global__ void cuda_saxpy(float *zd,
                           const float a,
                           float *xd,
                           float *yd,
                           const int n) {

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

SciNet

GPGPU Performance Tip #1

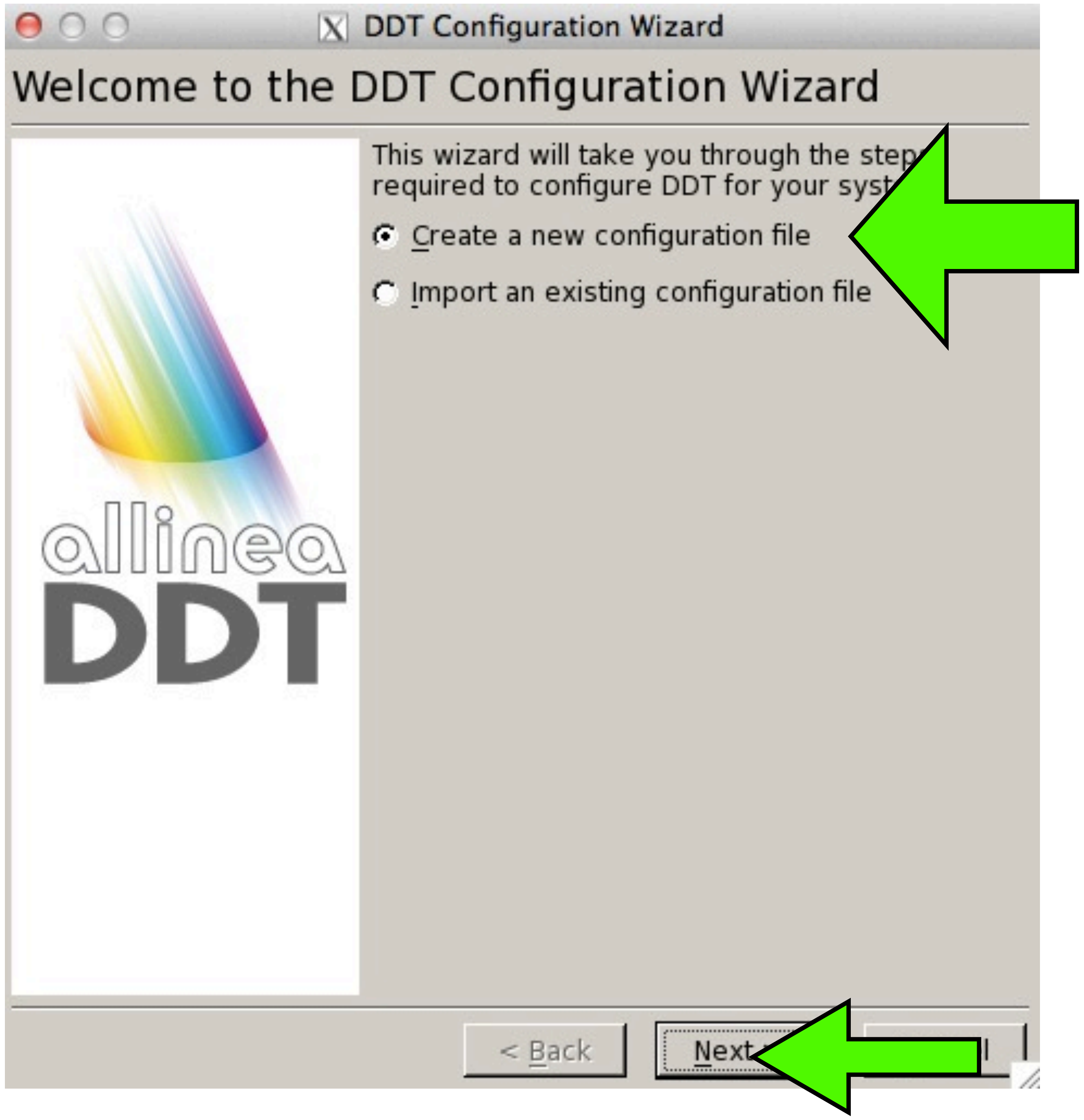
- Avoid lots of branches in GPGPU code.



```
__global__ void cuda_saxpy(float *zd,
                           const float a,
                           float *xd,
                           float *yd,
                           const int n) {

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

DDT

- Let's see what's going on here in more detail with a GPU debugger
- Get a node;
- `qsub -I -X -l nodes=1:ppn=8:gpus=2,walltime=1:00:00`
- cd intro-gpu, source setup
- Type 'ddt' to launch the Allinea DDT debugger:



DDT Configuration Wizard

Welcome to the DDT Configuration Wizard

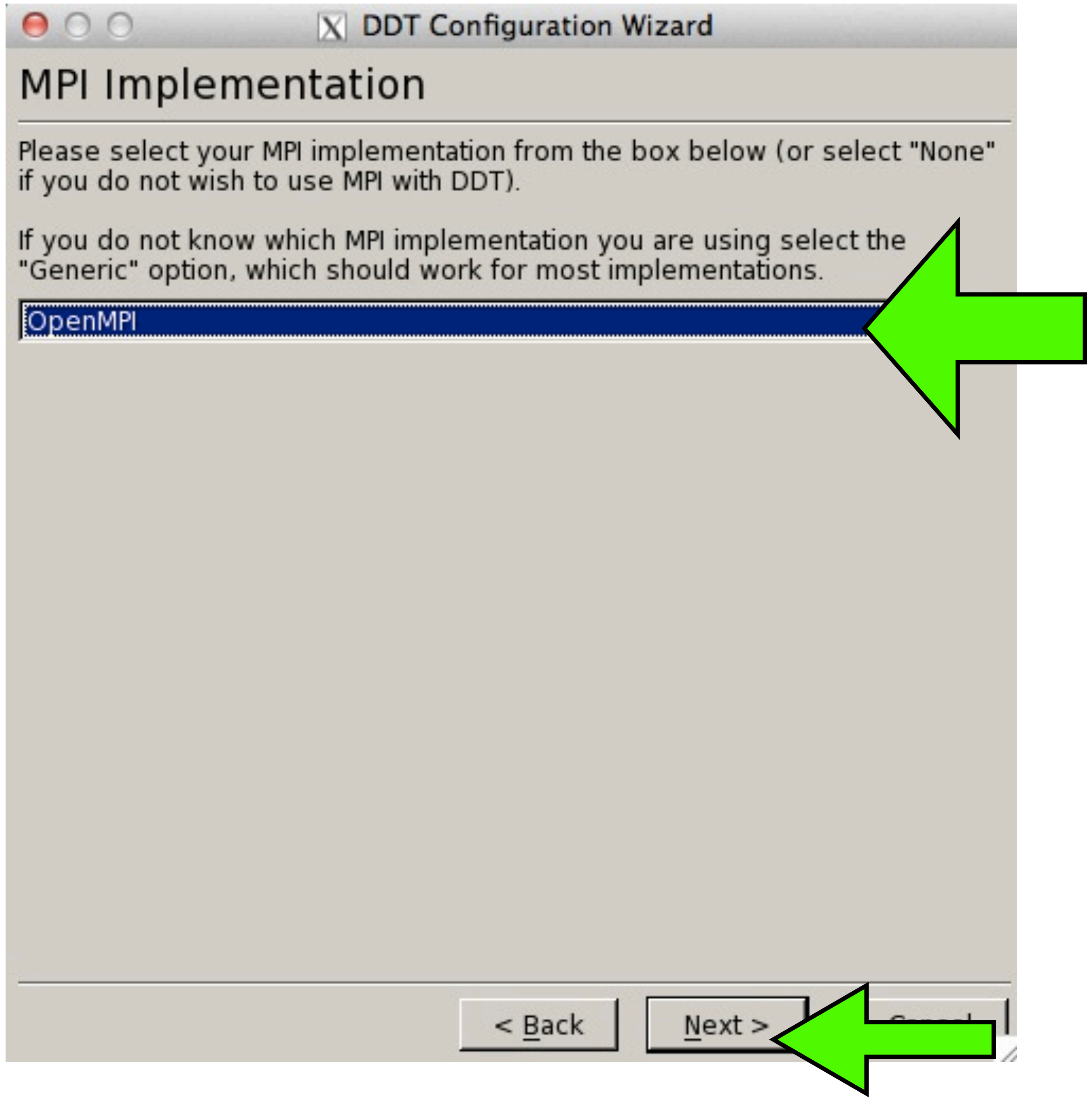
This wizard will take you through the step
required to configure DDT for your syst

(●) Create a new configuration file

( ) Import an existing configuration file

DDT

- Let's see what's going on here in more detail with a GPU debugger
- Type 'ddt' to launch the Allinea DDT debugger:



MPI Implementation

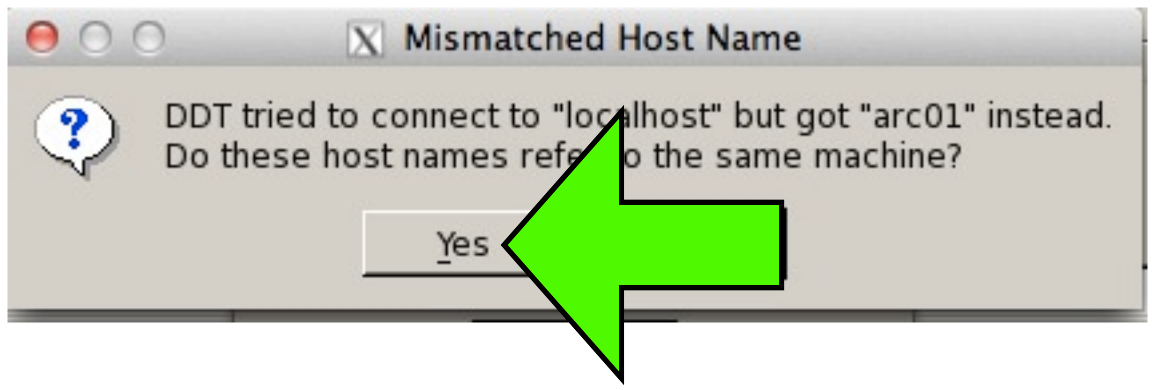
Please select your MPI implementation from the box below (or select "None" if you do not wish to use MPI with DDT).

If you do not know which MPI implementation you are using select the "Generic" option, which should work for most implementations.

OpenMPI

DDT

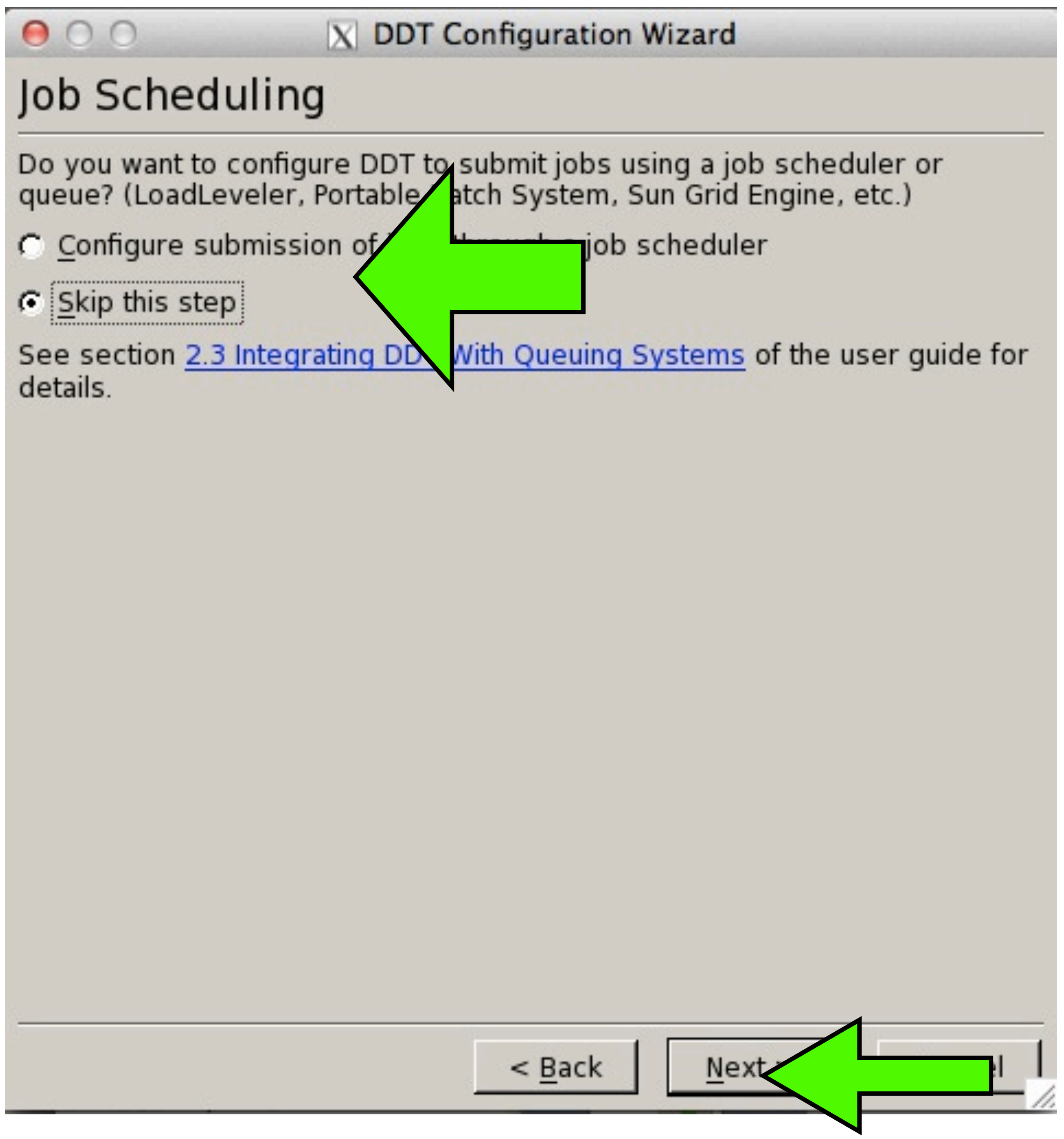
- Let's see what's going on here in more detail with a GPU debugger
- Type 'ddt' to launch the Allinea DDT debugger:



DDT tried to connect to "localhost" but got "arc01" instead. Do these host names refer to the same machine?

DDT

- Let's see what's going on here in more detail with a GPU debugger
- Type 'ddt' to launch the Allinea DDT debugger:



DDT Configuration Wizard

Job Scheduling

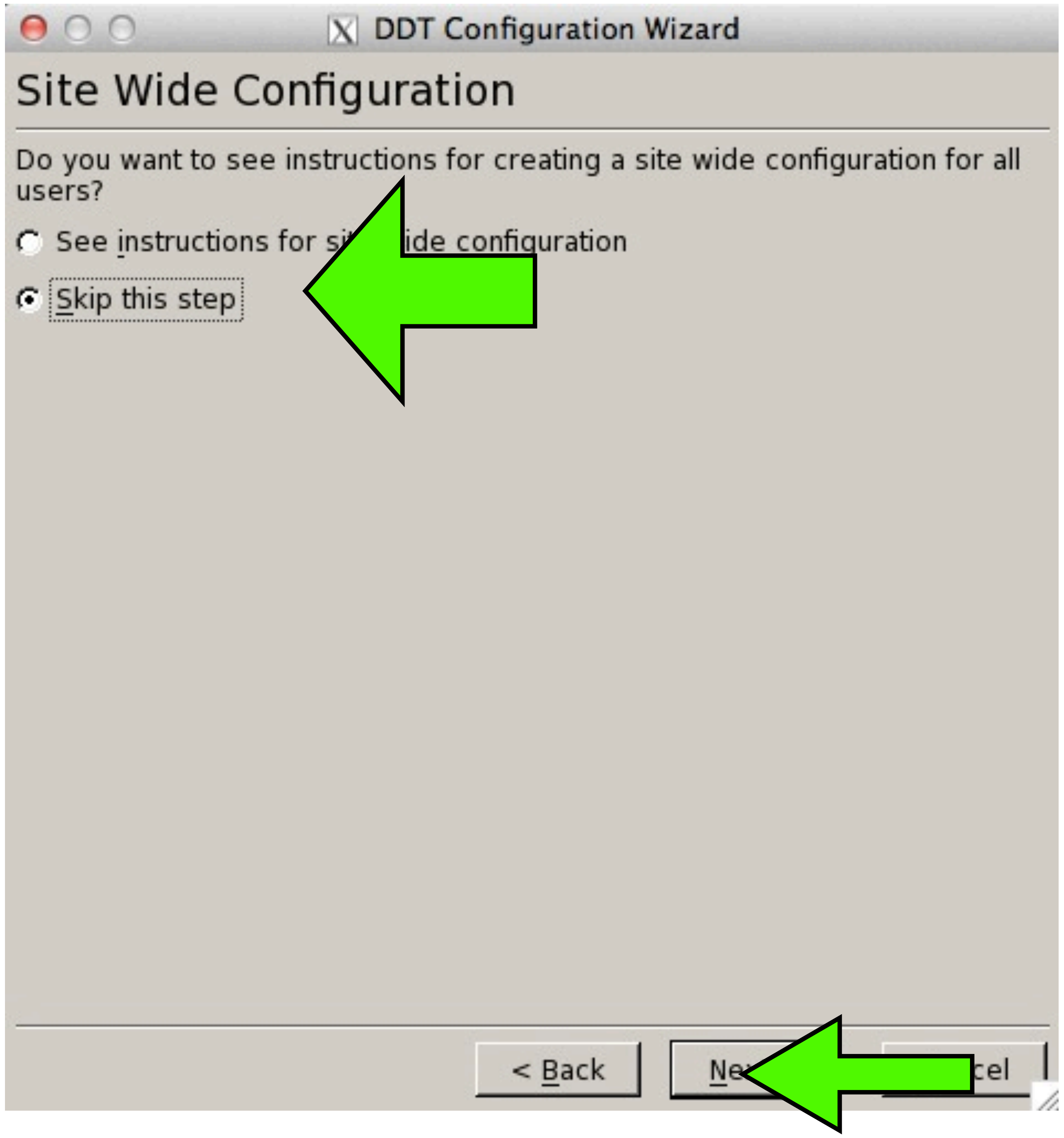
Do you want to configure DDT to submit jobs using a job scheduler or queue? (LoadLeveler, Portable Batch System, Sun Grid Engine, etc.)

- Configure submission of jobs to a job scheduler
- [✓ Skip this step]

See section 2.3 Integrating DDT With Queuing Systems of the user guide for details.

DDT

- Let's see what's going on here in more detail with a GPU debugger
- Type 'ddt' to launch the Allinea DDT debugger:



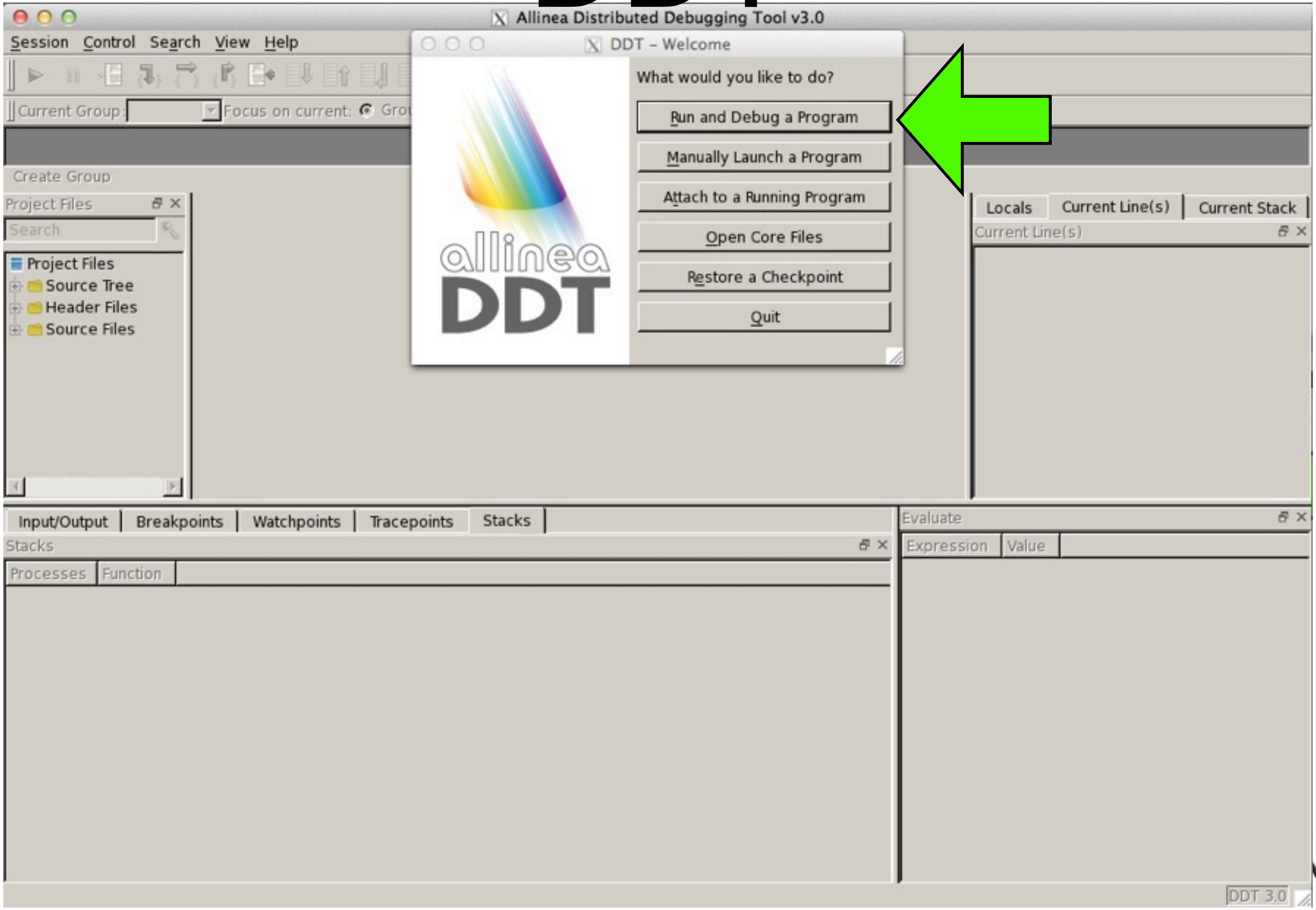
DDT Configuration Wizard

Site Wide Configuration

Do you want to see instructions for creating a site wide configuration for all users?

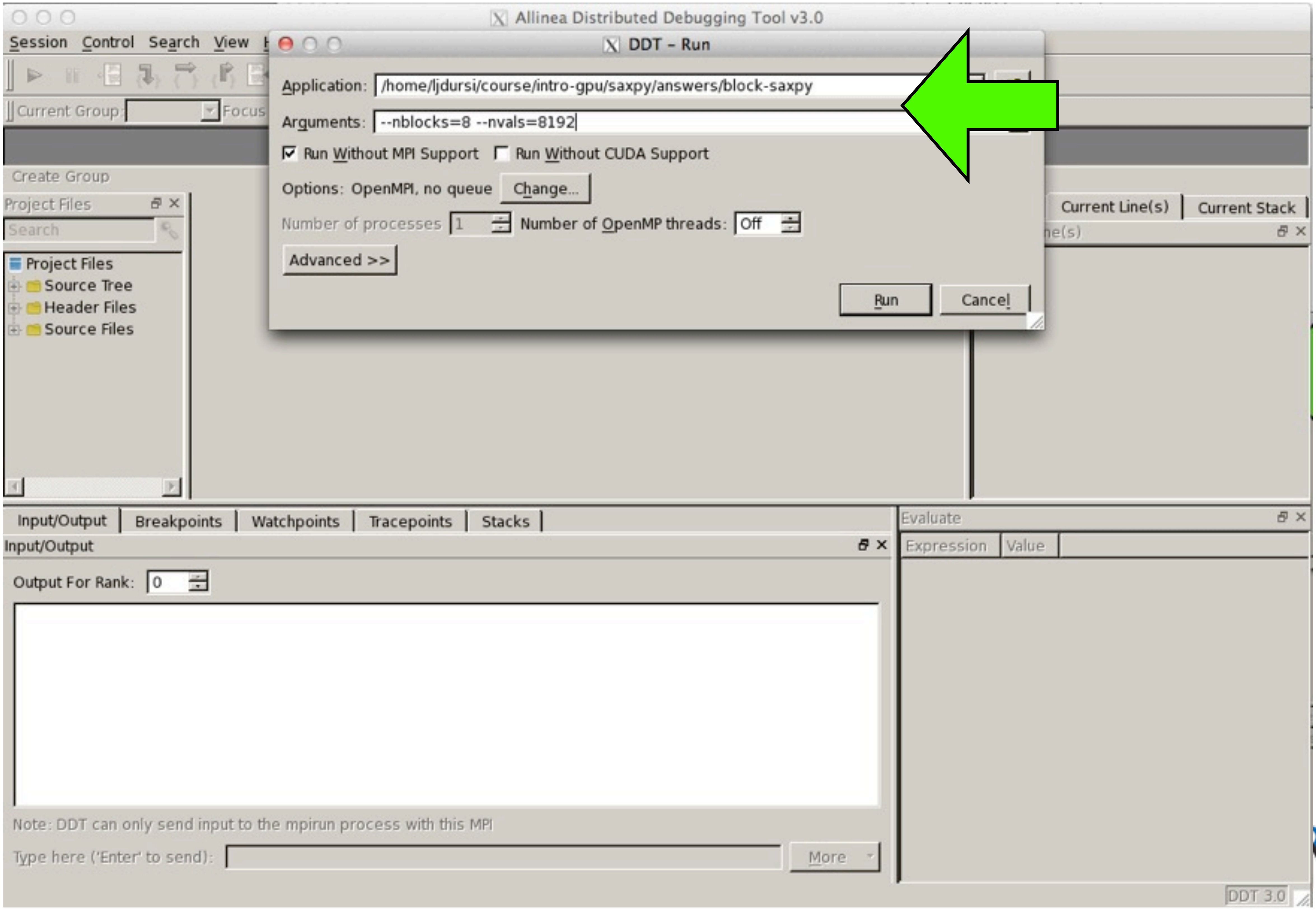
See instructions for si ide configuration [✓ Skip this step]

DDT


Allinea Distributed Debugging Tool v3.0
Session Control Search View Help
Current Group:
Focus on current:
Create Group
Project Files
Search
Project Files
Source Tree
Header Files
Source Files
DDT - Welcome
allinea DDT
What would you like to do?
Run and Debug a Program
Manually Launch a Program
Attach to a Running Program
Open Core Files
Restore a Checkpoint
Quit
Locals
Current Line(s)
Current Stack
Input/Output
Breakpoints
Watchpoints
Tracepoints
Stacks
Processes
Function
Evaluate
Expression
Value
DDT 3.0

Net

DDT



Allinea Distributed Debugging Tool v3.0

DDT - Run

Application: /home/ljdursi/course/intro-gpu/saxpy/answers/block-saxpy

Arguments: --nblocks=8 --nvals=8192

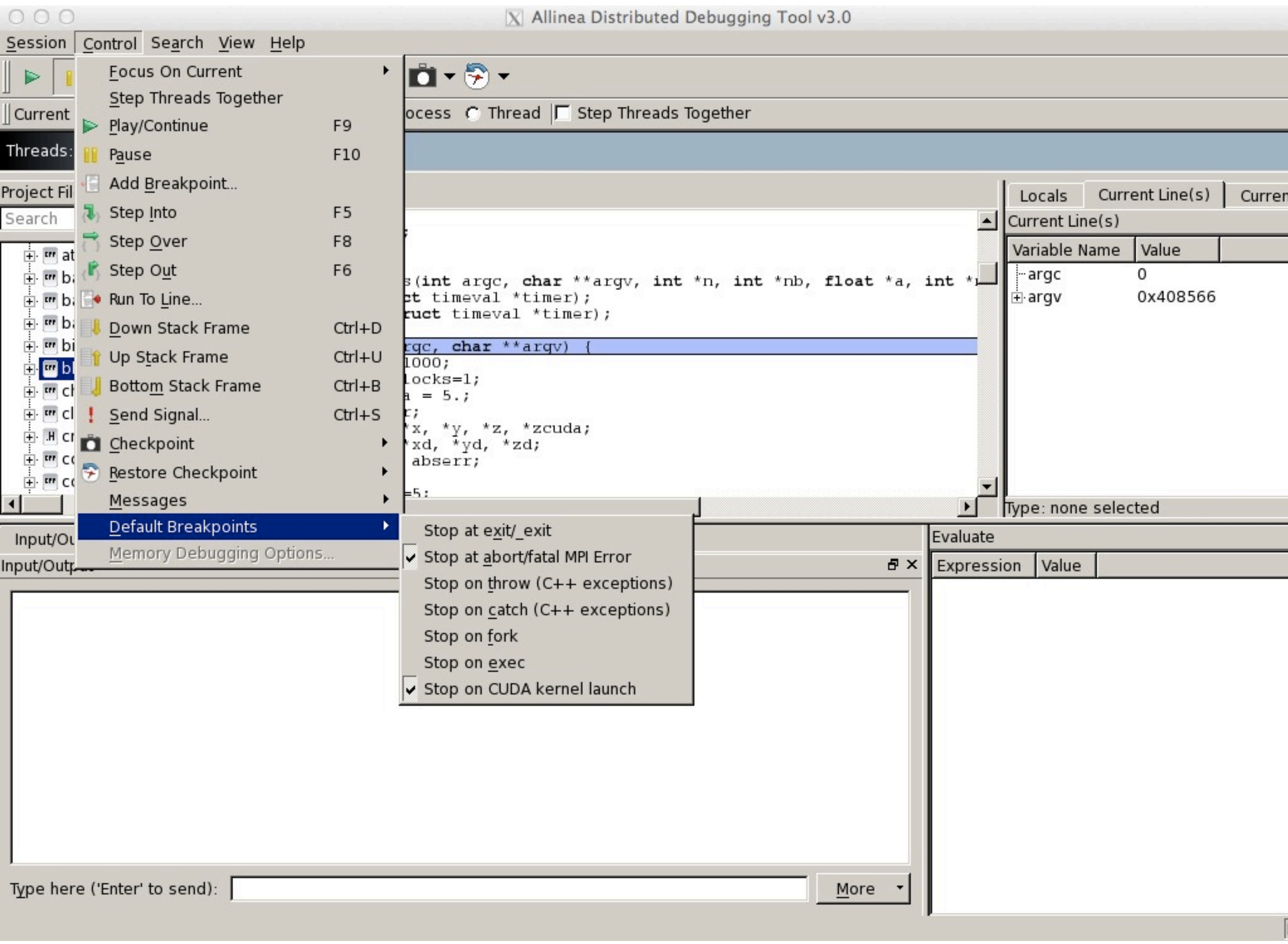
☑ Run Without MPI Support ☐ Run Without CUDA Support

Options: OpenMPI, no queue

Number of processes: 1 Number of OpenMP threads: Off

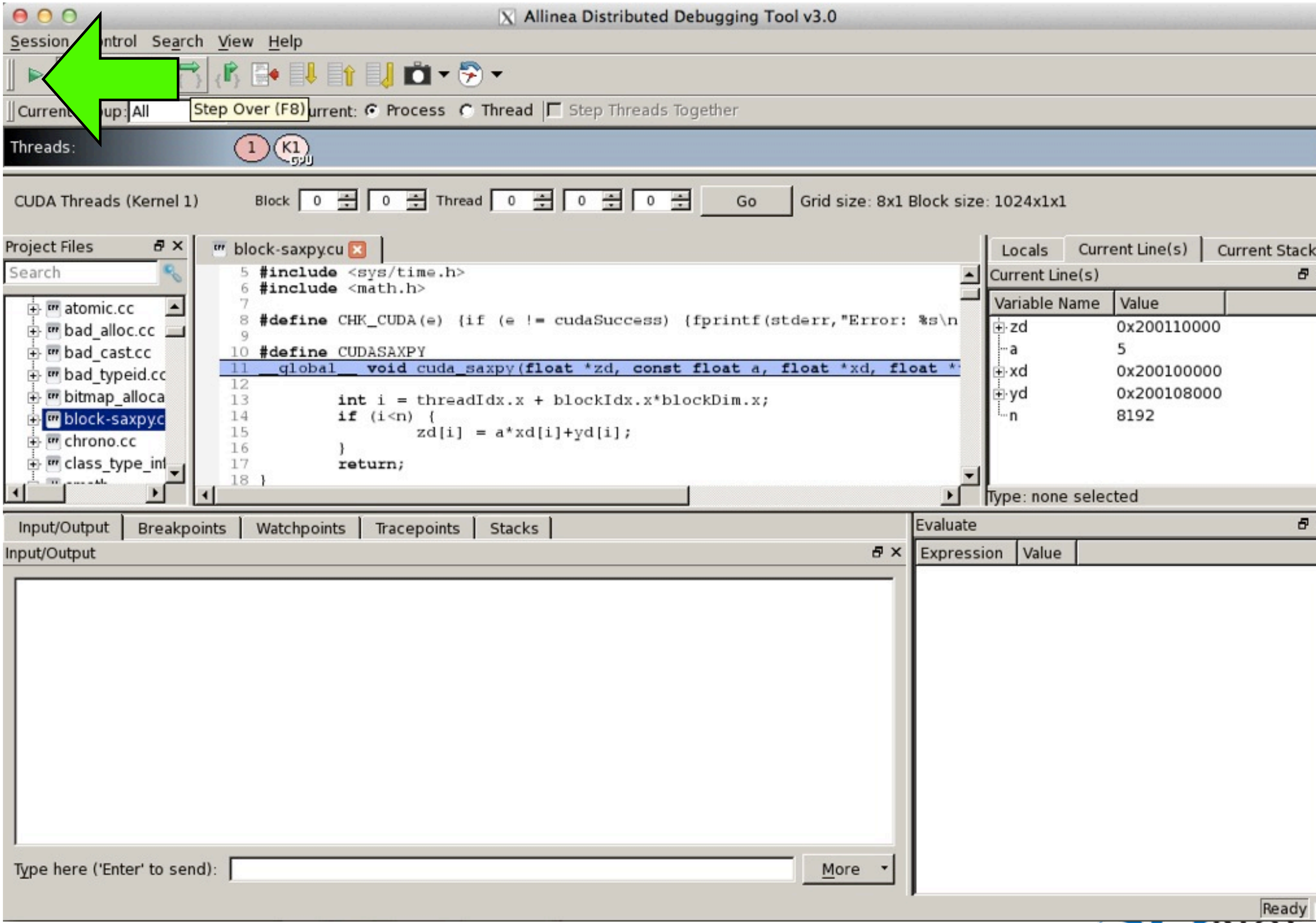
Output For Rank: 0

Note: DDT can only send input to the mpirun process with this MPI



```
s(int argc, char **argv, int *n, int *nb, float *a, int *
ct timeval *timer);
ruct timeval *timer);

rgc, char **argv) {
1000;
locks=1;
a = 5.;
r;
*x, *y, *z, *zcuda;
*xd, *yd, *zd;
 abserr;

=5;
```

Variable Name	Value
argc	0
argv	0x408566



Block 0 0 Thread 0 0 0

Grid size: 8x1 Block size: 1024x1x1

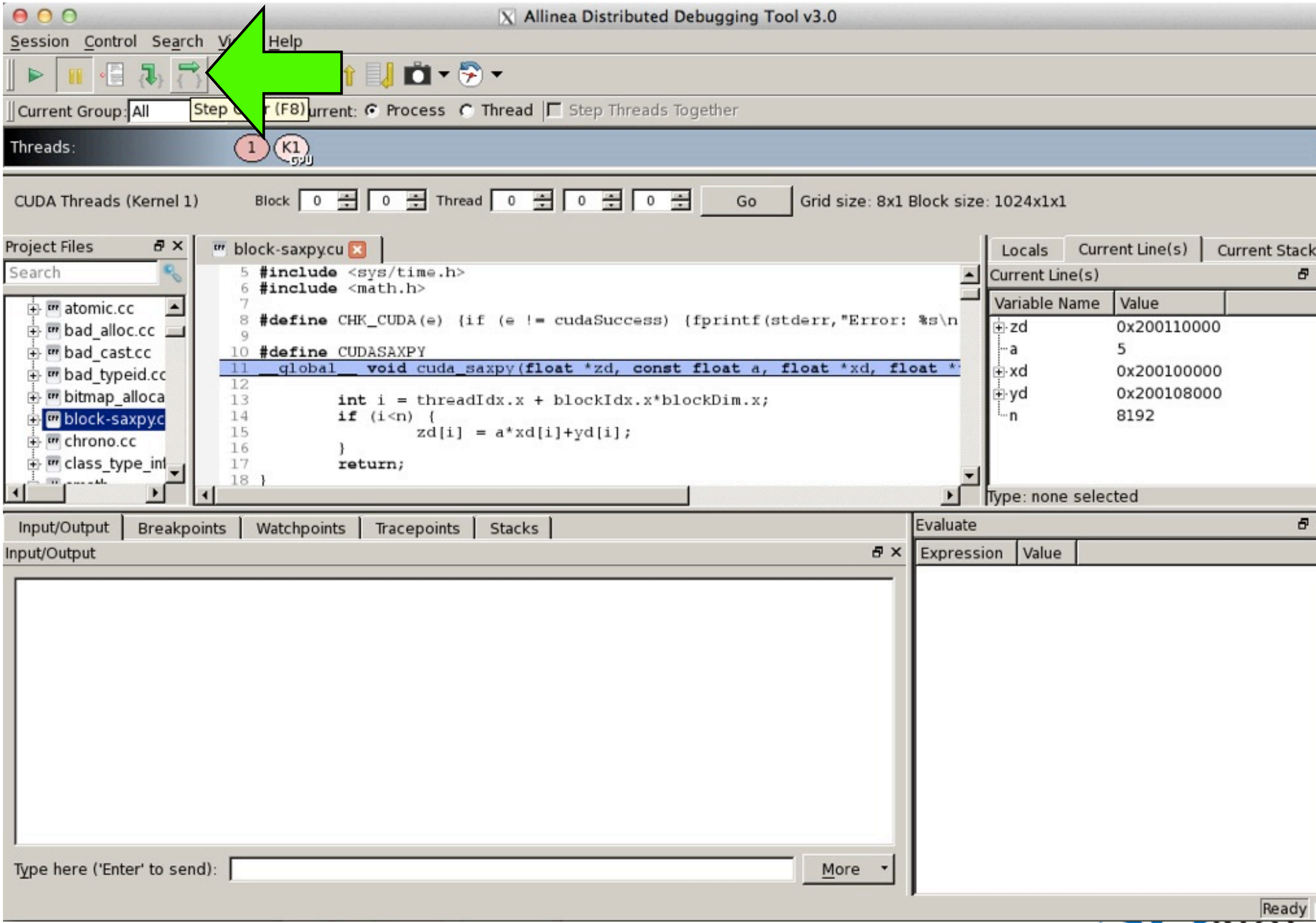
```
#include <sys/time.h>
#include <math.h>

#define CHK_CUDA(e) {if (e != cudaSuccess) {fprintf(stderr,"Error: %s\n

#define CUDASAXPY
__global__ void cuda_saxpy(float *zd, const float a, float *xd, float *

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

Current Line(s)

Variable Name	Value
zd	0x200110000
a	5
xd	0x200100000
yd	0x200108000
n	8192

Type: none selected

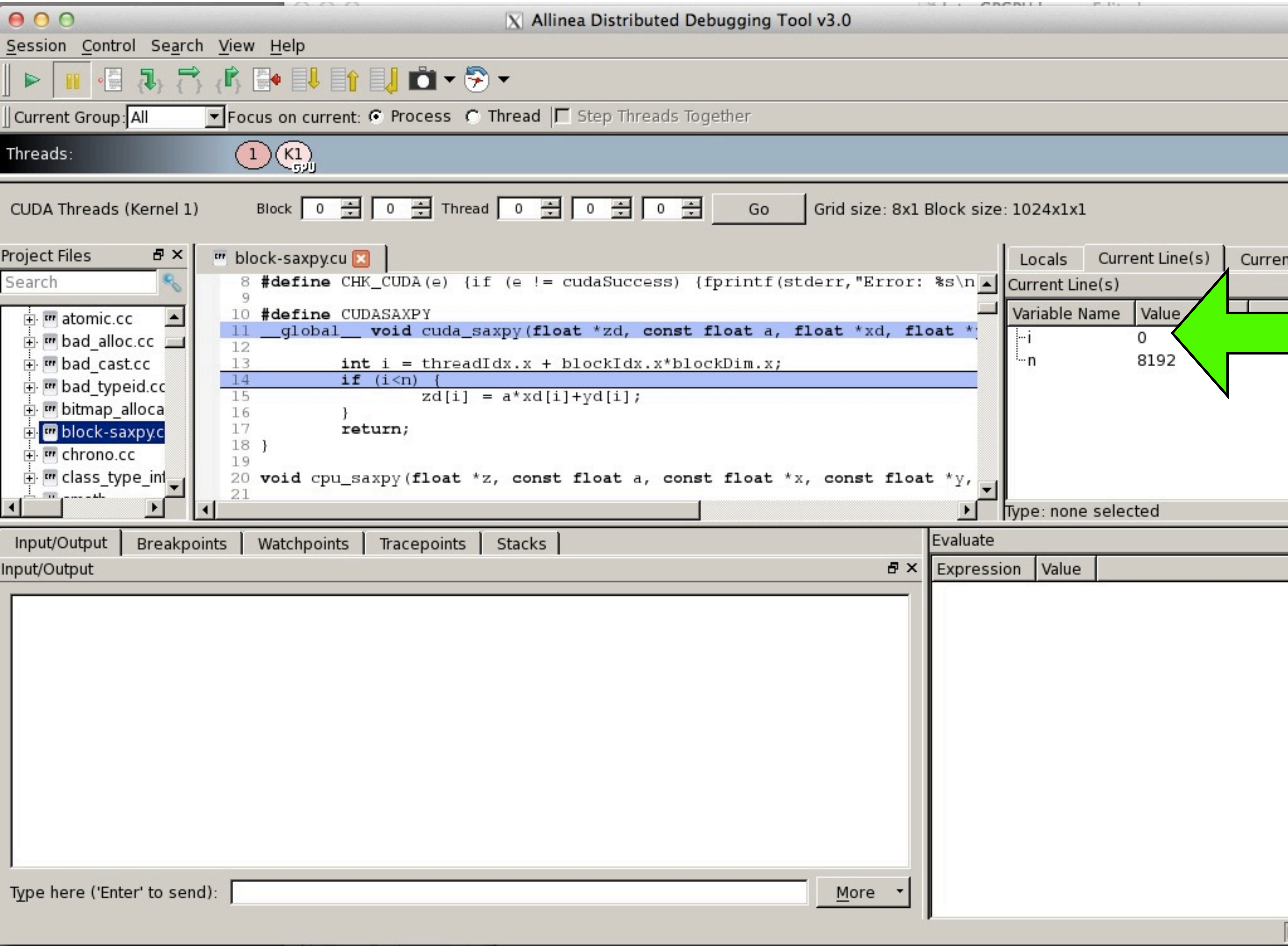
Input/Output



Allinea Distributed Debugging Tool v3.0
Session Control Search View Help
Step Over (F8)
Current Group: All
Current: Process Thread Step Threads Together
Threads: 1 K1 GPU
CUDA Threads (Kernel 1) Block 0 0 Thread 0 0 0 Go
Grid size: 8x1 Block size: 1024x1x1
Project Files
atomic.cc
bad_alloc.cc
bad_cast.cc
bad_typeid.cc
bitmap_alloca
block-saxpy.c
chrono.cc
class_type_inf
block-saxpy.cu
Locals Current Line(s) Current Stack
Type: none selected
Input/Output Breakpoints Watchpoints Tracepoints Stacks
Evaluate
Expression Value
Type here ('Enter' to send):
More
Ready

```
#include <sys/time.h>
#include <math.h>

#define CHK_CUDA(e) {if (e != cudaSuccess) {fprintf(stderr,"Error: %s\n

#define CUDASAXPY
__global__ void cuda_saxpy(float *zd, const float a, float *xd, float *

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

Variable Name	Value
zd	0x200110000
a	5
xd	0x200100000
yd	0x200108000
n	8192

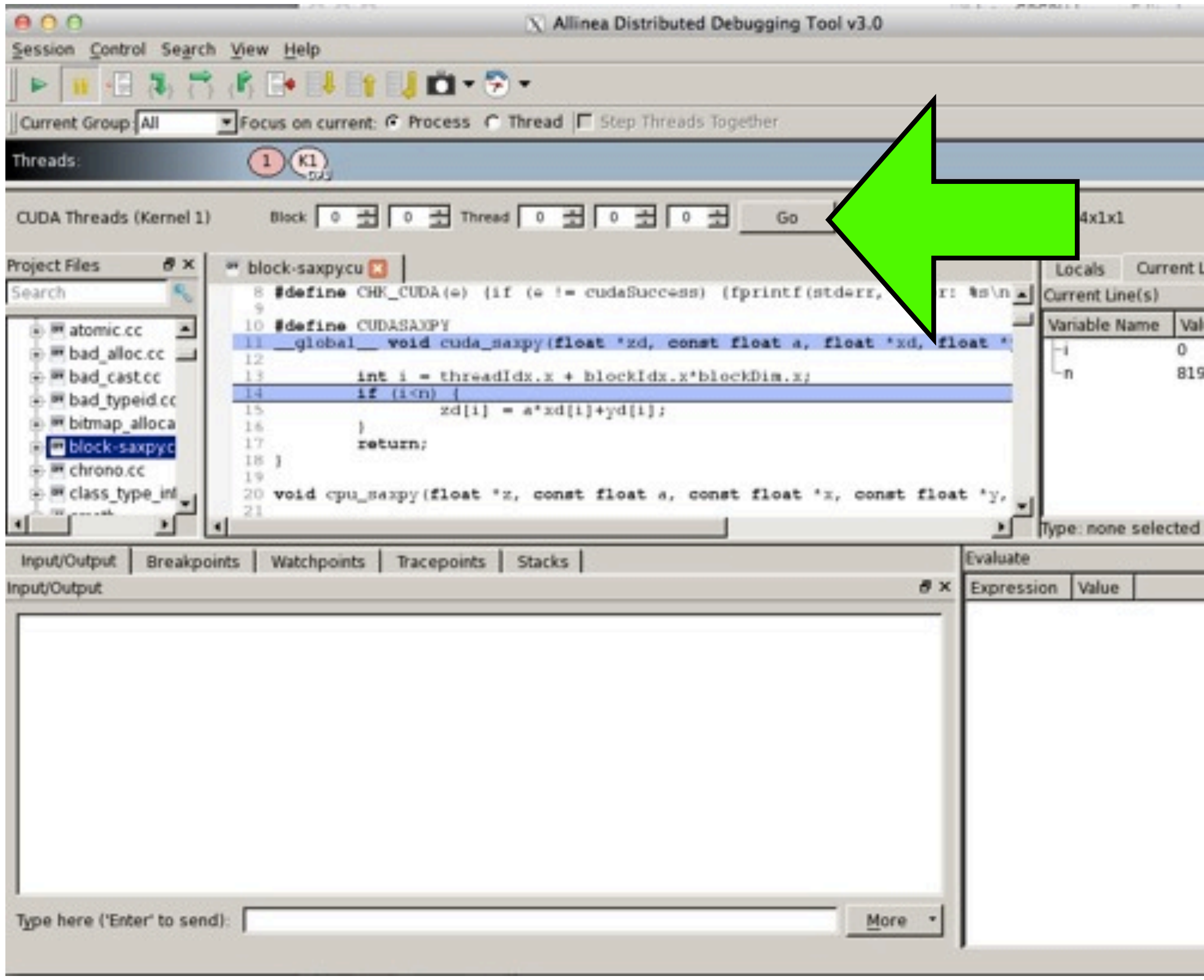


```
#define CHK_CUDA(e) {if (e != cudaSuccess) {fprintf(stderr,"Error: %s\n

#define CUDASAXPY
__global__ void cuda_saxpy(float *zd, const float a, float *xd, float *

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

void cpu_saxpy(float *z, const float a, const float *x, const float *y,
```

Variable Name	Value
i	0
n	8192

DDT

- Can play with first numbers of "block" and "thread" to see different block, thread
- value shown of i should change
- Does i give what you'd expect?



```
#define CHK_CUDA(e) {if (e != cudaSuccess) {fprintf(stderr,       r: %s\n

#define CUDASAXPY
__global__ void cuda_saxpy(float *xd, const float a, float *xd, float *

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

void cpu_saxpy(float *x, const float a, const float *x, const float *y,
```

cuda-gdb

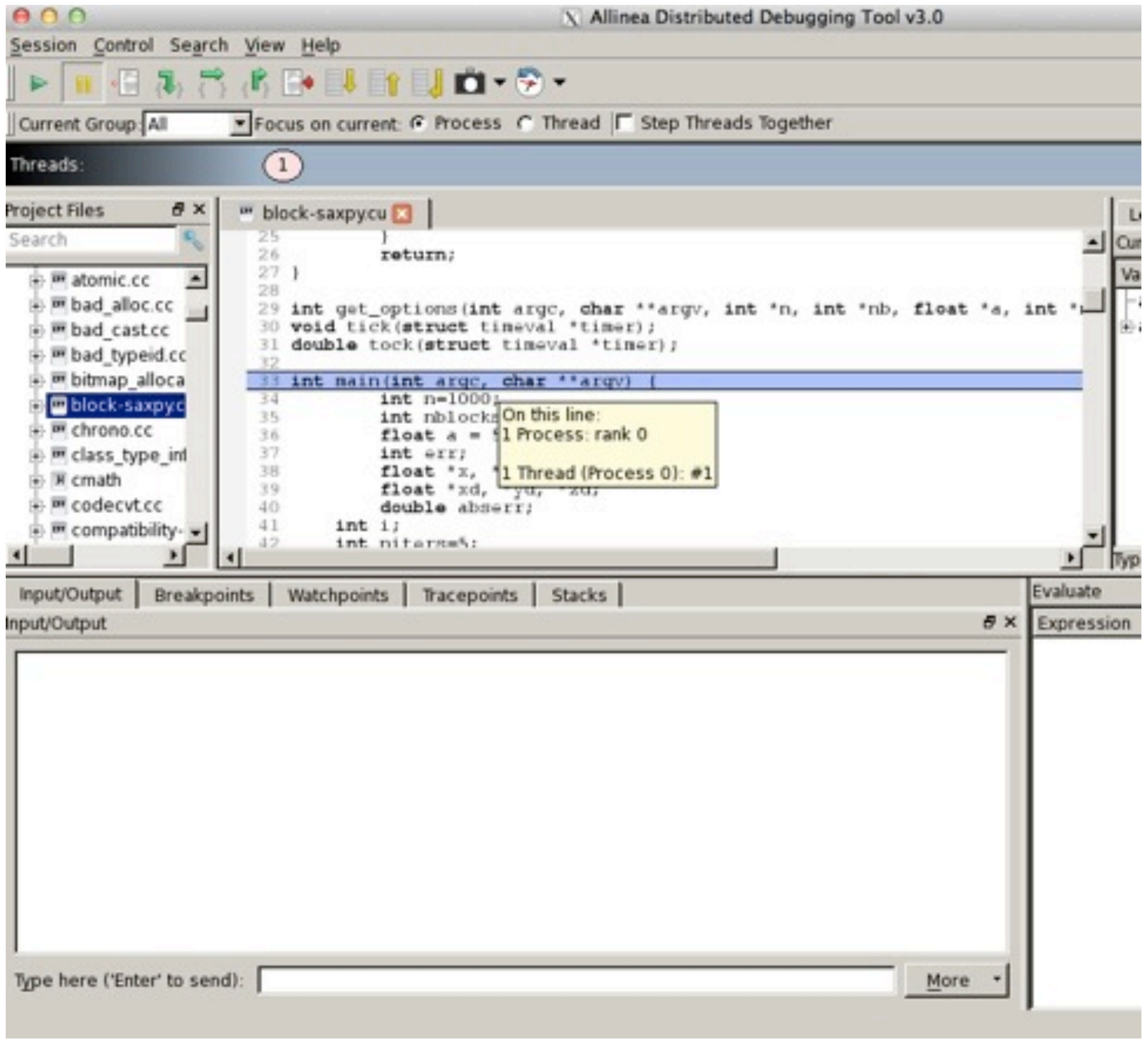
```
arc01-$ cuda-gdb ./block-saxpy
NVIDIA (R) CUDA Debugger
3.2 release
[...]
(cuda-gdb) break cuda_saxpy
(cuda-gdb) run --nvals=8192 --nblocks=8
Starting program: [...]
[Launch of CUDA Kernel 0 (cuda_saxpy) on Device 0]
[Switching to CUDA Kernel 0 (<<<(0,0),(0,0,0)>>>)]

Breakpoint 1, cuda_saxpy<<<(1,1),(1000,1,1)>>> (zd=0x200102000,
a=5,
    xd=0x200100000, yd=0x200101000, n=1000) at block-saxpy.cu:
13
13        int i = threadIdx.x + blockIdx.x*blockDim.x;
(cuda-gdb)
```

cuda-gdb

```
(cuda-gdb) step
14      if (i<n) {
(cuda-gdb) print i
$1 = 0
(cuda-gdb) cuda thread 8
[Switching to CUDA Kernel 0 (device 0, sm 0, warp 0, lane 8,
grid 1, block (0,0), thread (8,0,0))]
14      if (i<n) {
(cuda-gdb) print i
$2 = 8
(cuda-gdb) cuda block 2
[Switching to CUDA Kernel 0 (device 0, sm 3, warp 0, lane 8,
grid 1, block (2,0), thread (8,0,0))]
13      int i = threadIdx.x + blockIdx.x*blockDim.x;
(cuda-gdb) step
14      if (i<n) {
(cuda-gdb) print i
$4 = 2056
(cuda-gdb) quit
```

nvcc -G -g

- Note; the -g option to the compiler (nvcc) kept debugging symbols in the host code; the -G option kept the symbols in the kernel code
- Allows use of debugger, better diagnostics.
- But disables many optimizations...



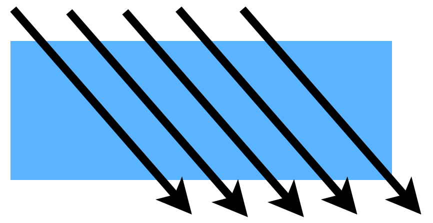
```
        }
        return;
}

int get_options(int argc, char **argv, int *n, int *nb, float *a, int *
void tick(struct timeval *timer);
double tock(struct timeval *timer);

int main(int argc, char **argv) {
        int n=1000;
        int nblocks
        float a =
        int err;
        float *x,
        float *xd,   yd,  zd;
        double abserr;
    int i;
    int niters=5;
```

On this line:
1 Process: rank 0

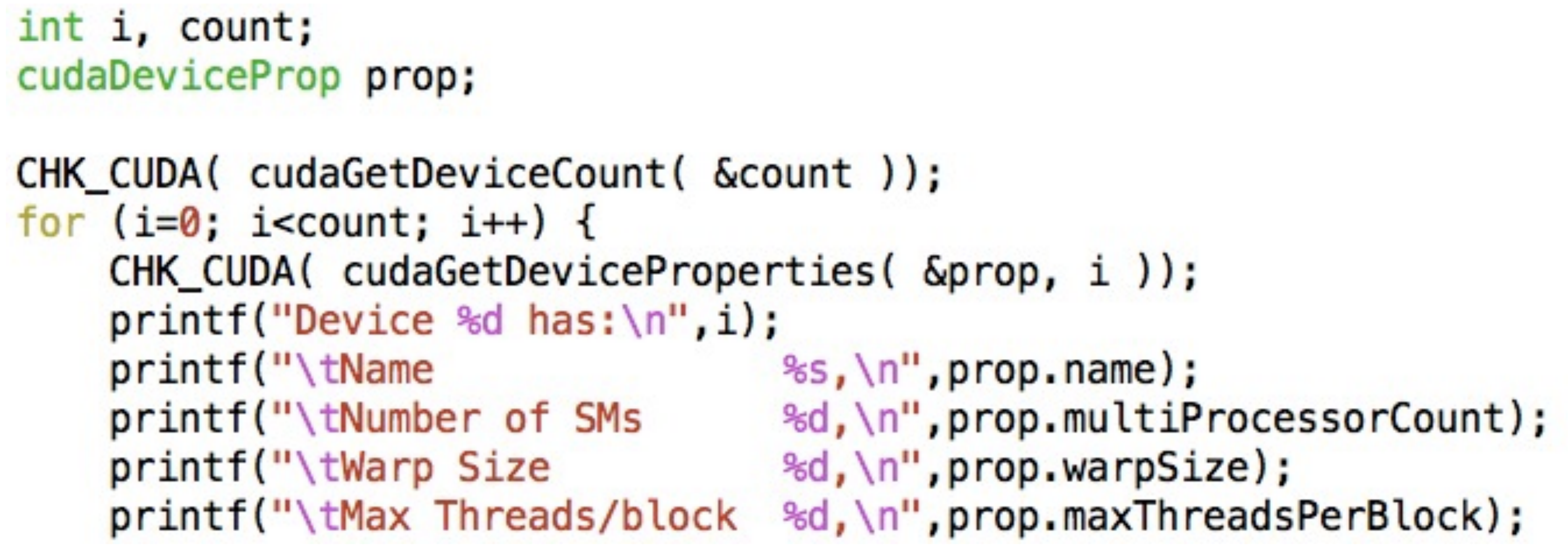
1 Thread (Process 0): #1

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:



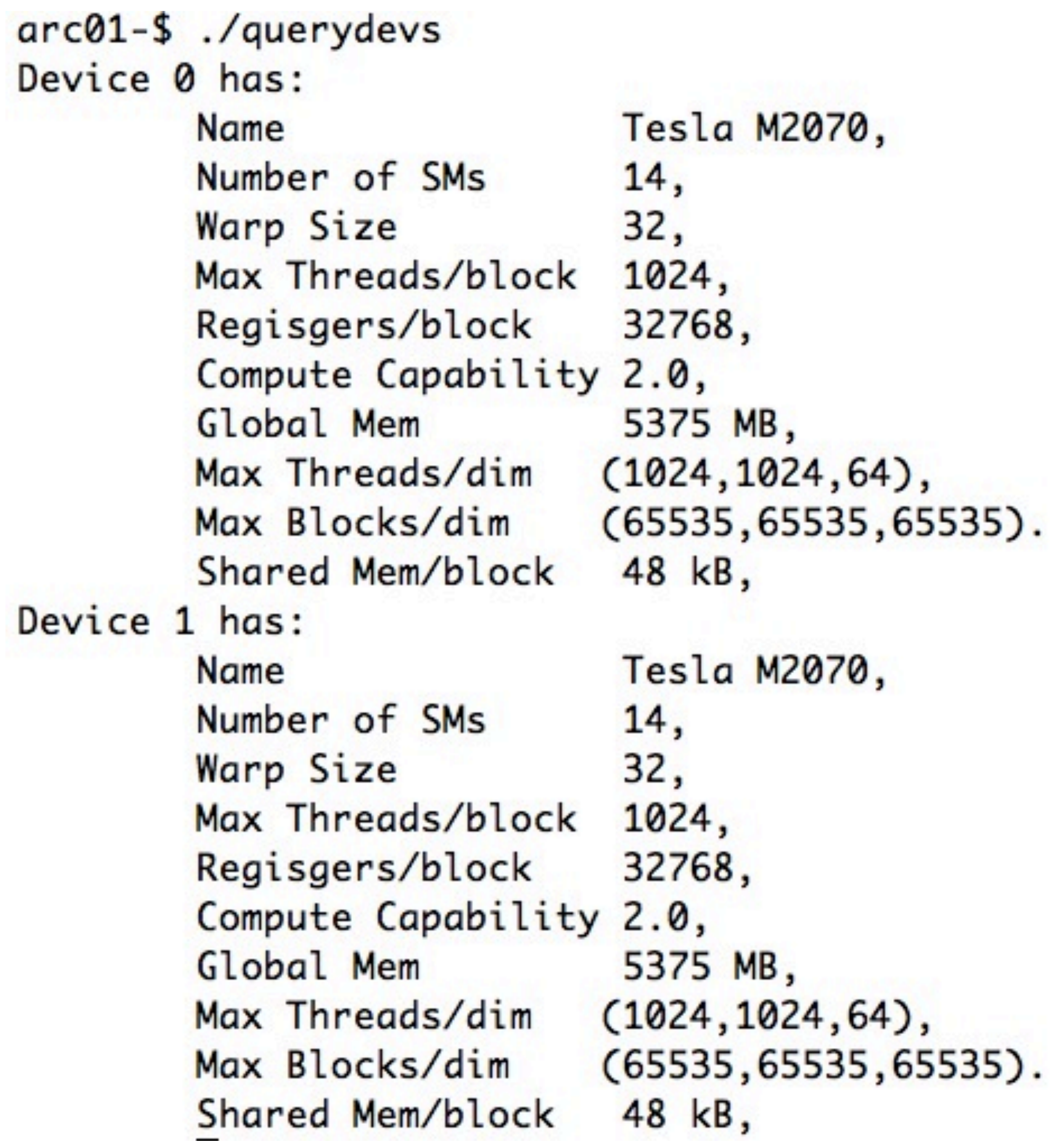
cudaGetDeviceProperty



```
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);
```

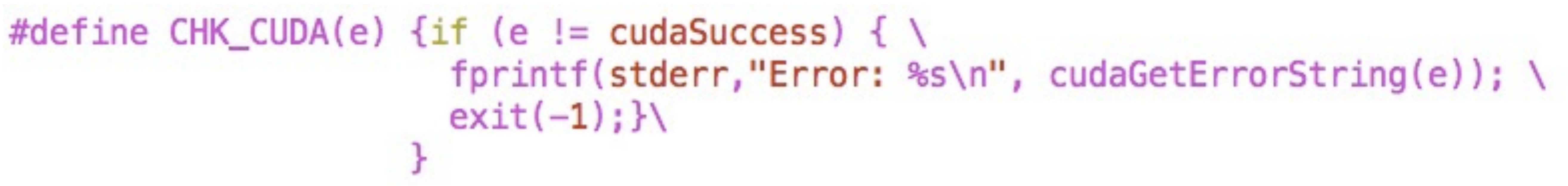
querydevs.cu

cudaGetDeviceProperty



```
arc01-$ ./querydevs
Device 0 has:
        Name                Tesla M2070,
        Number of SMs       14,
        Warp Size           32,
        Max Threads/block   1024,
        Regisgers/block     32768,
        Compute Capability  2.0,
        Global Mem          5375 MB,
        Max Threads/dim    (1024,1024,64),
        Max Blocks/dim     (65535,65535,65535).
        Shared Mem/block    48 kB,
Device 1 has:
        Name                Tesla M2070,
        Number of SMs       14,
        Warp Size           32,
        Max Threads/block   1024,
        Regisgers/block     32768,
        Compute Capability  2.0,
        Global Mem          5375 MB,
        Max Threads/dim    (1024,1024,64),
        Max Blocks/dim     (65535,65535,65535).
        Shared Mem/block    48 kB,
```

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.

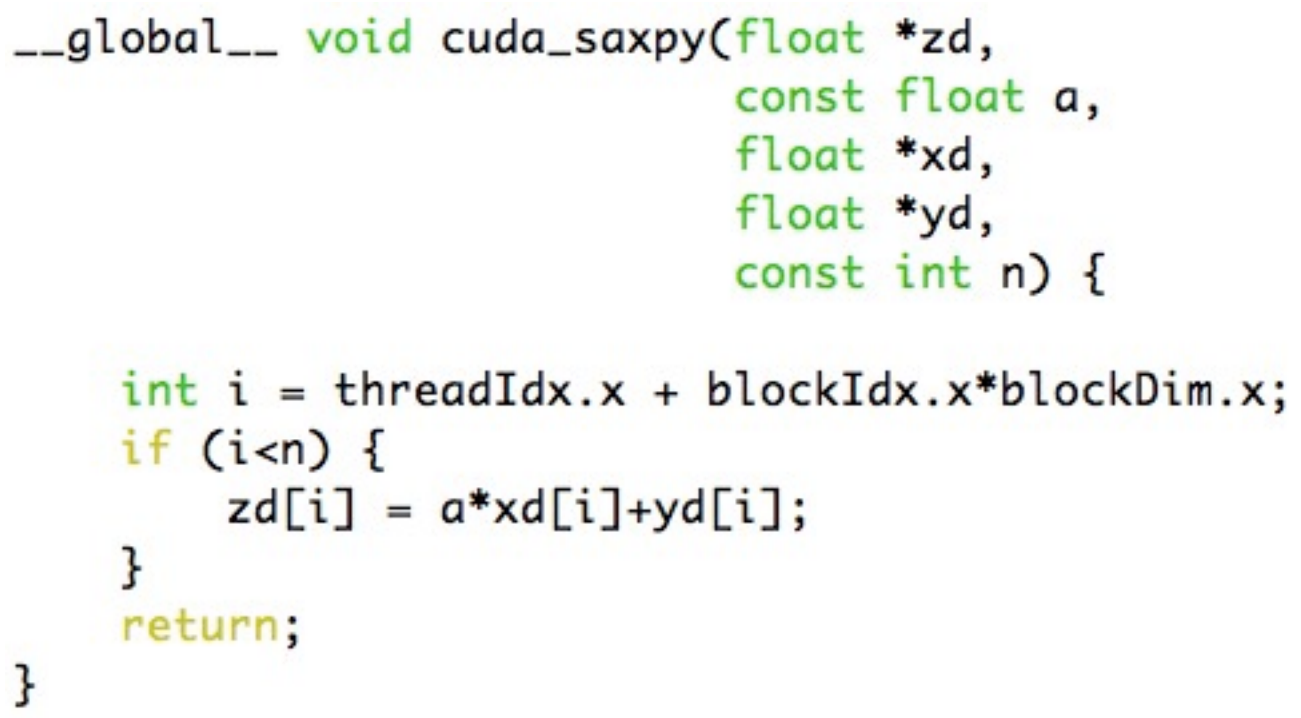
Why the .xs?

- For convenience, CUDA allows thread, block indices 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_saxpy(float *zd,
                           const float a,
                           float *xd,
                           float *yd,
                           const int n) {

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

Why the .xs?

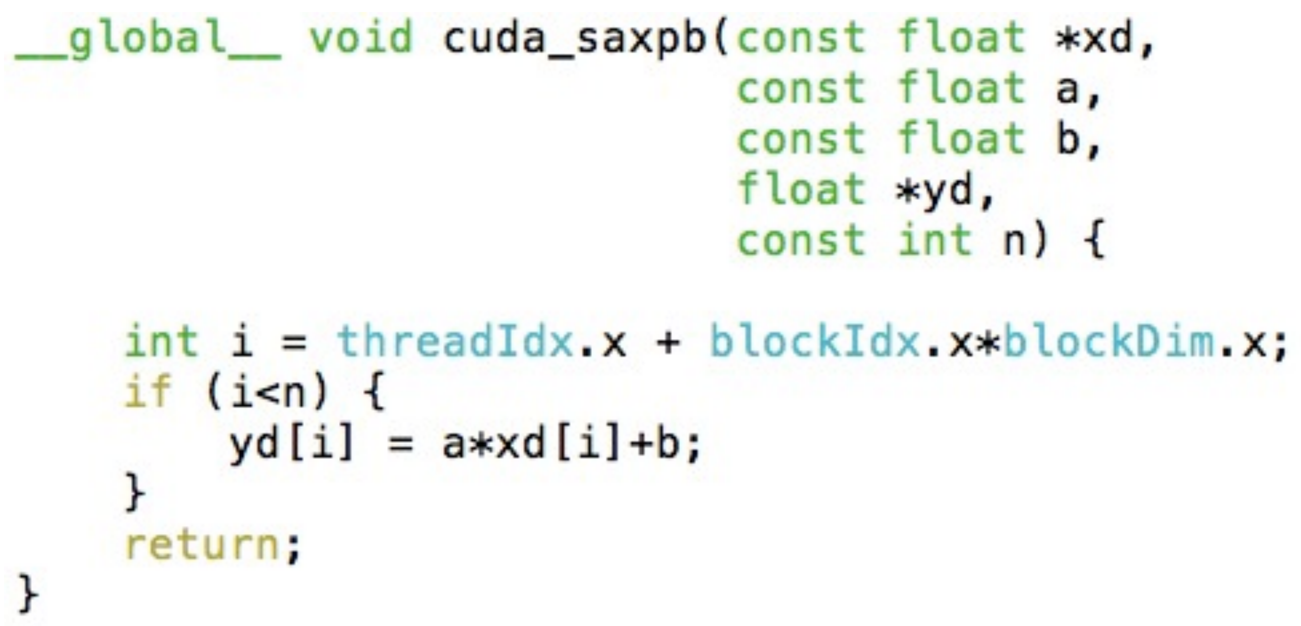
- threadIdx.{x,y,z} - thread index
- blockDim.{x,y,z} - size of block (# of threads in each dim)
- blockIdx.{x,y,z} - block index
- gridDim.{x,y,z} - size of grid (# of blocks in each dim)
- warpsize - size of warp (int)



```
__global__ void cuda_saxpy(float *zd,
                           const float a,
                           float *xd,
                           float *yd,
                           const int n) {

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

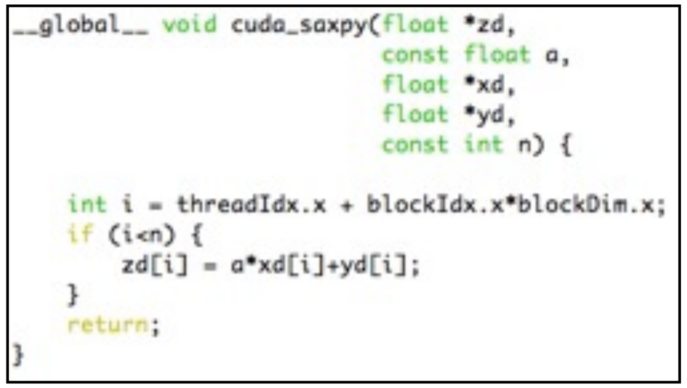
Why the __global__?

- `__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.

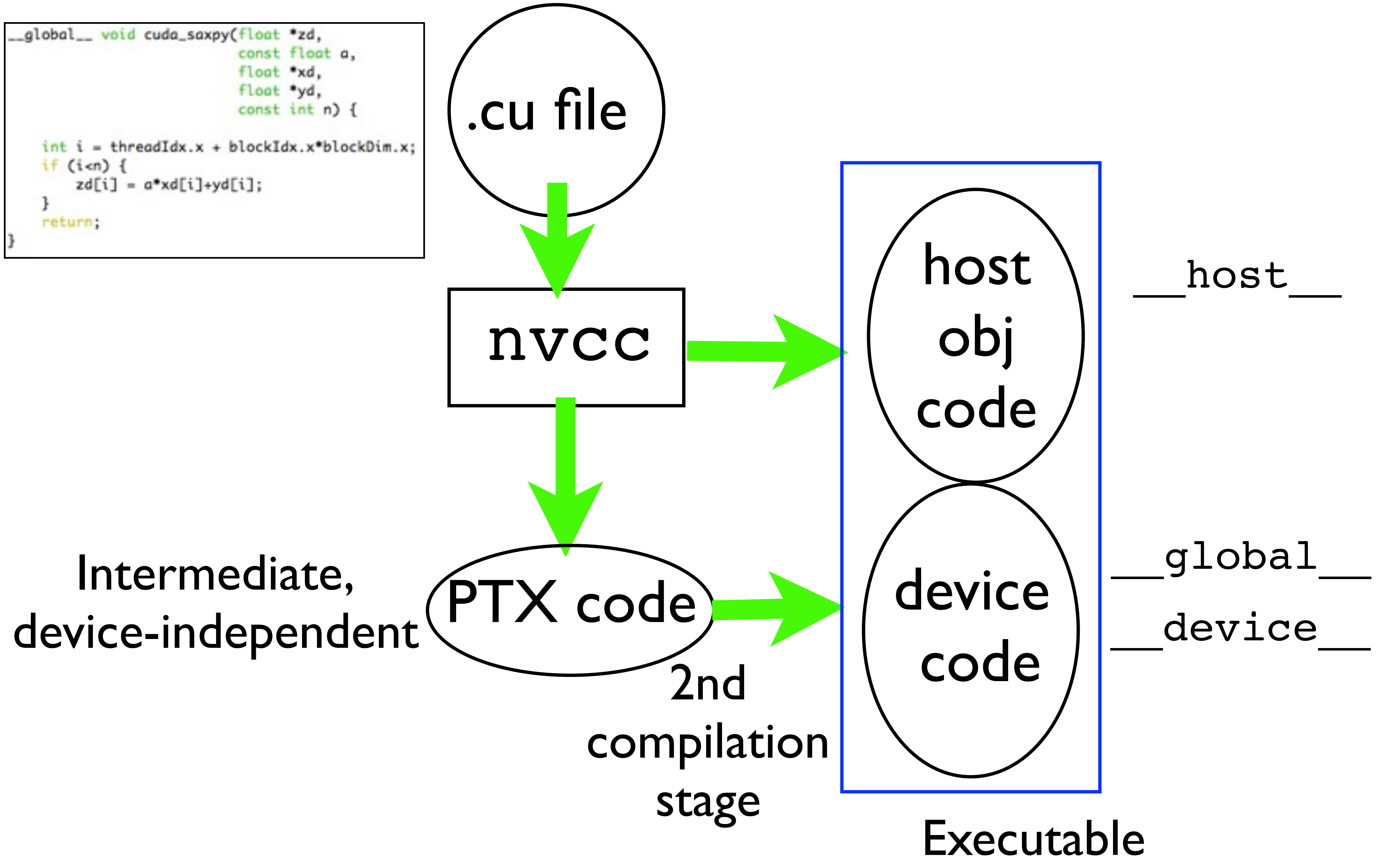


```
__global__ void cuda_saxpb(const float *xd,
                           const float a,
                           const float b,
                           float *yd,
                           const int n) {

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

Compilation process

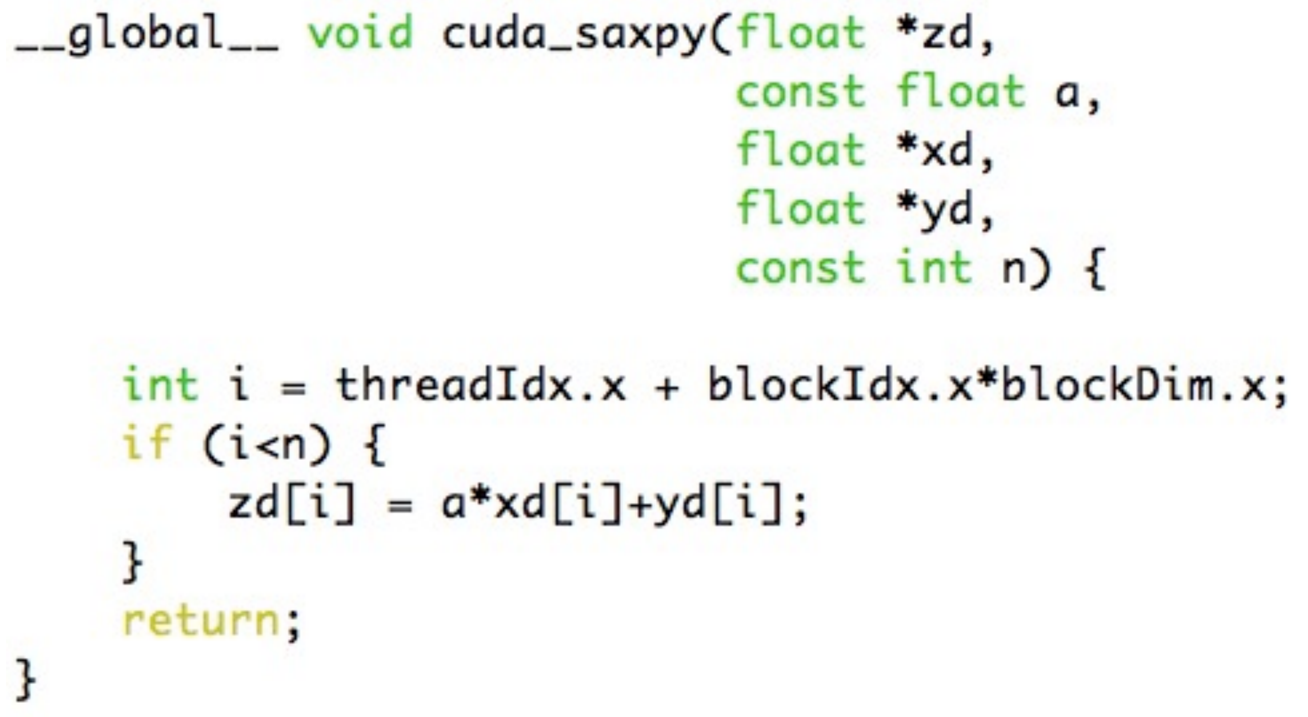


```
__global__ void cuda_saxpy(float *zd,
                           const float a,
                           float *xd,
                           float *yd,
                           const int n) {

    int i = threadIdx.x + blockIdx.x*blockDim.x;
    if (i<n) {
        zd[i] = a*xd[i]+yd[i];
    }
    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_saxpy(float *zd,
                           const float a,
                           float *xd,
                           float *yd,
                           const int n) {

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

Performance

```
arc01-$ ./block-saxpy --nvals=81920 --nblocks=160
Using: n=81920, nblocks=160, niters=5, a=5.000000
CPU time =      2.335 millisec, GFLOPS =   0.07017
GPU time =      0.764 millisec, GFLOPS =    0.2145
CUDA and CPU results differ by 0.000000
```

- Why such poor performance?   x5550 (CPU) ~ 10 GFLOPS. M2070 (GPU) ~ 1000 GFLOPS
- *Arithmetic intensity*.   Each operation involves taking 2 values from memory, doing very simple operation on them (*,+) and then storing a value into memory.
- Memory costs begin to dominate

Memory Bandwidth

```
arc01-$ ./block-saxpy --nvals=81920 --nblocks=160
Using: n=81920, nblocks=160, niters=5, a=5.000000
CPU time =      2.335 millisec, GFLOPS =   0.07017
GPU time =      0.764 millisec, GFLOPS =    0.2145
CUDA and CPU results differ by 0.000000
```

- CPU: 3x(81920) floats read/written in 2.335 ms
  - 401 MB/s
  - Peak ~6GB/sec
  - Max possible flops in this mode: ~1GFLOP (as vs 10)
- GPU:
  - 1227 MB/s
  - Peak ~ 150GB/s
  - Max possible flops in this mode: ~25GFLOP (as vs 1000)

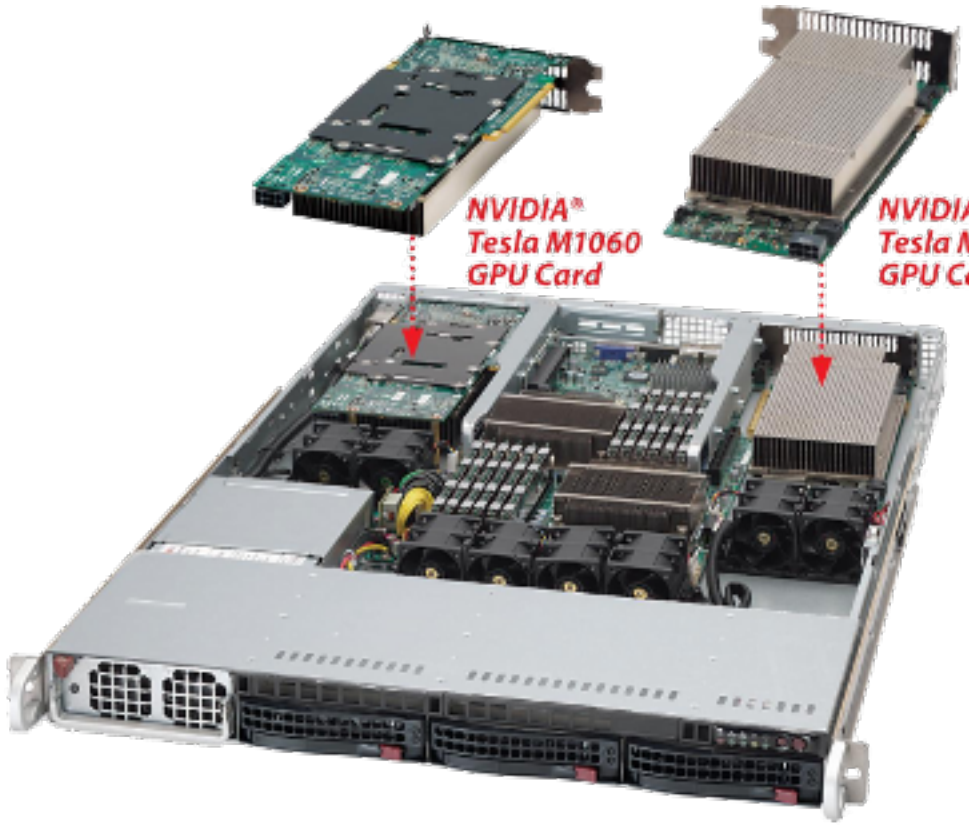
Memory Bandwidth

- For all modern processors, memory access is much more expensive than operating on data once it's local.
- Key to high performance is pulling data from memory into cache, registers, etc and operating on it a *lot* once it is local.

```
arc01-$ ./block-saxpy --nvals=81920 --nblocks=16
Using: n=81920, nblocks=160, niters=5, a=5.00000
CPU time =      2.335 millisec, GFLOPS =   0.0701
GPU time =      0.764 millisec, GFLOPS =    0.214
CUDA and CPU results differ by 0.000000
```

Memory Bandwidth

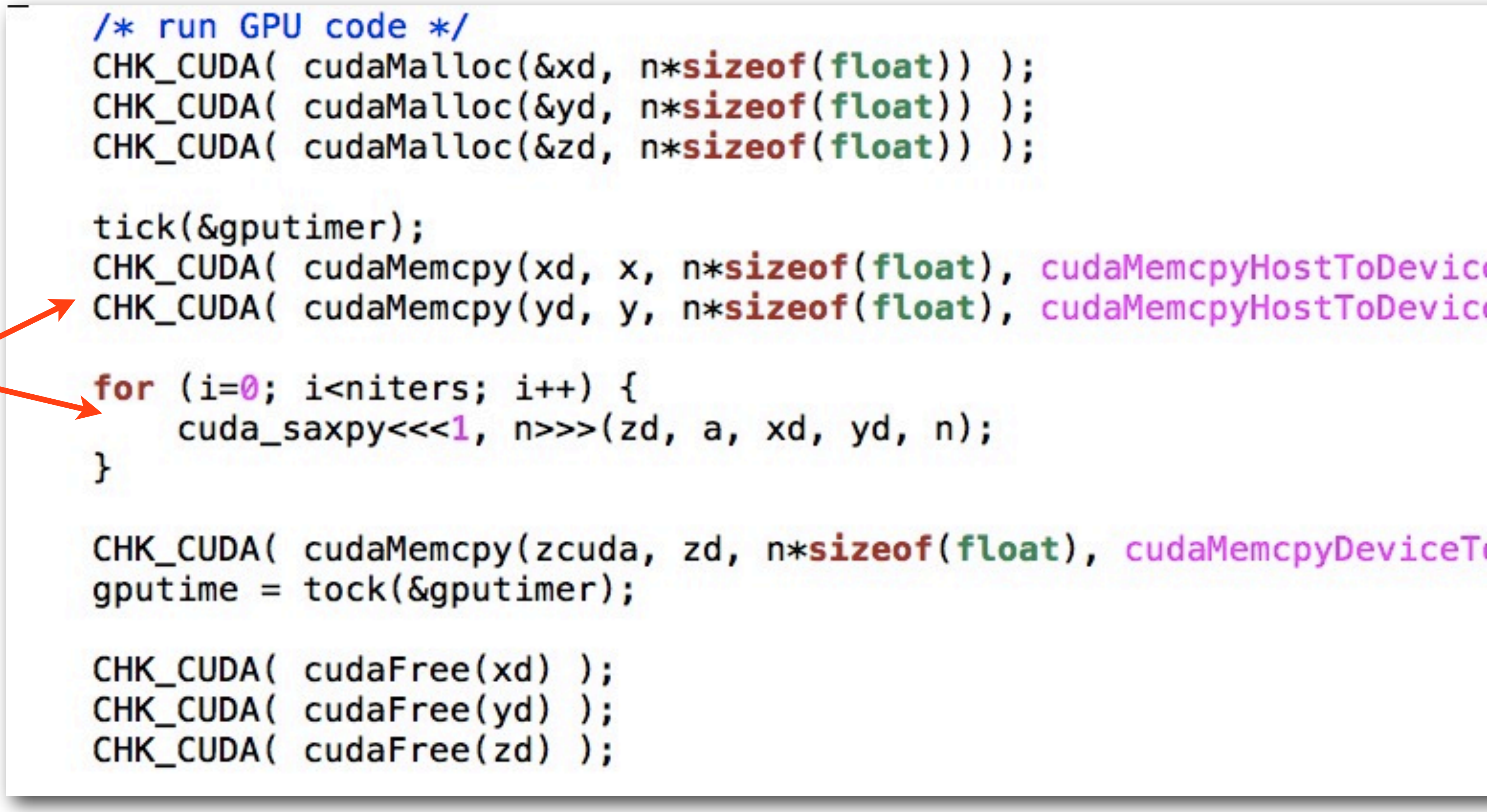
- For GPU, Memory bandwidth is even more important
- Data has to get from host memory to on-card
- PCIe 3.0 16x - 16GB/s
- 1/10 of on-card bandwidth!



http://www.microway.com/tesla/1UGPUchassis.html

Memory Bandwidth

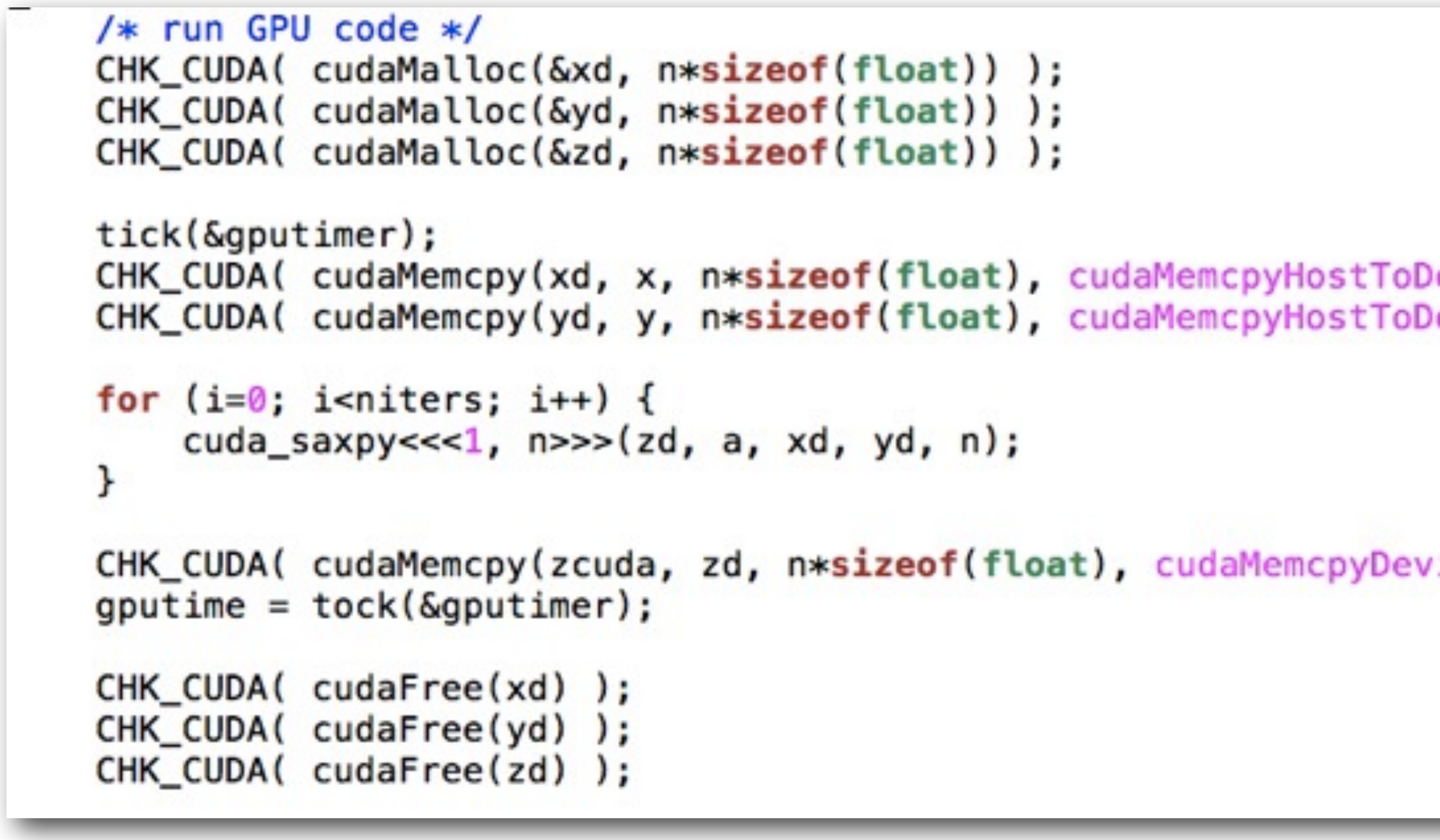
Performance numbers would be much worse if cudaMemcpy's were **in** this loop!!



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

tick(&gputimer);
CHK_CUDA( cudaMemcpy(xd, x, n*sizeof(float), cudaMemcpyHostToDevic
CHK_CUDA( cudaMemcpy(yd, y, n*sizeof(float), cudaMemcpyHostToDevic

for (i=0; i<niters; i++) {
    cuda_saxpy<<<1, n>>>(zd, a, xd, yd, n);
}

CHK_CUDA( cudaMemcpy(zcuda, zd, n*sizeof(float), cudaMemcpyDeviceT
gputime = tock(&gputimer);

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

GPGPU Performance Tip #2

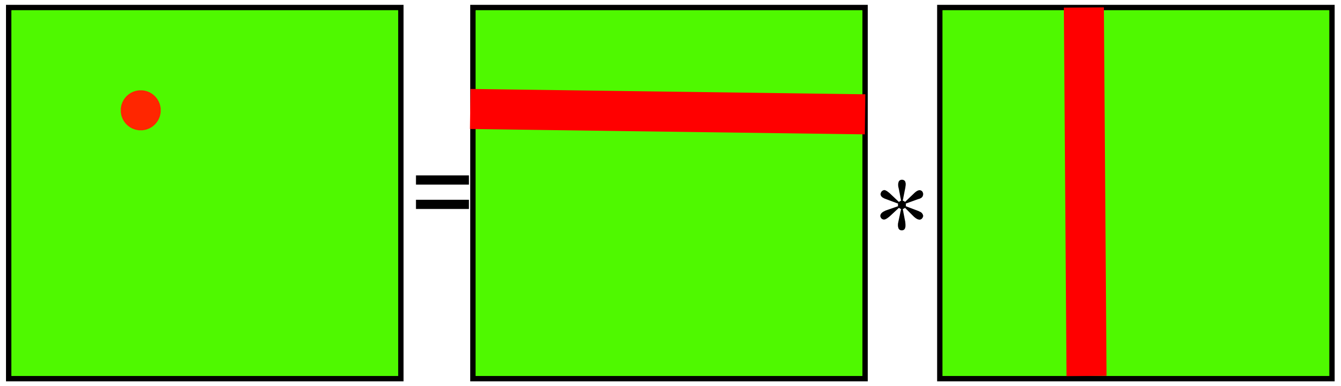
- Wherever possible, avoid copying data back and forth between GPU and CPU.



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

tick(&gputimer);
CHK_CUDA( cudaMemcpy(xd, x, n*sizeof(float), cudaMemcpyHostToD
CHK_CUDA( cudaMemcpy(yd, y, n*sizeof(float), cudaMemcpyHostToD

for (i=0; i<niters; i++) {
    cuda_saxpy<<<1, n>>>(zd, a, xd, yd, n);
}

CHK_CUDA( cudaMemcpy(zcuda, zd, n*sizeof(float), cudaMemcpyDev
gputime = tock(&gputimer);

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

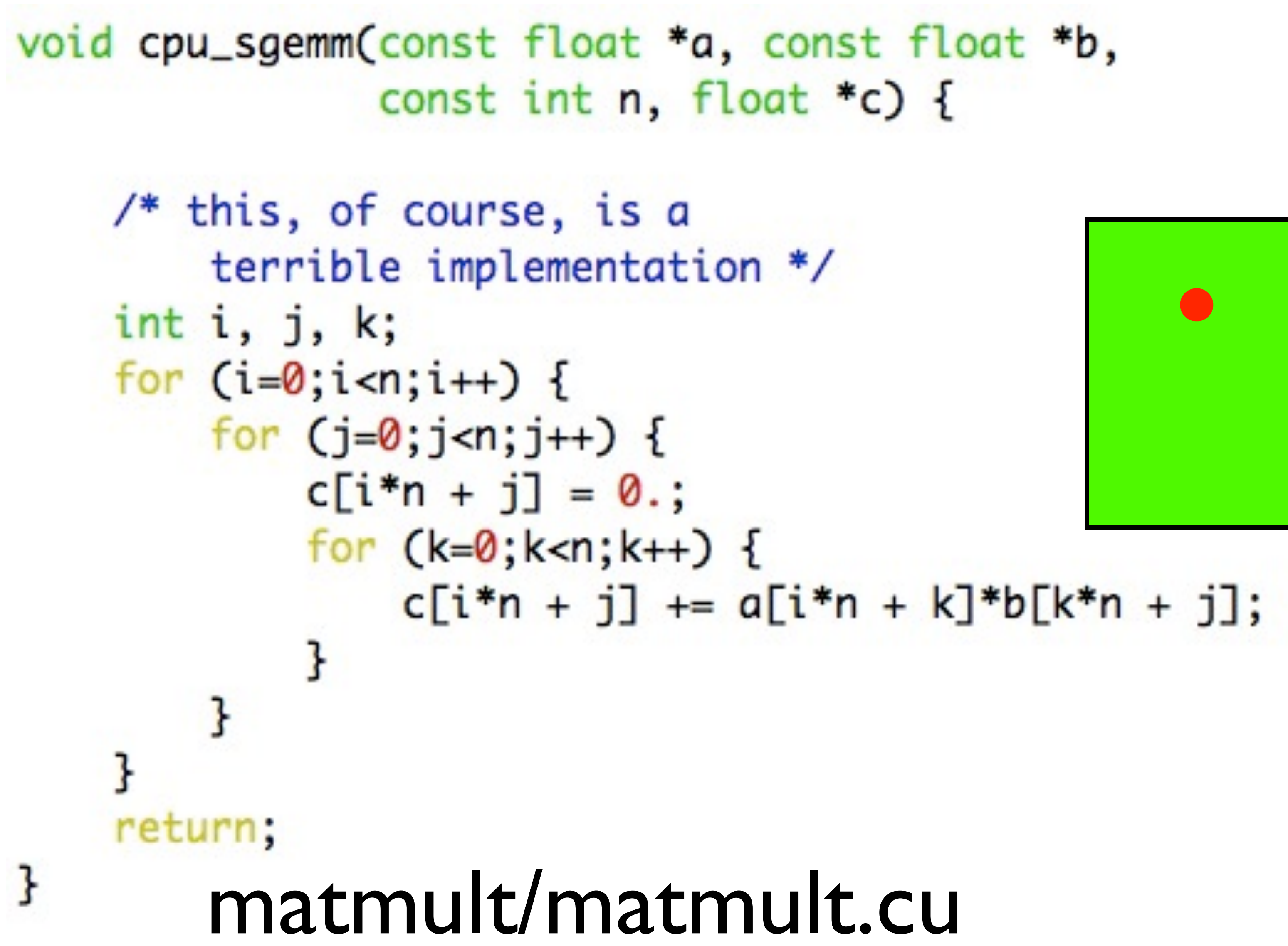
2-Dimensional Blocks

- blockDim.x, threadIdx.x...
- Use of 2/3d thread blocks, or 2d grids, never strictly necessary...
- But can make code clearer, shorter.
- Clearer code = fewer bugs = good.
- Matrix multiplication

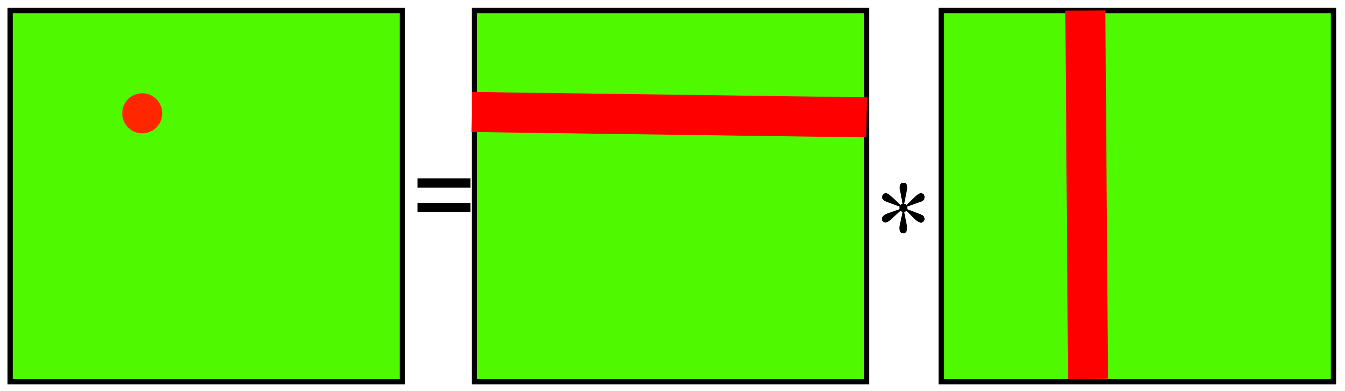


$$C_{i,j} = \sum_k A_{i,k} B_{k,j}$$

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;
    for (i=0;i<n;i++) {
        for (j=0;j<n;j++) {
            c[i*n + j] = 0.;
            for (k=0;k<n;k++) {
                c[i*n + j] += a[i*n + k]*b[k*n + j];
            }
        }
    }
    return;
}
```

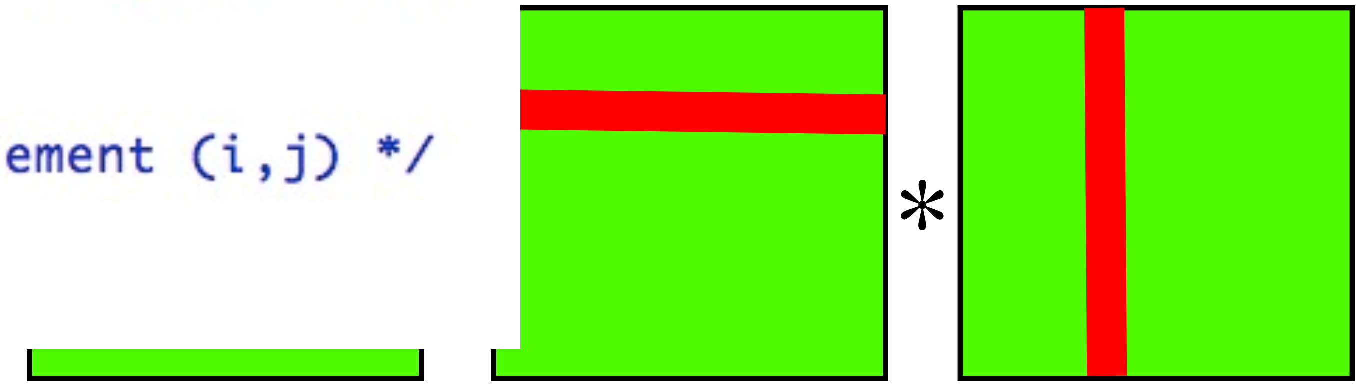


$$C_{i,j} = \sum_k A_{i,k} B_{k,j}$$

matmult/matmult.cu

2-Dimensional Blocks

```
//#define HAVECUDA1
__global__
void cuda_sgemm(const float *ad, const float *bd,
                           const int n, float *cd) {

    int i, j, k;
    /* each thread does one matrix element (i,j) */
    return;
}
```



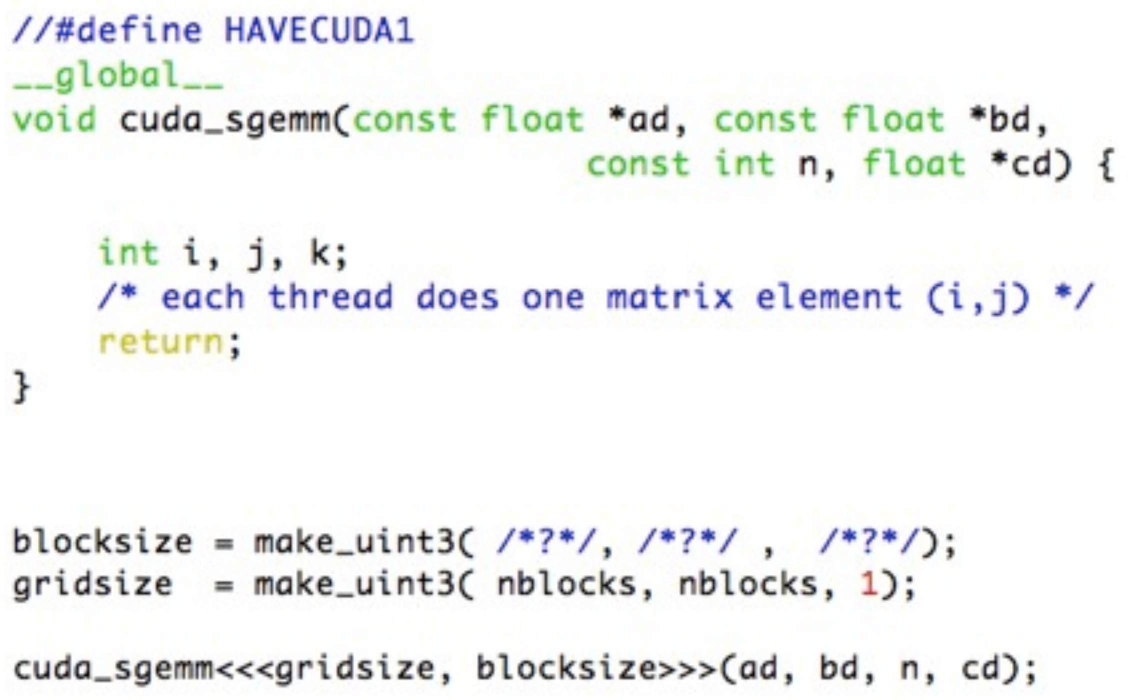
```
blocksize = make_uint3( /*?*/, /*?*/ ,  /*?*/);
gridsize  = make_uint3( nblocks, nblocks, 1);

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

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

How are we going to write the simple CUDA version?

2-Dimensional Blocks

- Hands-on:
  - Fill in the blanks:
    - kernel for cuda_sgemm
    - uncomment #define HAVECUDA1
    - calculate block size
  - Compile, run, compare performance and results
  - Play with different matrix sizes, block numbers



```
//#define HAVECUDA1
__global__
void cuda_sgemm(const float *ad, const float *bd,
                                  const int n, float *cd) {

     int i, j, k;
     /* each thread does one matrix element (i,j) */
     return;
}


blocksize = make_uint3( /*?*/, /*?*/ ,  /*?*/);
gridsize  = make_uint3( nblocks, nblocks, 1);

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

```
#define HAVECUDA1
__global__
void cuda_sgemm(const float *ad, const float *bd,
                           const int n, float *cd) {

    int i, j, k;
    i = threadIdx.y + blockIdx.y*blockDim.y;
    j = threadIdx.x + blockIdx.x*blockDim.x;
    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;
}
```

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

```
arc01-$ ./matmult
Matrix size = 160, Number of blocks = 10.
CPU  time = 36.556 millisec, GFLOPS=0.224095
GPU  time = 0.532 millisec, GFLOPS=15.398496, diff = 0.029795.

arc01-$ ./matmult --matsize=640 --nblocks=40
Matrix size = 640, Number of blocks = 40.
CPU  time = 3008.66 millisec, GFLOPS=0.174260
GPU  time = 13.635 millisec, GFLOPS=38.451632, diff = 1.897964.
```

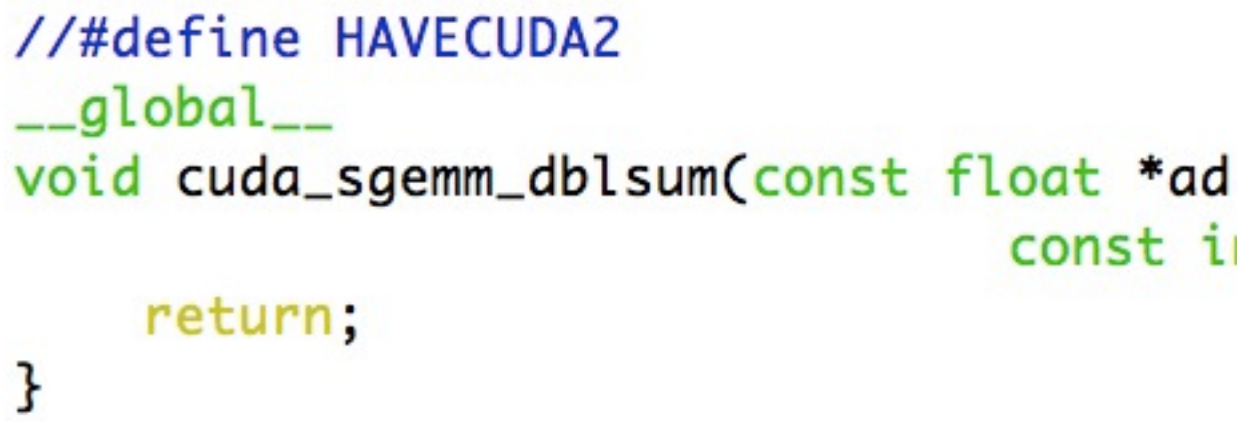
2-Dimensional Blocks

- Good speedup (including memory copy), but results slightly different
- x86: floating pt arithmetic done in registers higher than nominal precision
- Let's fix this by doing math in both kernels with double precision
- cuda_sgemm_dblsum:

```
arc01-$ ./matmult
Matrix size = 160, Number of blocks = 10.
CPU  time = 36.556 millisec, GFLOPS=0.224095
GPU  time = 0.532 millisec, GFLOPS=15.398496, diff = 0.029795.

arc01-$ ./matmult --matsize=640 --nblocks=40
Matrix size = 640, Number of blocks = 40.
CPU  time = 3008.66 millisec, GFLOPS=0.174260
GPU  time = 13.635 millisec, GFLOPS=38.451632, diff = 1.897964.
```

2-Dimensional Blocks

- Hands-on:
  - Fill in the blanks:
    - kernel for cuda_sgemm_dblsum
    - uncomment #define HAVECUDA2
  - Compile, run, compare performance and results
  - Play with different matrix sizes, block numbers



```
//#define HAVECUDA2
__global__
void cuda_sgemm_dblsum(const float *ad
                                const i

    return;
}
```

```
#define HAVECUDA2
__global__
void cuda_sgemm_reg(const float *ad, const float *bd,
                              const int n, float *cd) {

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

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

```
arc01-$ ./matmult --matsize=640 --nblocks=40
Matrix size = 640, Number of blocks = 40.
CPU  time = 3053.9 millisec, GFLOPS=0.171678
GPU  time = 13.635 millisec, GFLOPS=38.451632, diff = 1.897964.
GPU2 time = 10.968 millisec, GFLOPS=47.801605, diff = 0.000000.
```

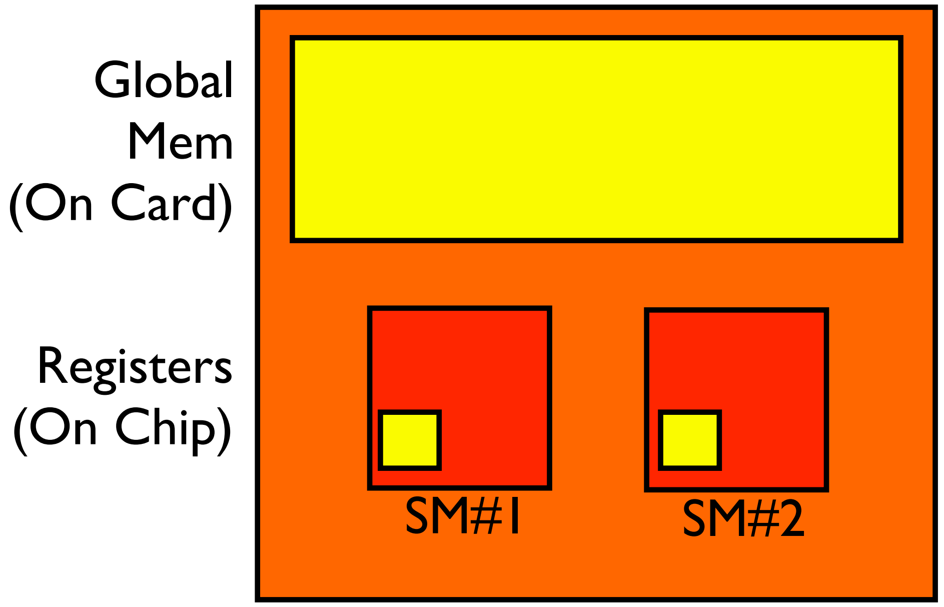
Timings:

```
arc01-$ ./matmult --matsize=640 --nblocks=40
Matrix size = 640, Number of blocks = 40.
CPU  time = 3053.9 millisec, GFLOPS=0.171678
GPU  time = 13.635 millisec, GFLOPS=38.451632, diff = 1.897964.
GPU2 time = 10.968 millisec, GFLOPS=47.801605, diff = 0.000000.
```

Faster, even with double precision sums - why?

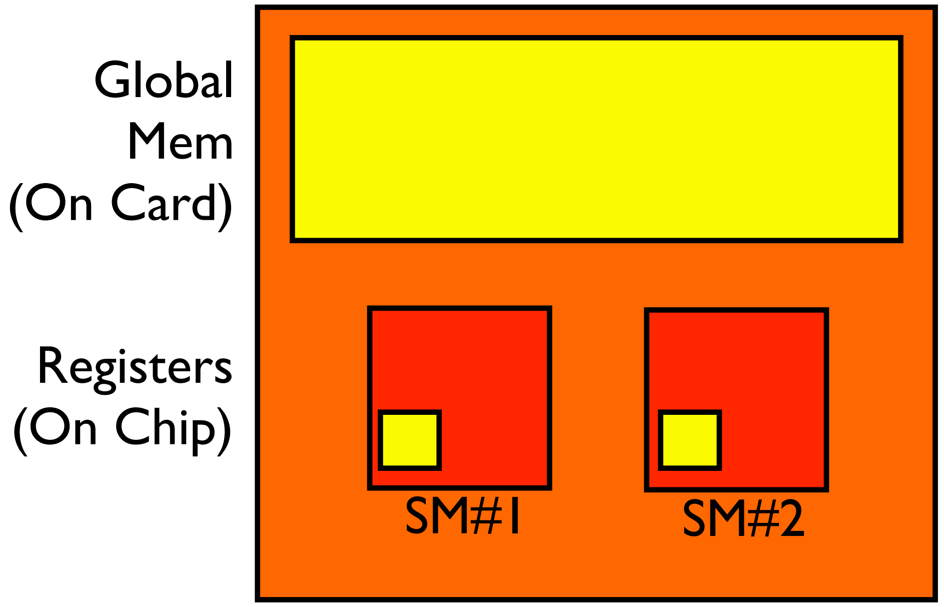
CUDA Memories

- 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

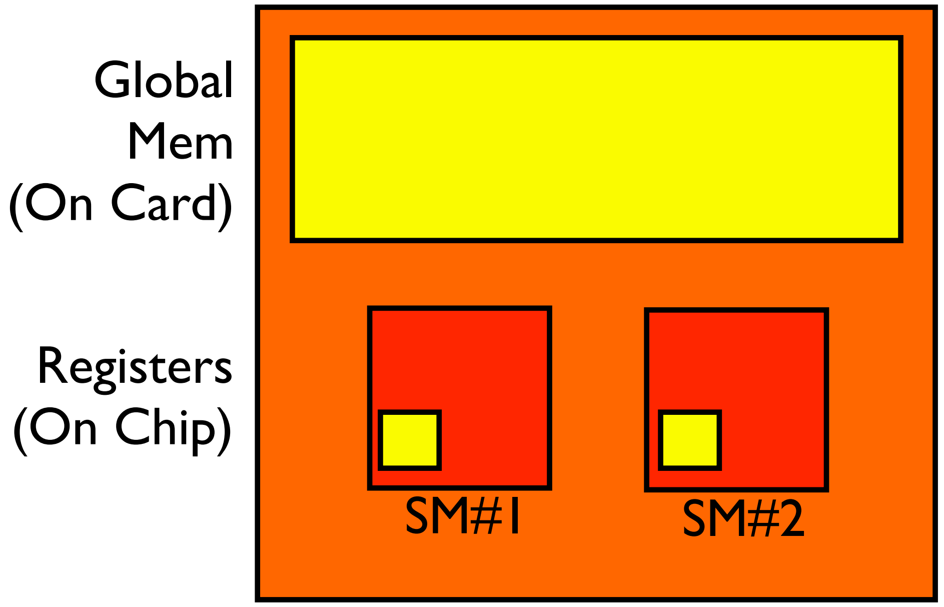
Memory	On Chip?	Cached?	R/W	Scope
Register	On	No	R/W	**Thread**
Shared	On	No	R/W	**Block**
Global	Off	No	R/W	Kernel, Host
Constant	Off	Yes	R	Kernel, Host
Texture	Off	Yes	R(W?)	Kernel, Host
'Local'*	Off	No	R/W	Thread



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

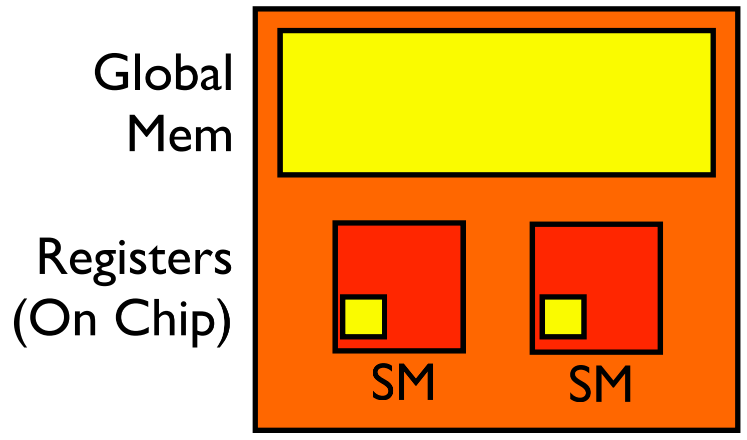
GPGPU Performance Tip #3

- To make the most of the GPU, pull often-used data from large/slow memory to close/small/fast memory
- Tradeoff -- only so much of the fast memory.
- Question - would saxpy benefit from loading data onto on-chip memory first?



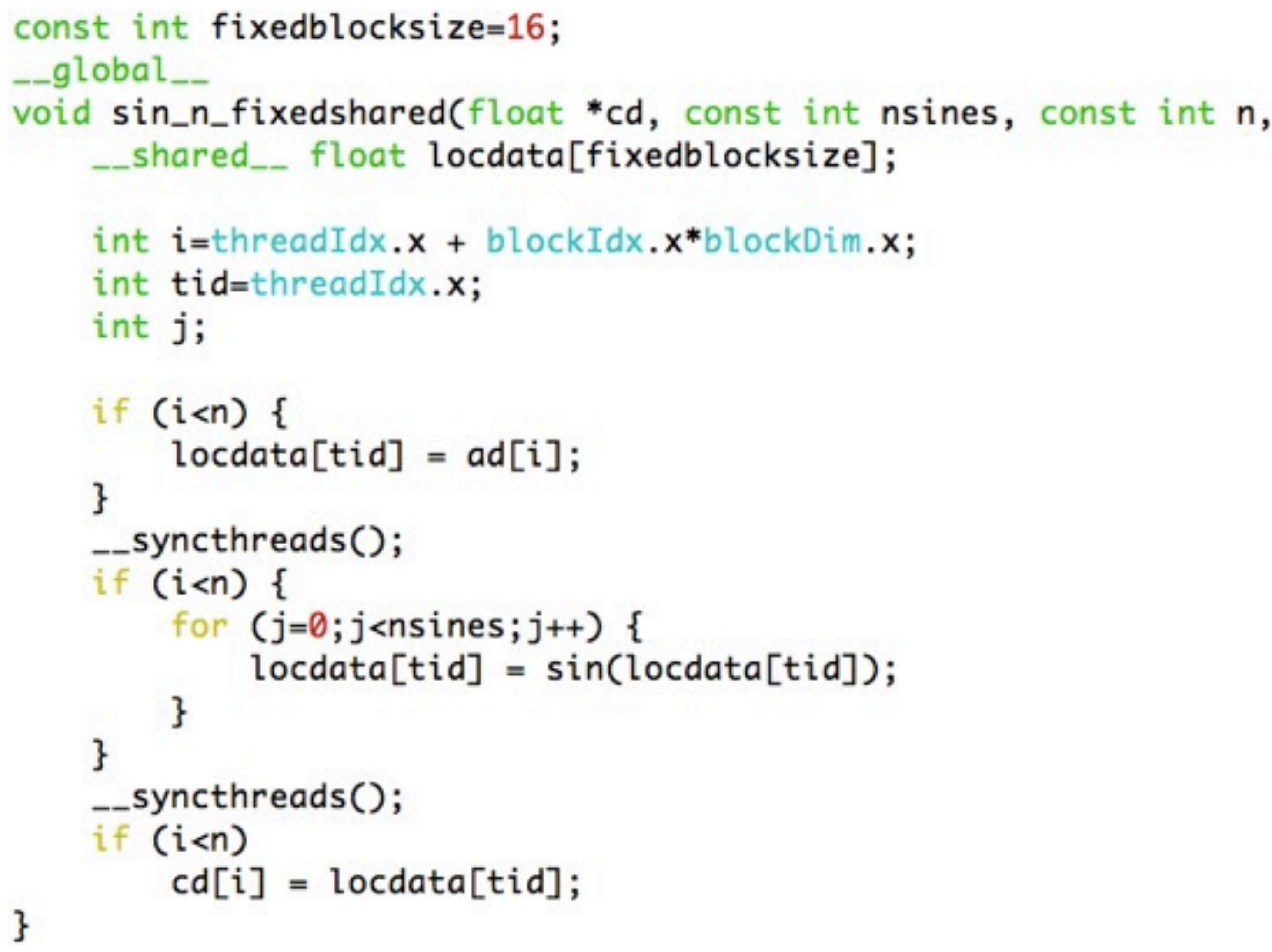
Shared memory

- Registers are great if each thread needs its own
- Shared memory is seen across all threads within a block
- Declared with __shared__
- Can define shared array sizes at compile time or at runtime.



Shared memory

- Silly example: repeatedly take sines of a 1d array.
- Let's put it in a blocksize-sized shared array (much faster than repeatedly using global memory)
- (but could just use register)

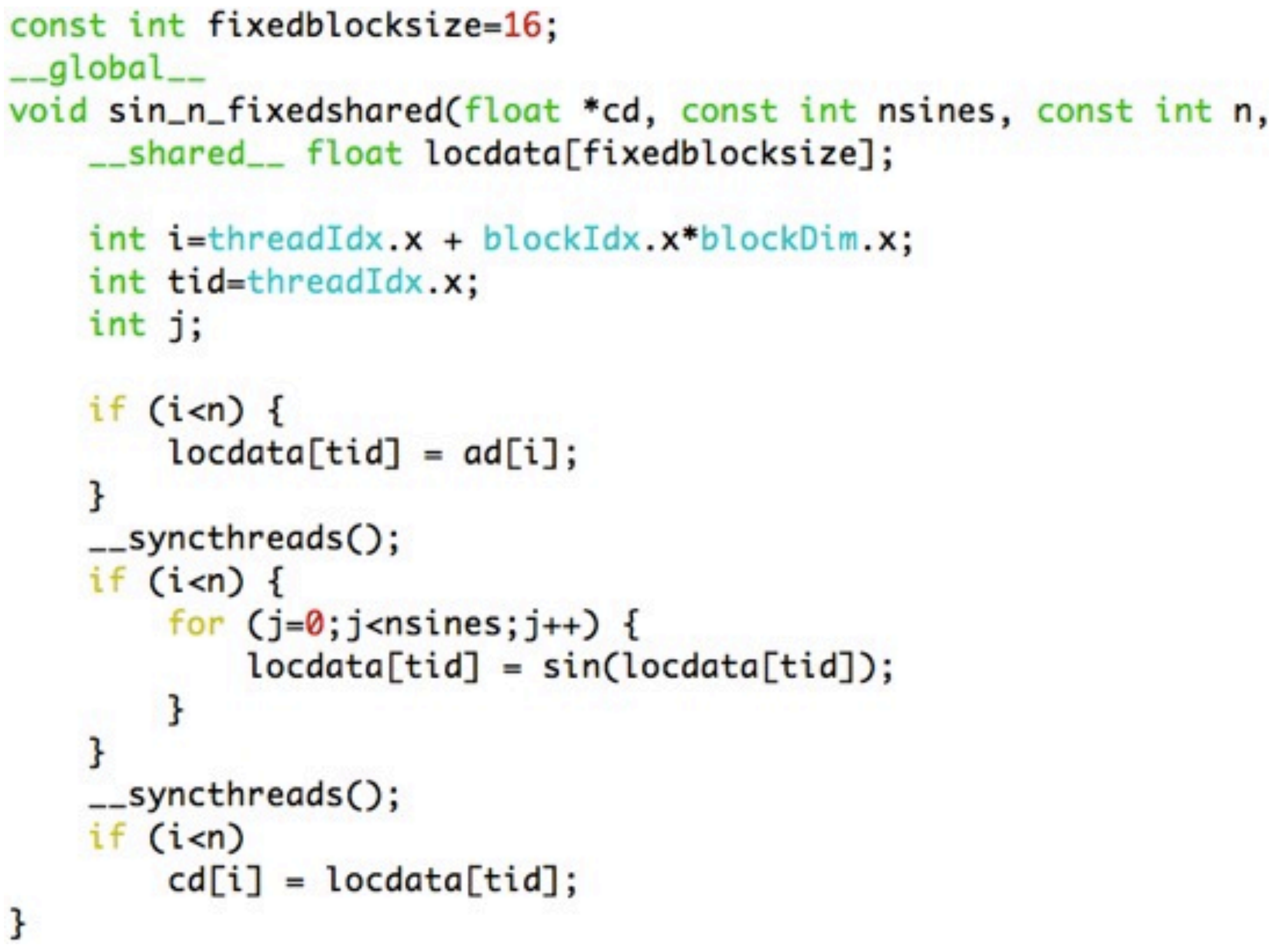


```
const int fixedblocksize=16;
__global__
void sin_n_fixedshared(float *cd, const int nsines, const int n,
    __shared__ float locdata[fixedblocksize];

    int i=threadIdx.x + blockIdx.x*blockDim.x;
    int tid=threadIdx.x;
    int j;

    if (i<n) {
        locdata[tid] = ad[i];
    }
    __syncthreads();
    if (i<n) {
        for (j=0;j<nsines;j++) {
            locdata[tid] = sin(locdata[tid]);
        }
    }
    __syncthreads();
    if (i<n)
        cd[i] = locdata[tid];
}
```

sharedex.cu

Shared memory

- Copy data from global memory (each thread responsible for index i) into shared (responsible for index idx)
- Do computation.

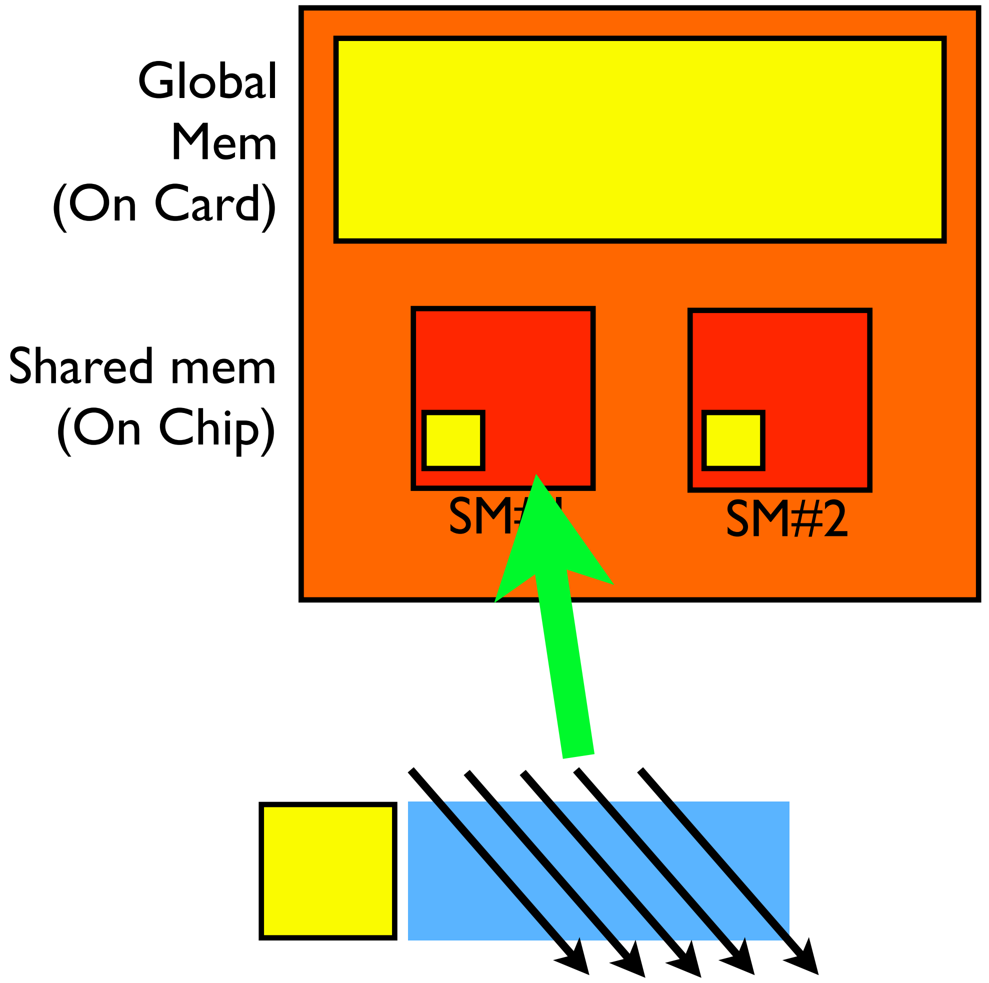


```
const int fixedblocksize=16;
__global__
void sin_n_fixedshared(float *cd, const int nsines, const int n,
    __shared__ float locdata[fixedblocksize];

    int i=threadIdx.x + blockIdx.x*blockDim.x;
    int tid=threadIdx.x;
    int j;

    if (i<n) {
        locdata[tid] = ad[i];
    }
    __syncthreads();
    if (i<n) {
        for (j=0;j<nsines;j++) {
            locdata[tid] = sin(locdata[tid]);
        }
    }
    __syncthreads();
    if (i<n)
        cd[i] = locdata[tid];
}
```

sharedex.cu

__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, or other approach to maintain flexibility

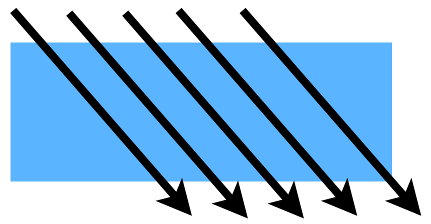


```
const int fixedblocksize=16;
__global__
void sin_n_fixedshared(float *cd, const int nsines, const int n, float *ad) {
    __shared__ float locdata[fixedblocksize];

    int i=threadIdx.x + blockIdx.x*blockDim.x;
    int tid=threadIdx.x;
    int j;

    if (i<n) {
        locdata[tid] = ad[i];
    }
    __syncthreads();
    if (i<n) {
        for (j=0;j<nsines;j++) {
            locdata[tid] = sin(locdata[tid]);
        }
    }
    __syncthreads();
    if (i<n)
        cd[i] = locdata[tid];
}

sin_n_fixedshared<<<gridsize, blocksize>>>(cd, N, n, ad);
```

sharedex.cu

__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



Atomic operations

- When accessing shared memory, must be sure multiple threads are *not* updating same value at same time
- Overwrite or worse!
- Race condition
- Some atomic operations. *Serialize* results; only if no other way

```
int atomicAdd(int* address, int val);
```

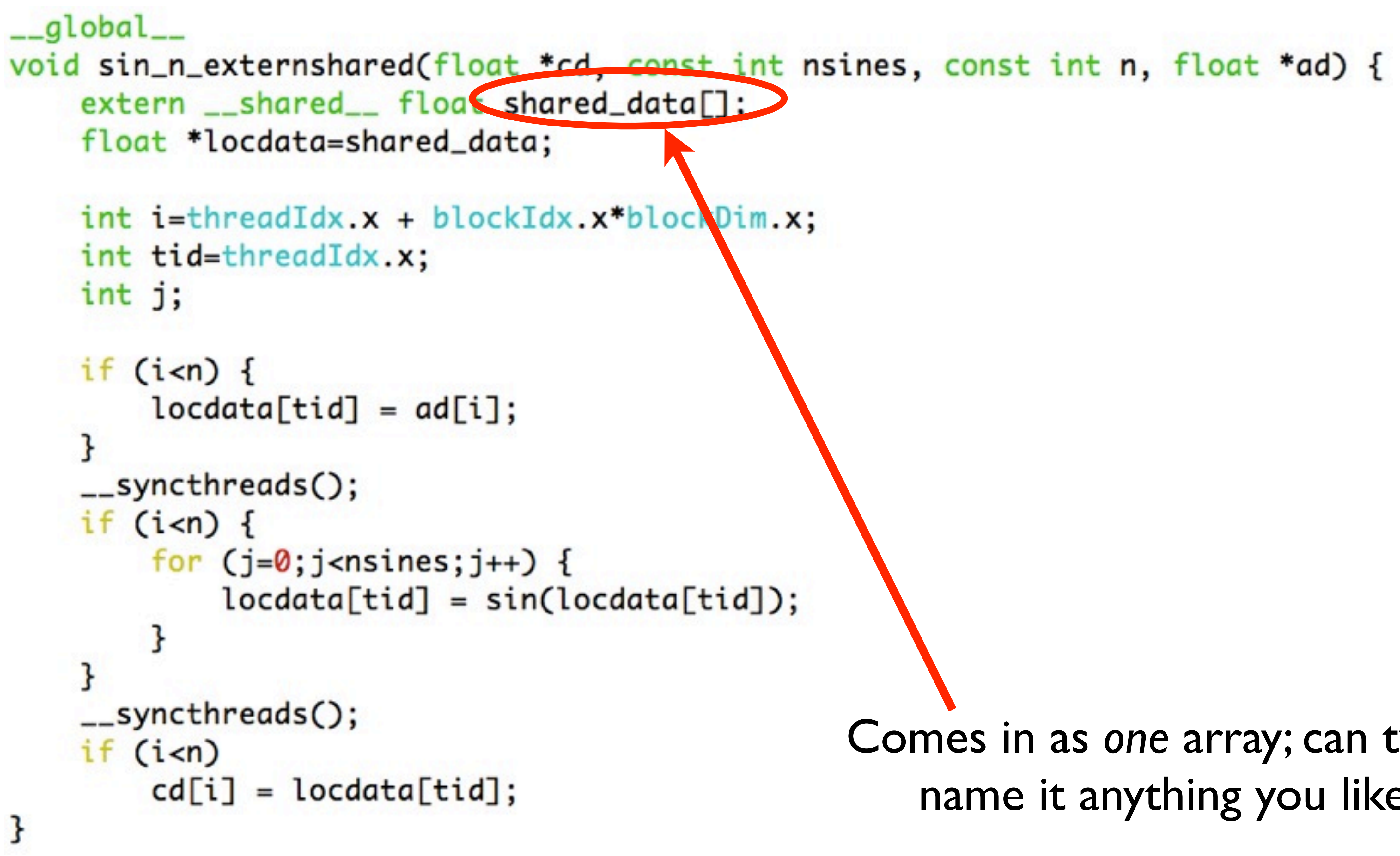
```
__global__
void sin_n_externshared(float *cd, const int nsines, const int n, float *ad) {
    extern __shared__ float shared_data[];
    float *locdata=shared_data;

    int i=threadIdx.x + blockIdx.x*blockDim.x;
    int tid=threadIdx.x;
    int j;

    if (i<n) {
        locdata[tid] = ad[i];
    }
    __syncthreads();
    if (i<n) {
        for (j=0;j<nsines;j++) {
            locdata[tid] = sin(locdata[tid]);
        }
    }
    __syncthreads();
    if (i<n)
        cd[i] = locdata[tid];
}

sin_n_externshared<<<gridsize, blocksize, blocksize.x*sizeof(float)>>>(cd, N, n, a
```

sharedex.cu

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

SciNet

extern __shared__

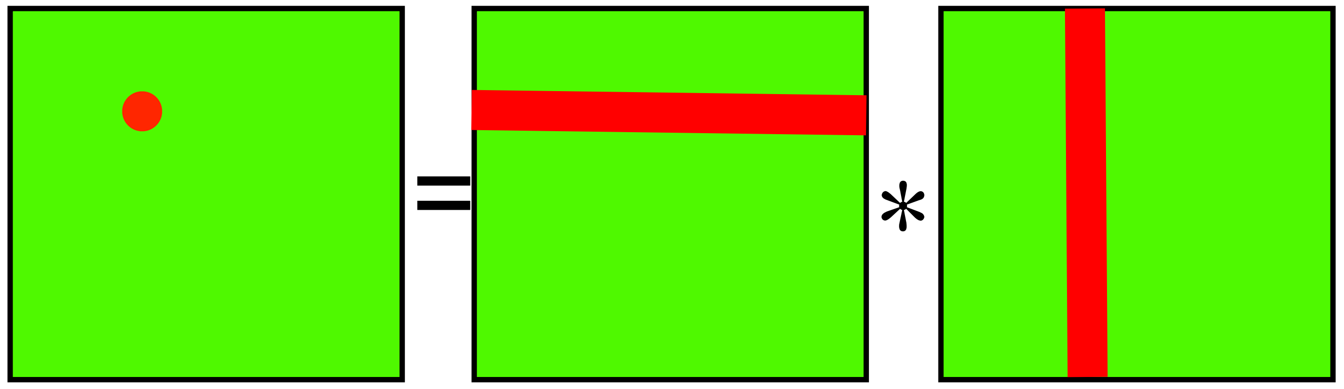


```
__global__
void sin_n_externshared(float *cd, const int nsines, const int n, float *ad) {
    extern __shared__ float shared_data[];
    float *locdata=shared_data;

    int i=threadIdx.x + blockIdx.x*blockDim.x;
    int tid=threadIdx.x;
    int j;

    if (i<n) {
        locdata[tid] = ad[i];
    }
    __syncthreads();
    if (i<n) {
        for (j=0;j<nsines;j++) {
            locdata[tid] = sin(locdata[tid]);
        }
    }
    __syncthreads();
    if (i<n)
        cd[i] = locdata[tid];
}
```

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

Memory usage in SGEMM

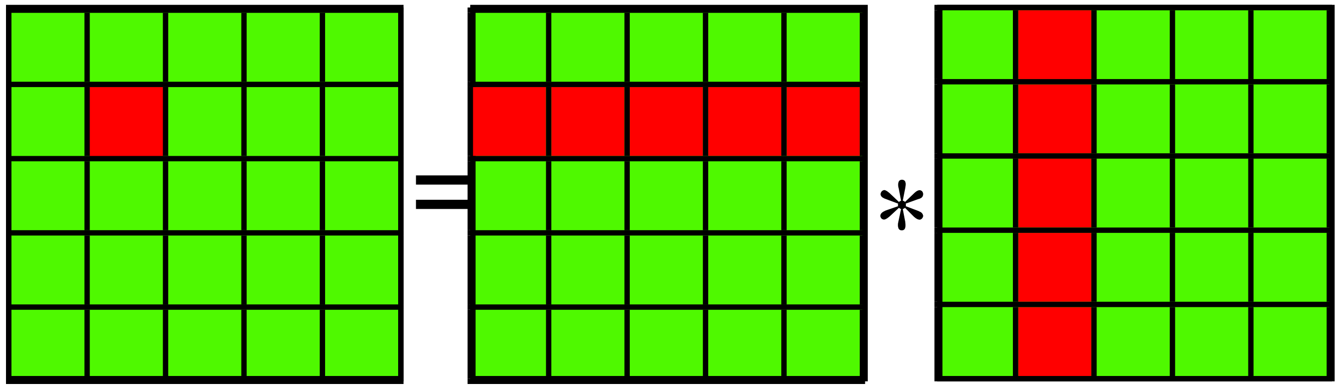
- How can we exploit this?
- N^3 multiplies, adds
- 2N^2 data
- Regular access
- Opportunity for high memory re-use
- Need to find ways to bring data into shared memory (incurring global mem overhead once), use it several times



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

Memory usage in SGEMM

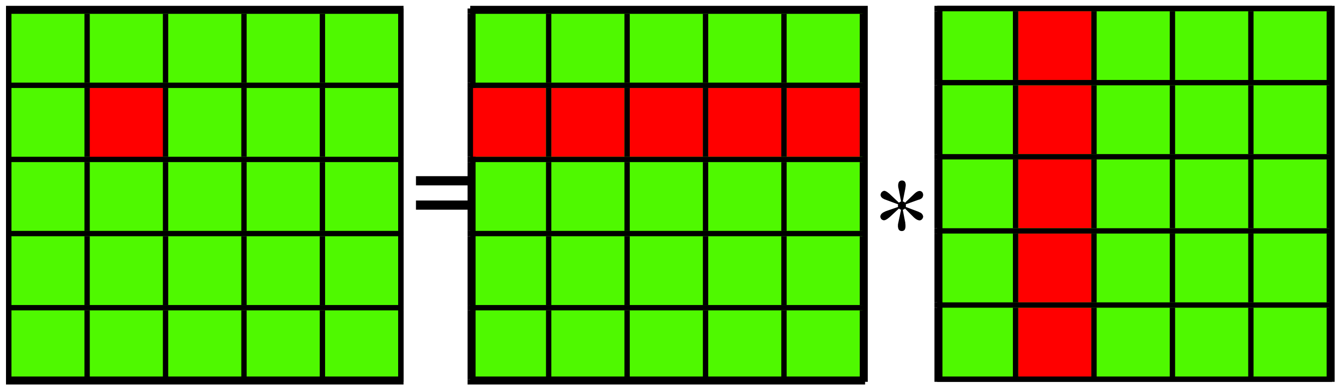
- One nice thing about matrix multiplication - same as block multiplication, each sub-block is a matrix mult
- Neighbouring threads within block all see nearby rows, columns
- Pull whole block in
- If *b* blocks in each dim, each data only pulled in *2b* times, not *2n* times



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

Hands on

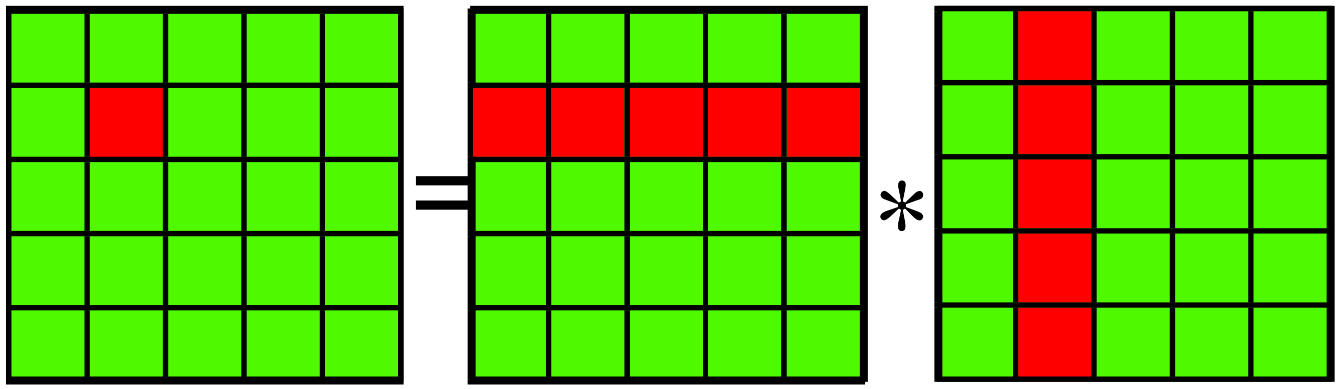
- Change one of the matrix multiplier kernels to use shared memory
- use fixed blocksize if you like (easier)
- Assume blocksize divides matrix size (easier)
- Two "tiles" of A and B, and loop from k=0..n/(blocksize) to do block matrix mult.



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

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;
```



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

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
```

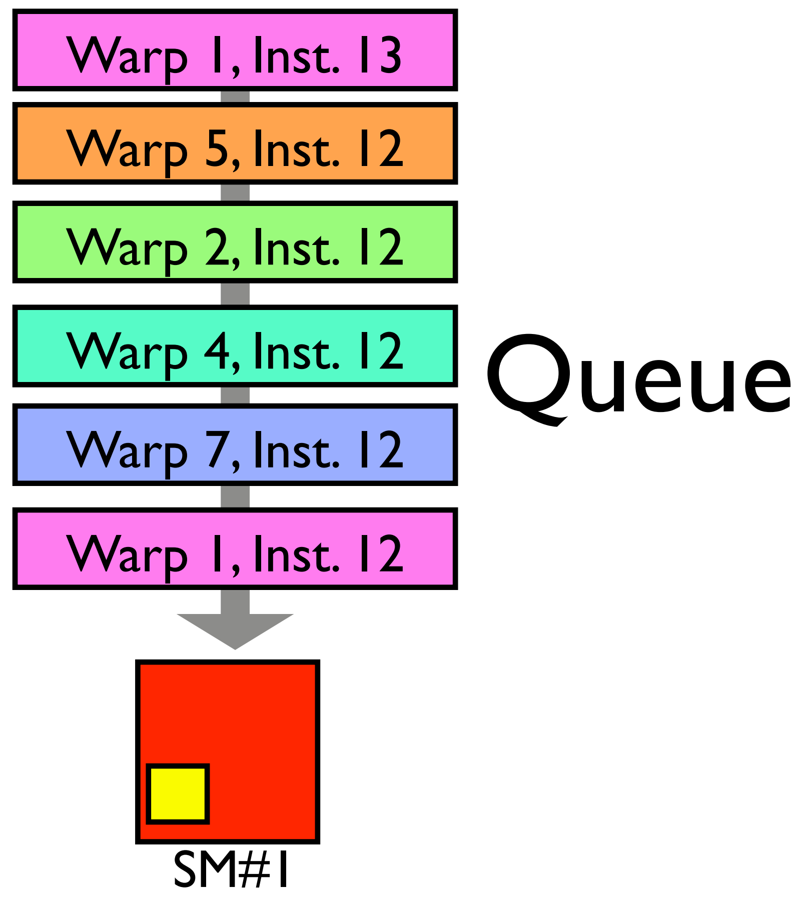
Making effective use of CUDA memories

- 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

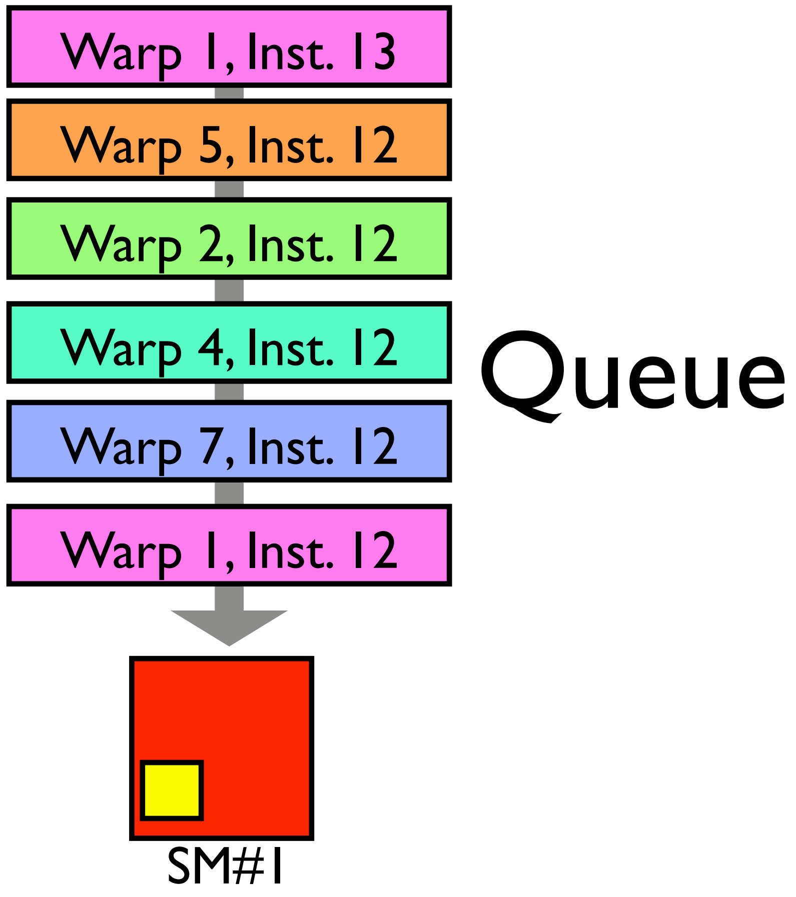
Stalling on Memory Access

- 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.



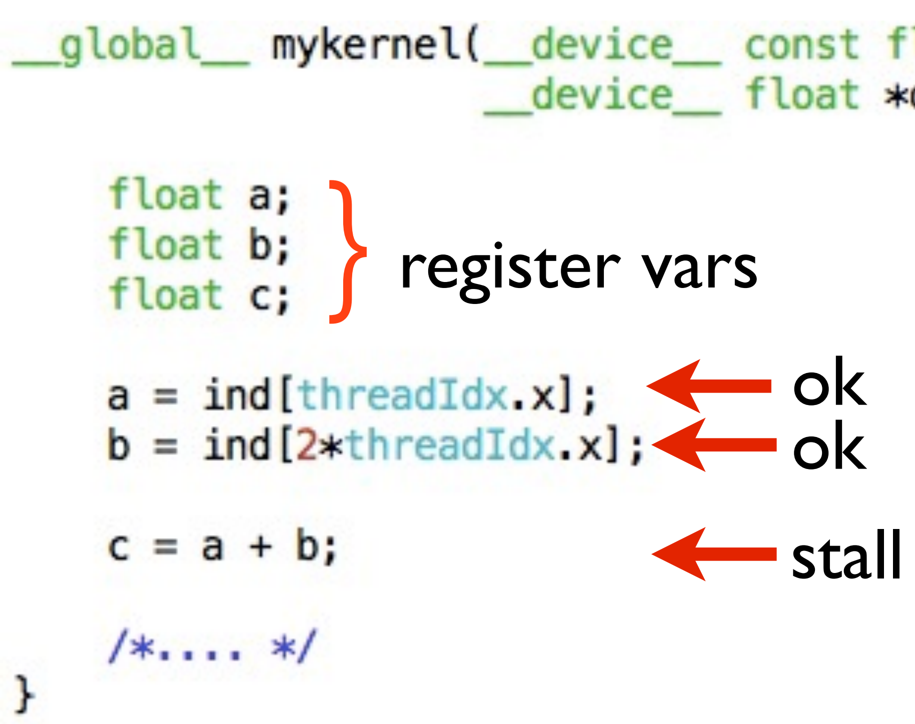
Stalling on Memory Access

- 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.



Stalling happens on *use*.

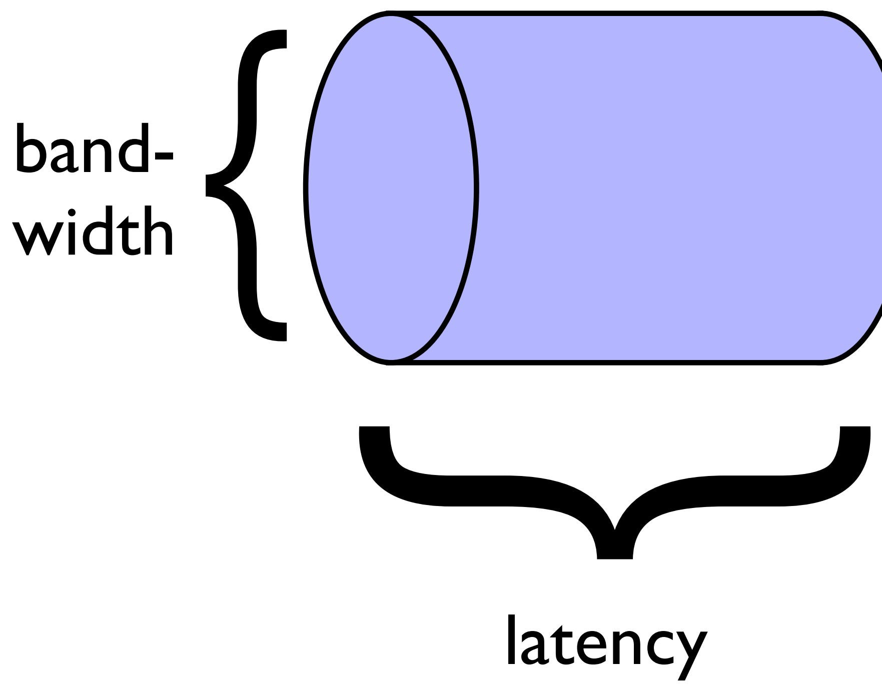
- Kernel does not stall on loading data
- Stalls when data not yet ready needs to be used
- Can “preload” data that you will need at beginning of kernel
- Hide latency by doing as much work as possible before need bulk of data.



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

    float a;
    float b;
    float c;

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

    c = a + b;

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

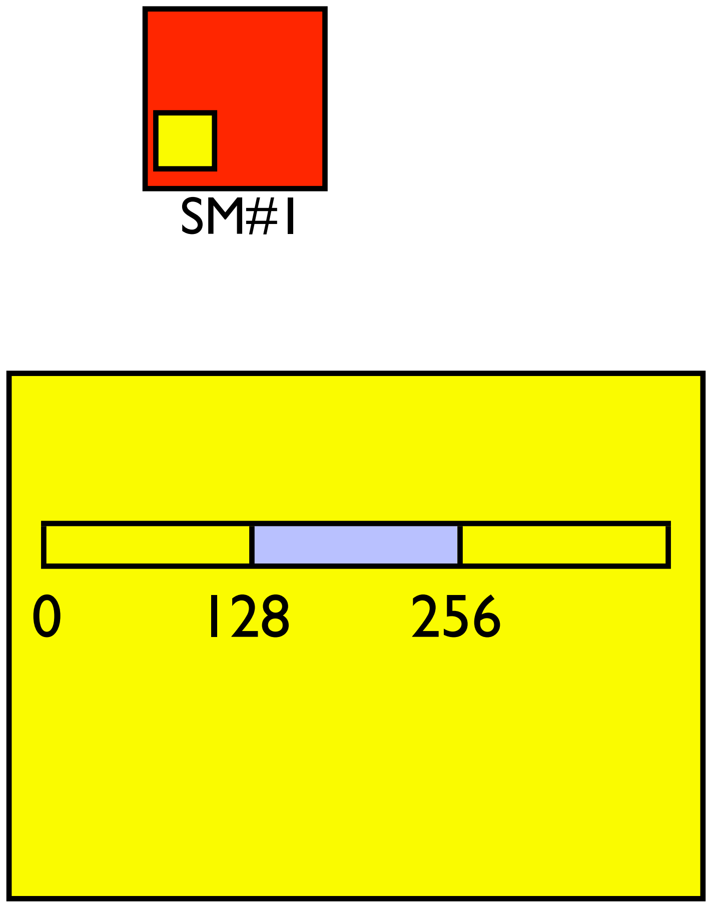
Keep memory accesses going

- 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



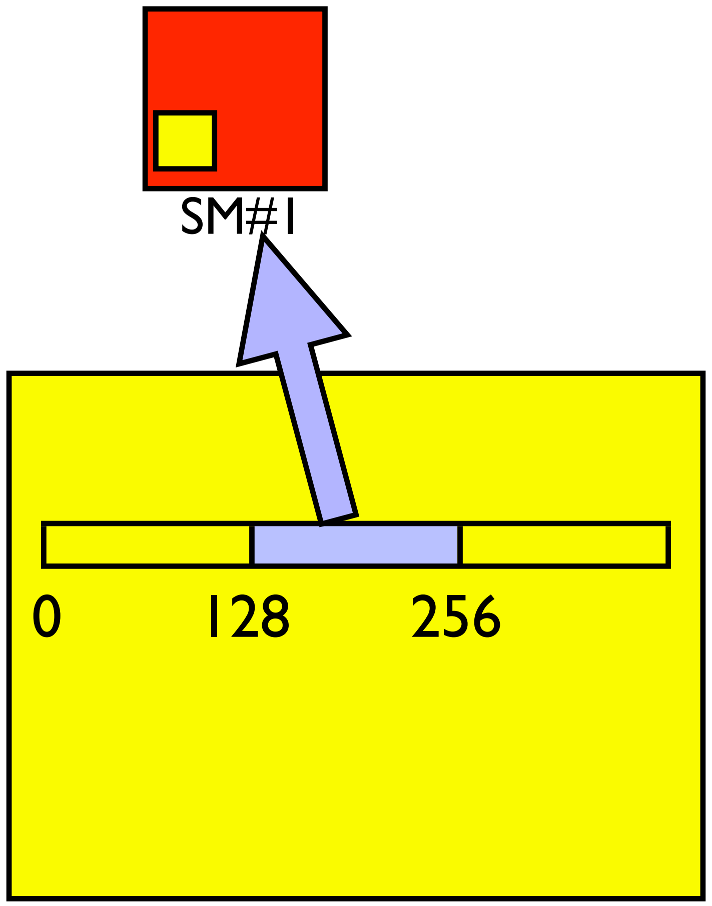
Coalesced Memory Access

- 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



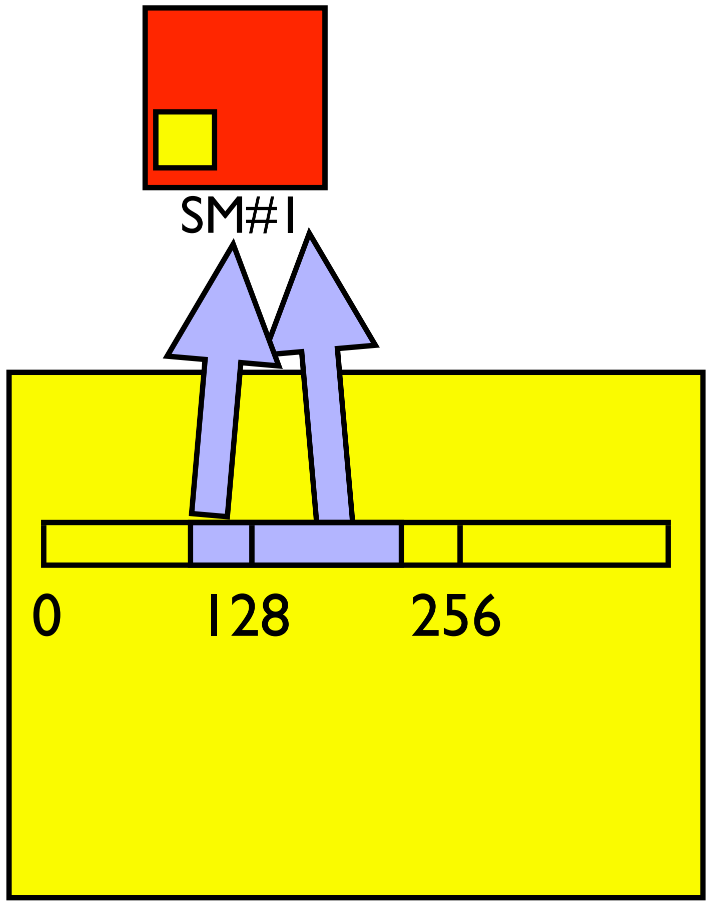
Coalesced Memory Access

- 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



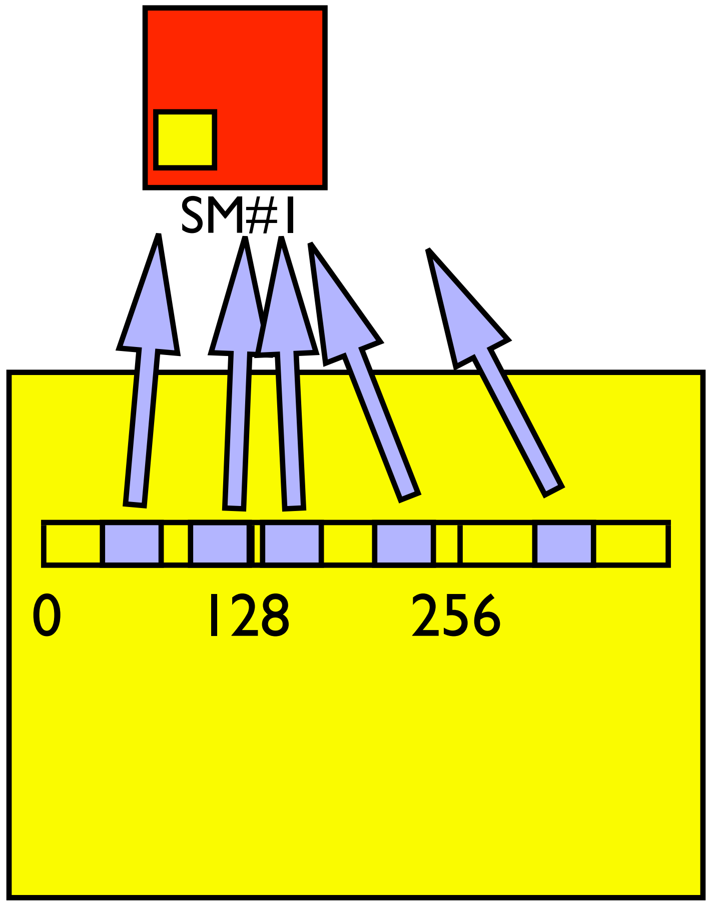
Coalesced Memory Access

- 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)



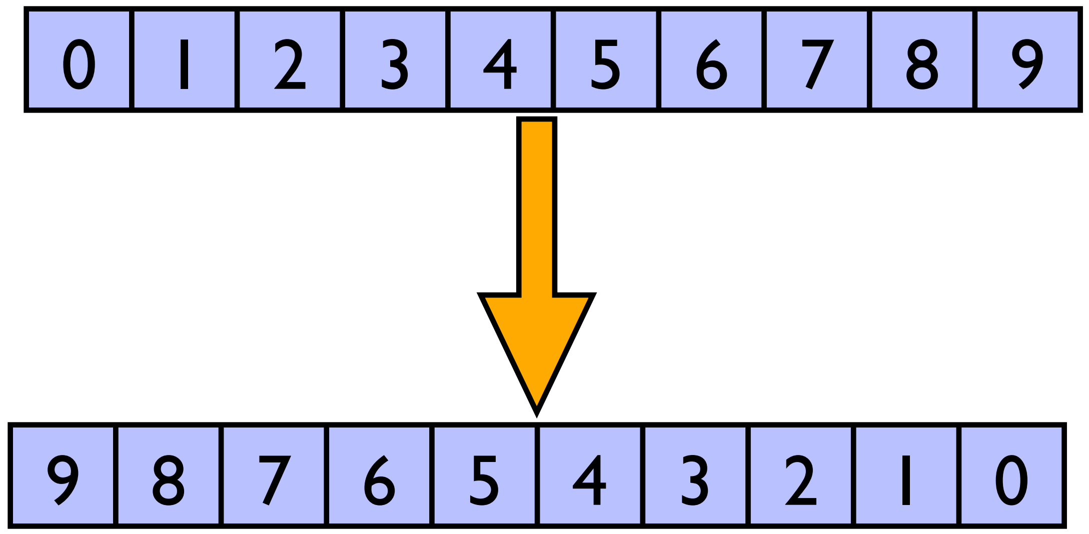
Coalesced Memory Access

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

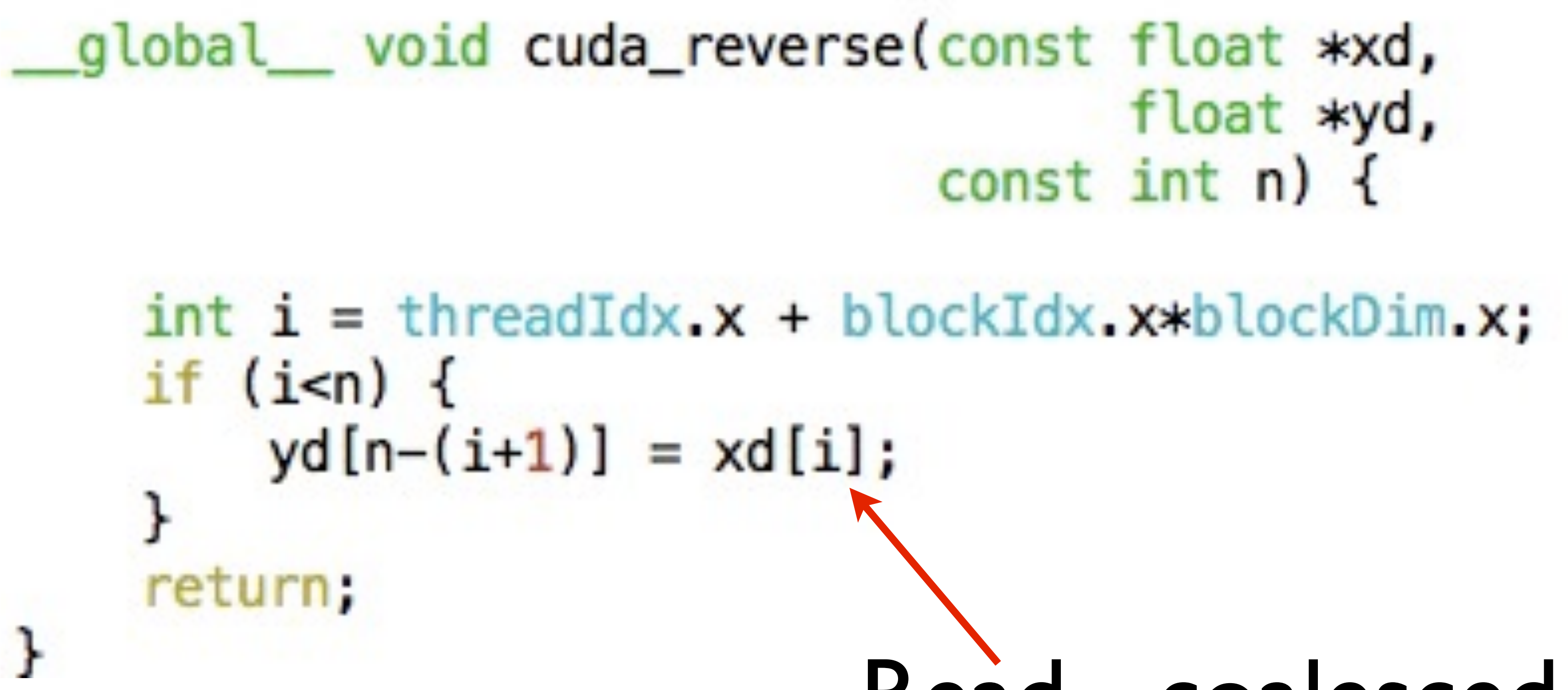


List reversal

- 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:



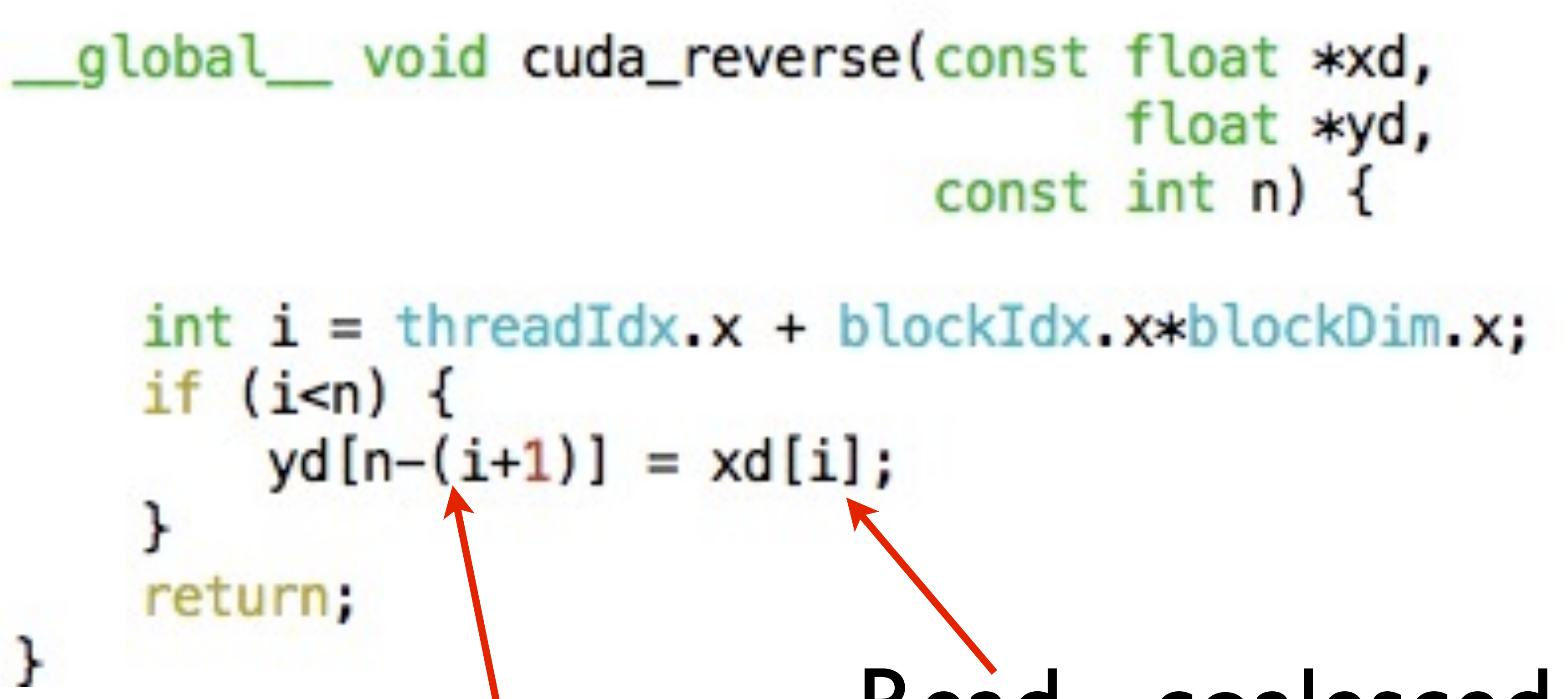
List reversal



```
__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

List reversal

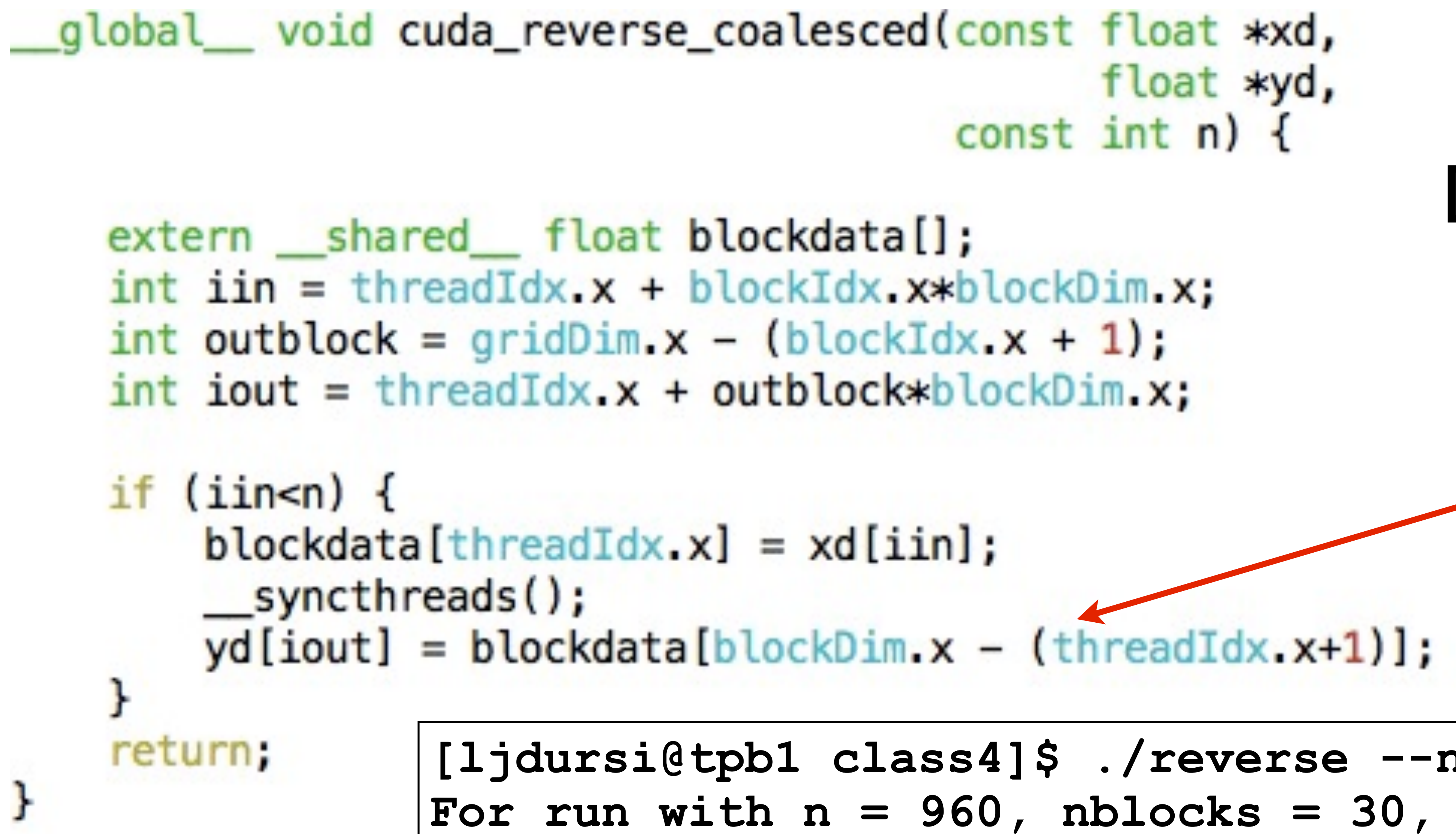


```
__global__ void cuda_reverse(const float *xd,
                                   float *yd,
                             const int n) {

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

Read - coalesced

Write - reversed - possibly noncoalesced

List reversal



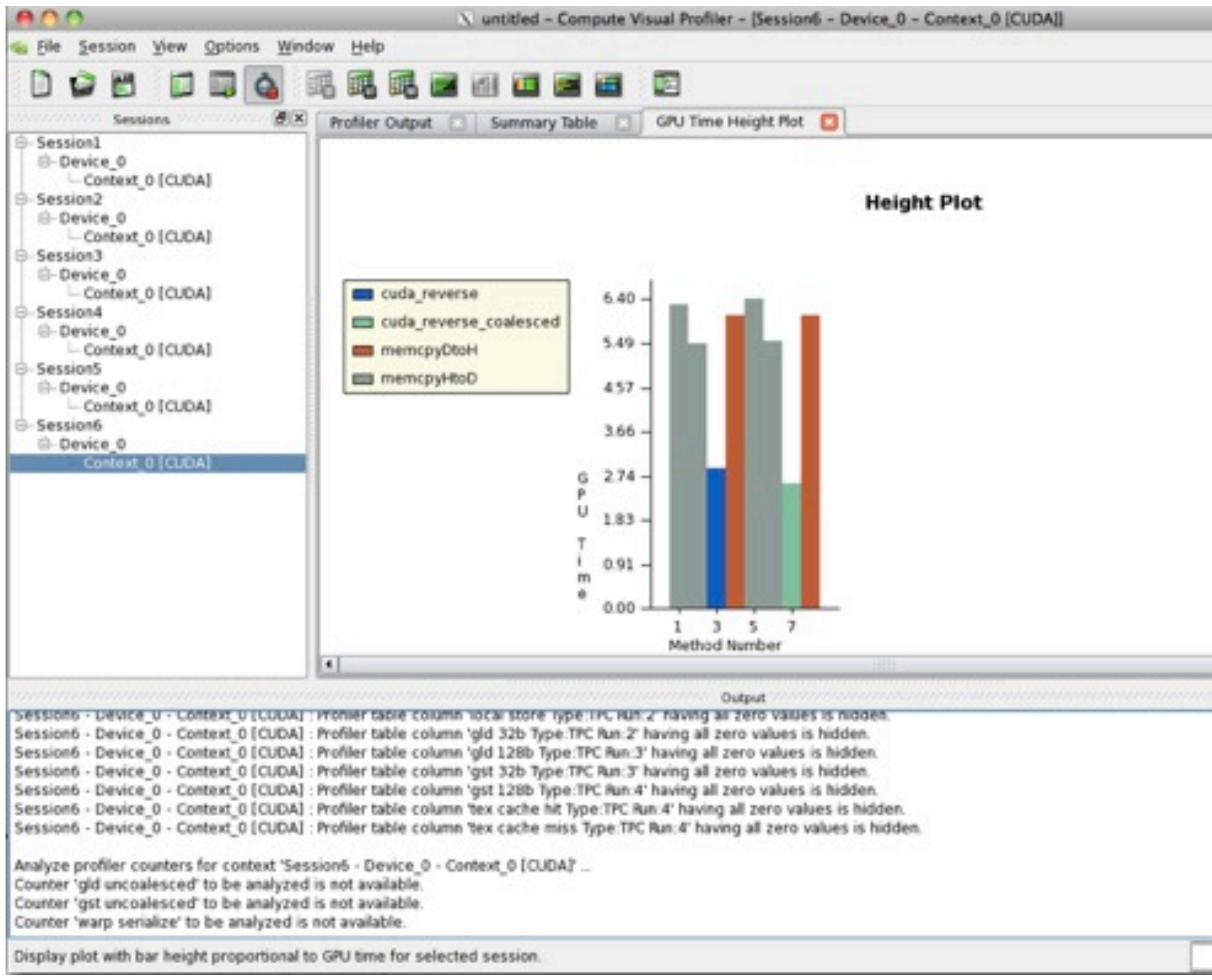
```
__global__ void cuda_reverse_coalesced(const float *xd,
                                             float *yd,
                                       const int n) {

    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;
}
```

Do permutation in **shared** memory

```
[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.
```

Visual Profiler

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

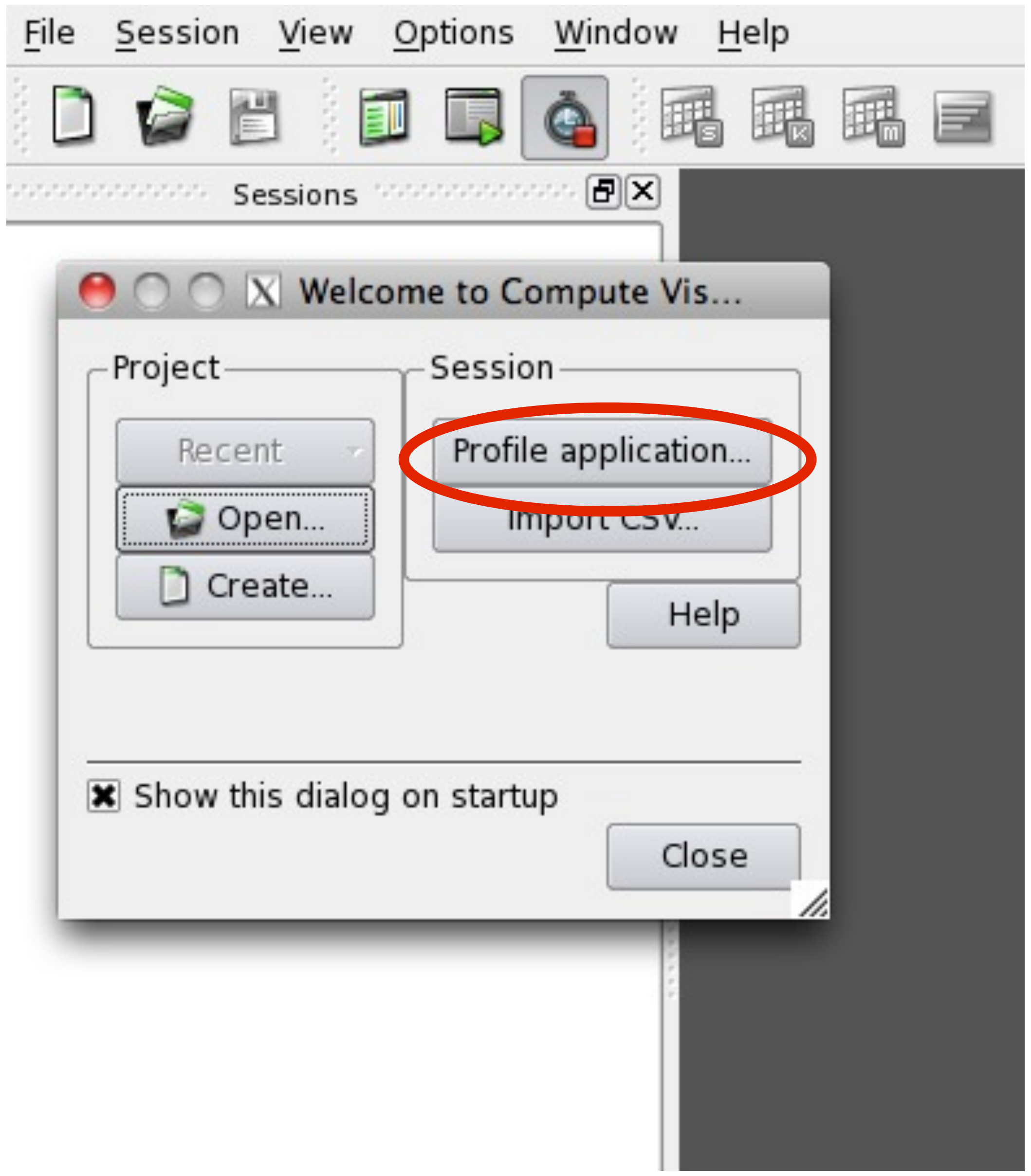


```
Session6 - Device_0 - Context_0 [CUDA] : Profiler table column 'local store Type:TPC Run:2' having all zero values is hidden.
Session6 - Device_0 - Context_0 [CUDA] : Profiler table column 'gld 32b Type:TPC Run:2' having all zero values is hidden.
Session6 - Device_0 - Context_0 [CUDA] : Profiler table column 'gld 128b Type:TPC Run:3' having all zero values is hidden.
Session6 - Device_0 - Context_0 [CUDA] : Profiler table column 'gst 32b Type:TPC Run:3' having all zero values is hidden.
Session6 - Device_0 - Context_0 [CUDA] : Profiler table column 'gst 128b Type:TPC Run:4' having all zero values is hidden.
Session6 - Device_0 - Context_0 [CUDA] : Profiler table column 'tex cache hit Type:TPC Run:4' having all zero values is hidden.
Session6 - Device_0 - Context_0 [CUDA] : Profiler table column 'tex cache miss Type:TPC Run:4' having all zero values is hidden.

Analyze profiler counters for context 'Session6 - Device_0 - Context_0 [CUDA]' ...
Counter 'gld uncoalesced' to be analyzed is not available.
Counter 'gst uncoalesced' to be analyzed is not available.
Counter 'warp serialize' to be analyzed is not available.
```

Display plot with bar height proportional to GPU time for selected session.

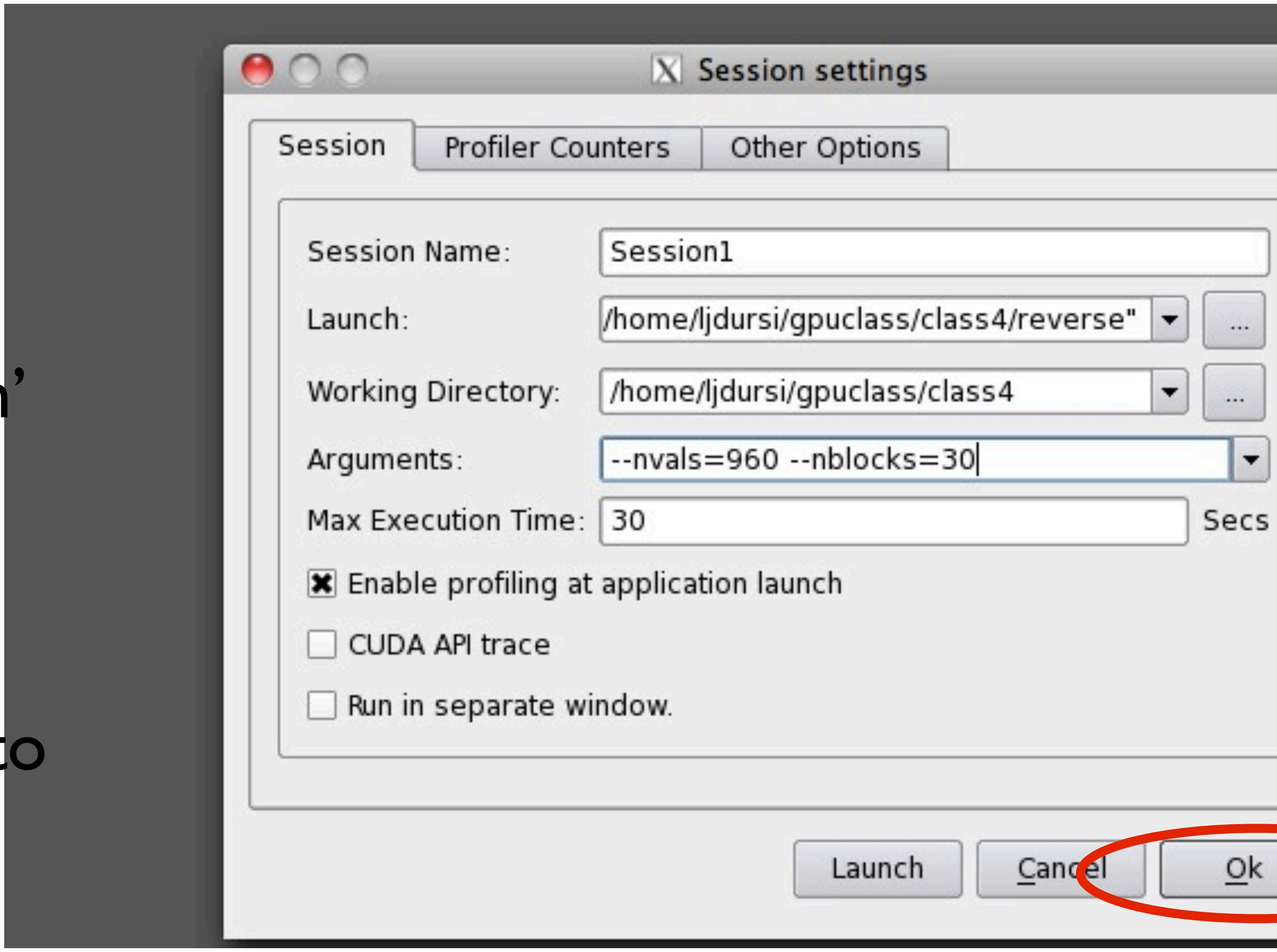
Visual Profiler

- 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,

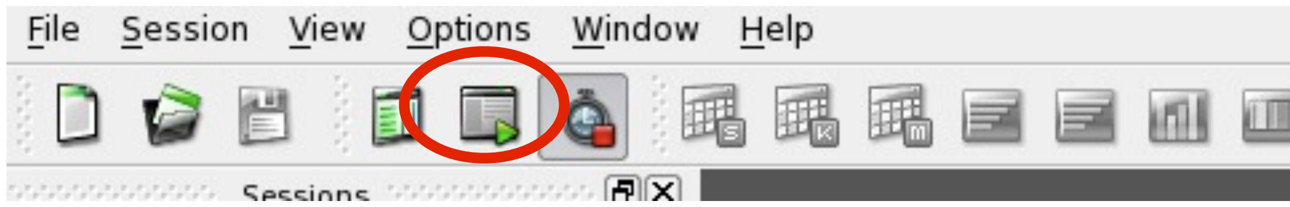


Session Name: Session1
Launch: /home/ljdursi/gpuclass/class4/reverse"
Working Directory: /home/ljdursi/gpuclass/class4
Arguments: --nvals=960 --nblocks=30
Max Execution Time: 30

[✓] Enable profiling at application launch
[ ] CUDA API trace
[ ] Run in separate window.

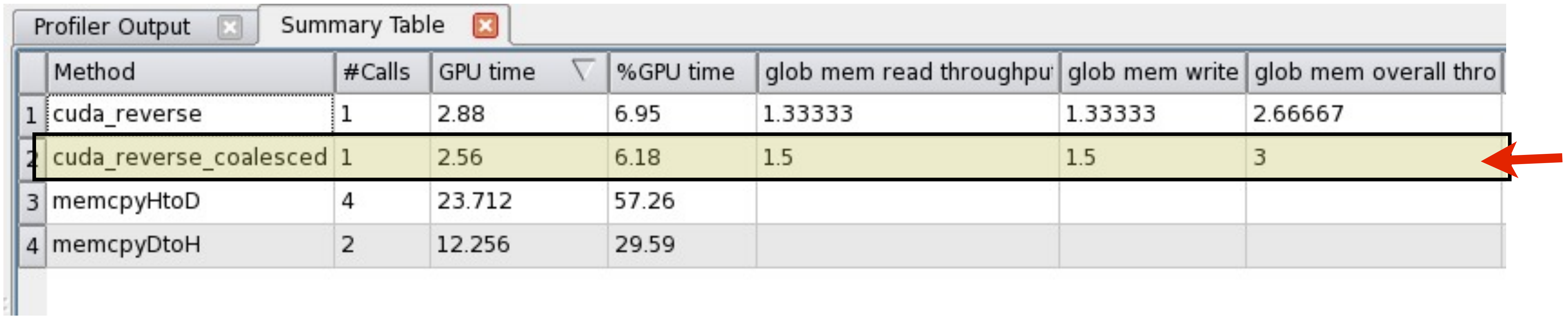
Visual Profiler

- 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.

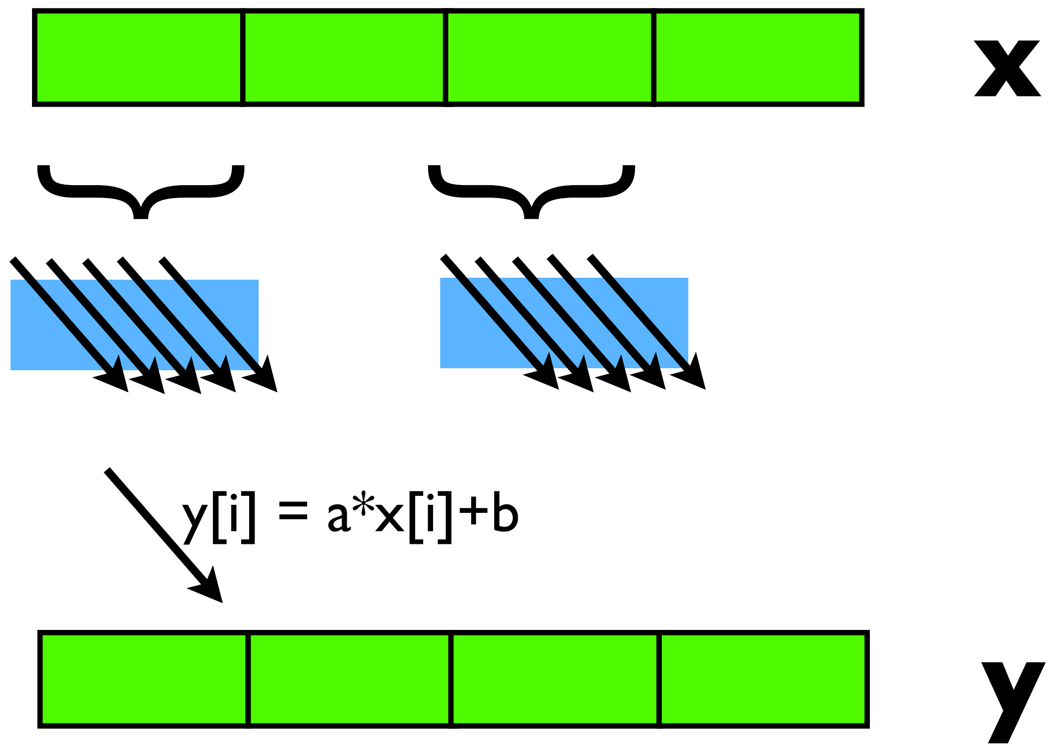


Profiler Output | Summary Table

	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			

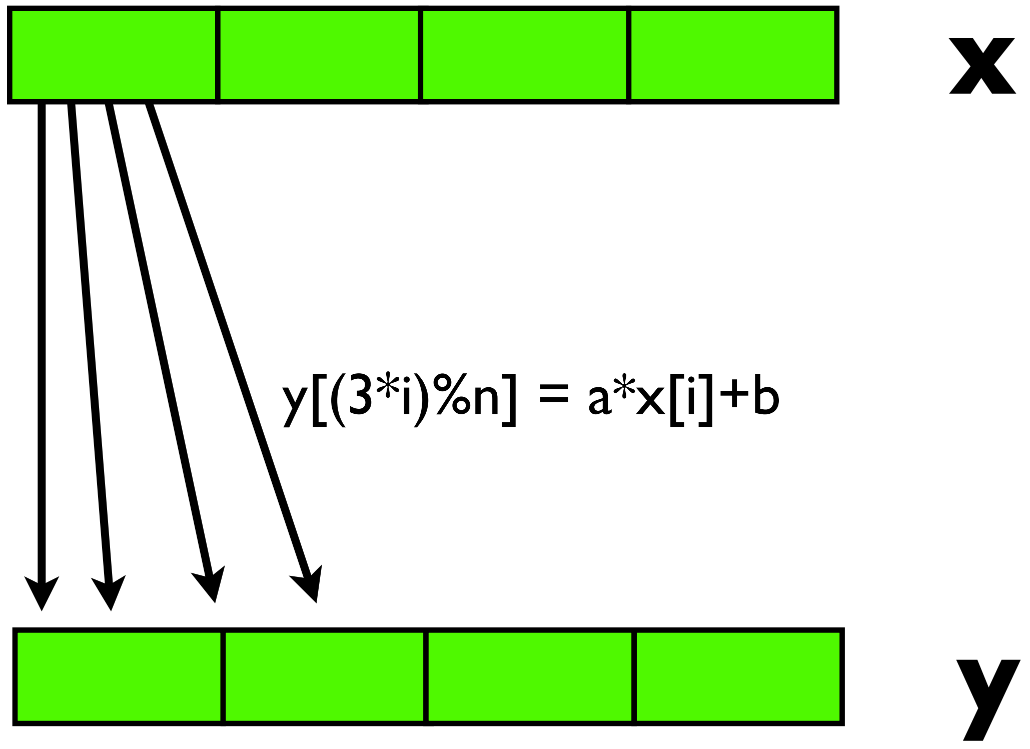
Another Example: Multi-block y=ax+b

- 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]



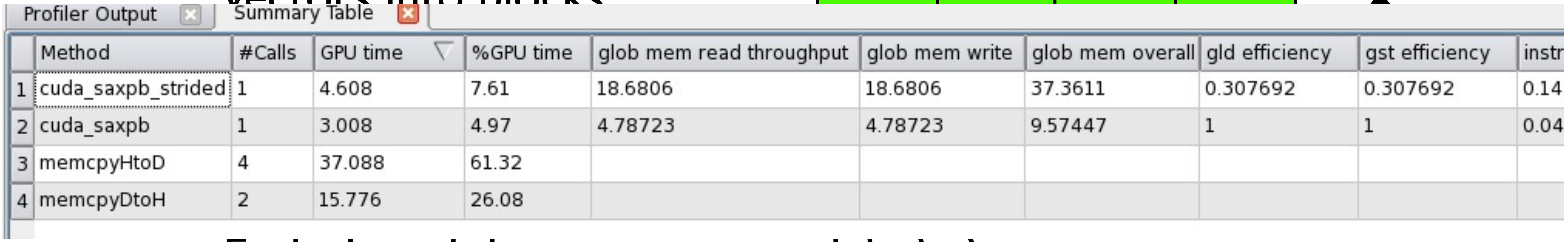
Another Example: Multi-block y=ax+b

- 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.



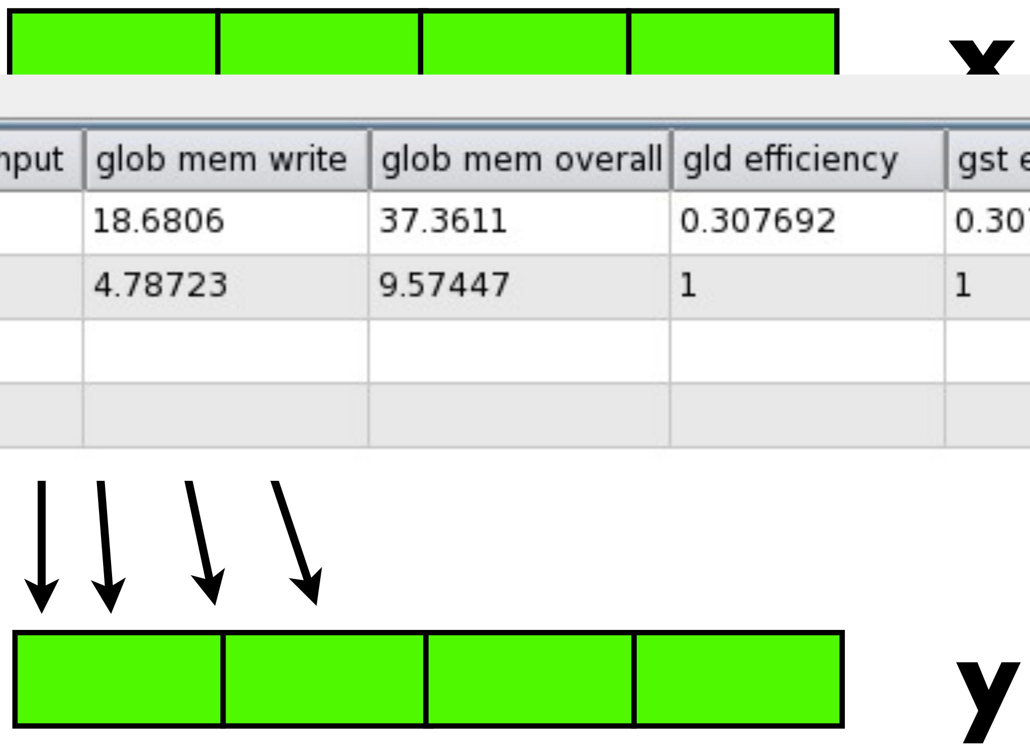
Another Example: Multi-block y=ax+b

- Break input, output vectors into blocks



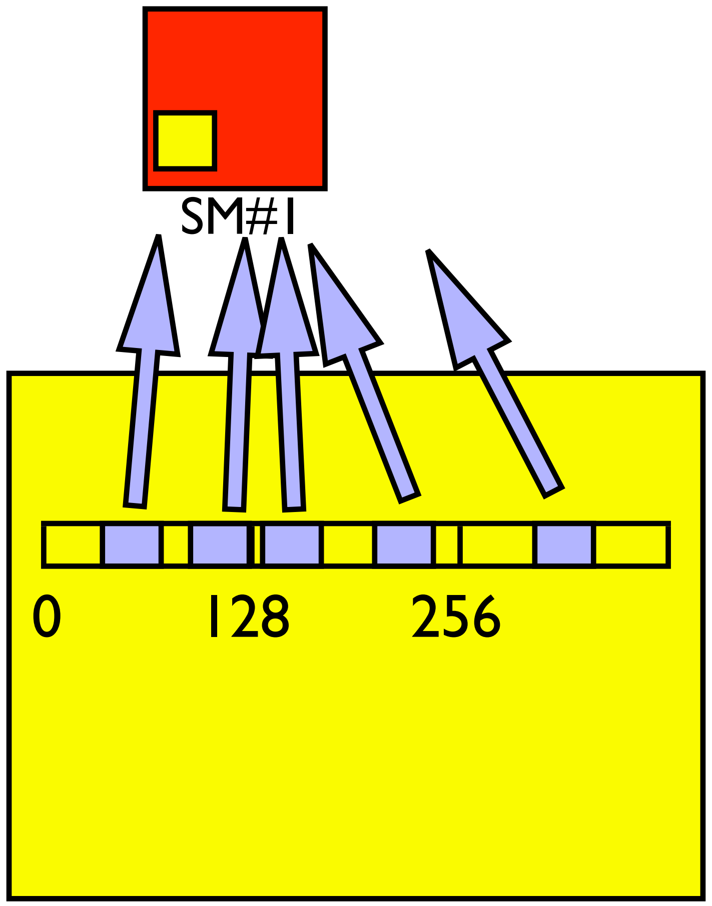
	Method	#Calls	GPU time	%GPU time	glob mem read throughput	glob mem write	glob mem overall	gld efficiency	gst efficiency	instr
1	cuda_saxpb_strided	1	4.608	7.61	18.6806	18.6806	37.3611	0.307692	0.307692	0.14
2	cuda_saxpb	1	3.008	4.97	4.78723	4.78723	9.57447	1	1	0.04
3	memcpyHtoD	4	37.088	61.32						
4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



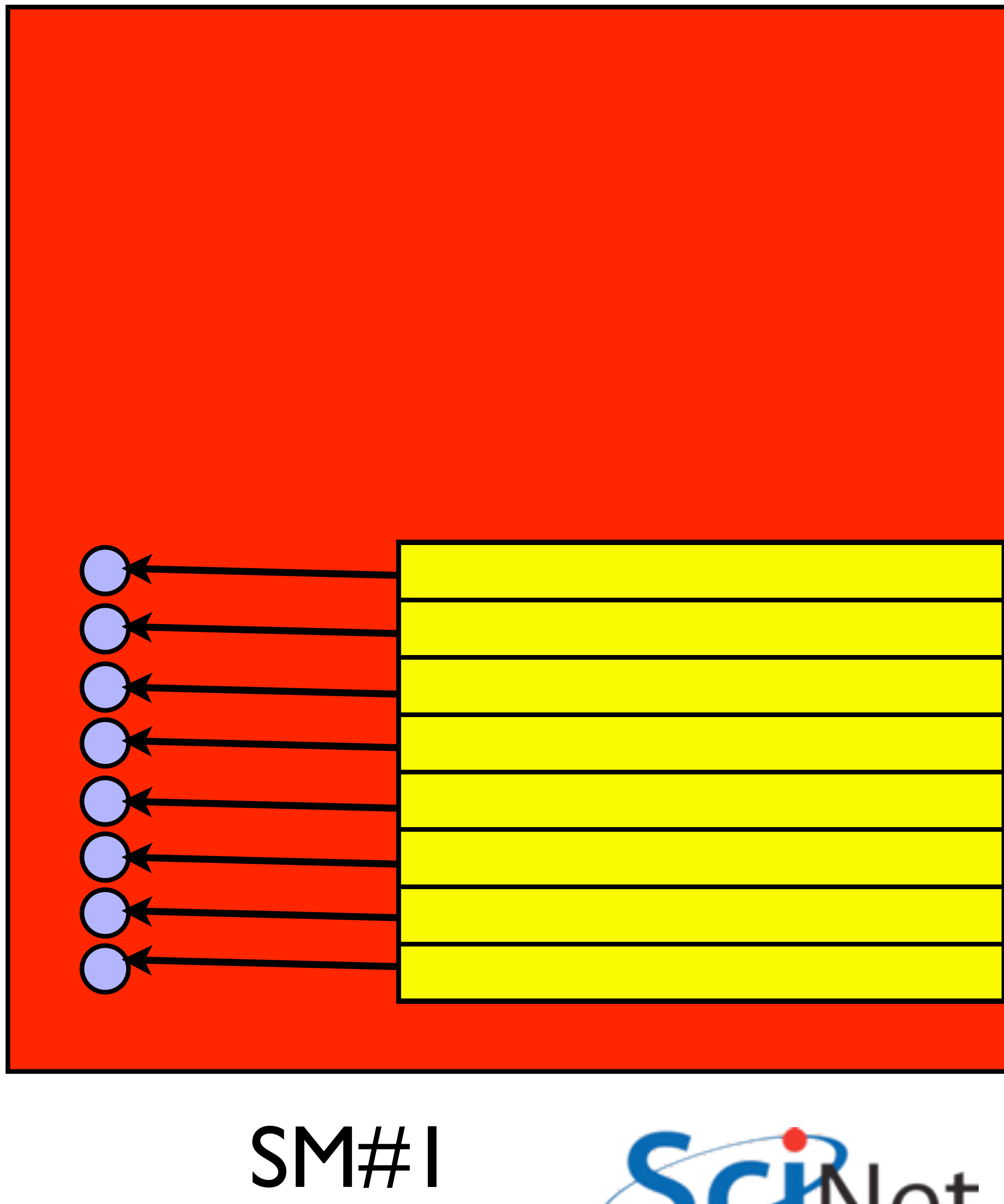
Coalesced Memory Access

- 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.



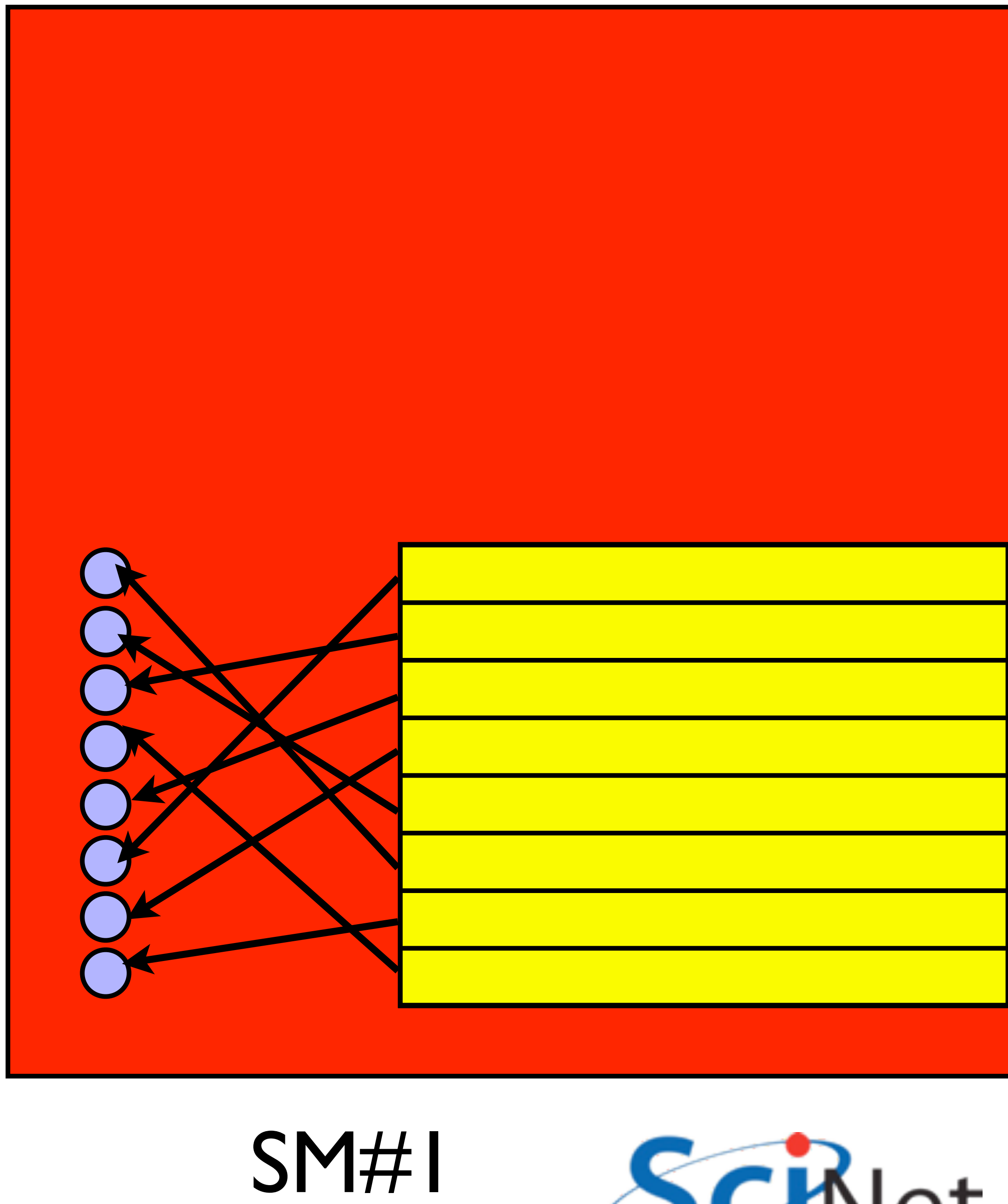
Shared Memory Bank Conflicts

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



Shared Memory Bank Conflicts

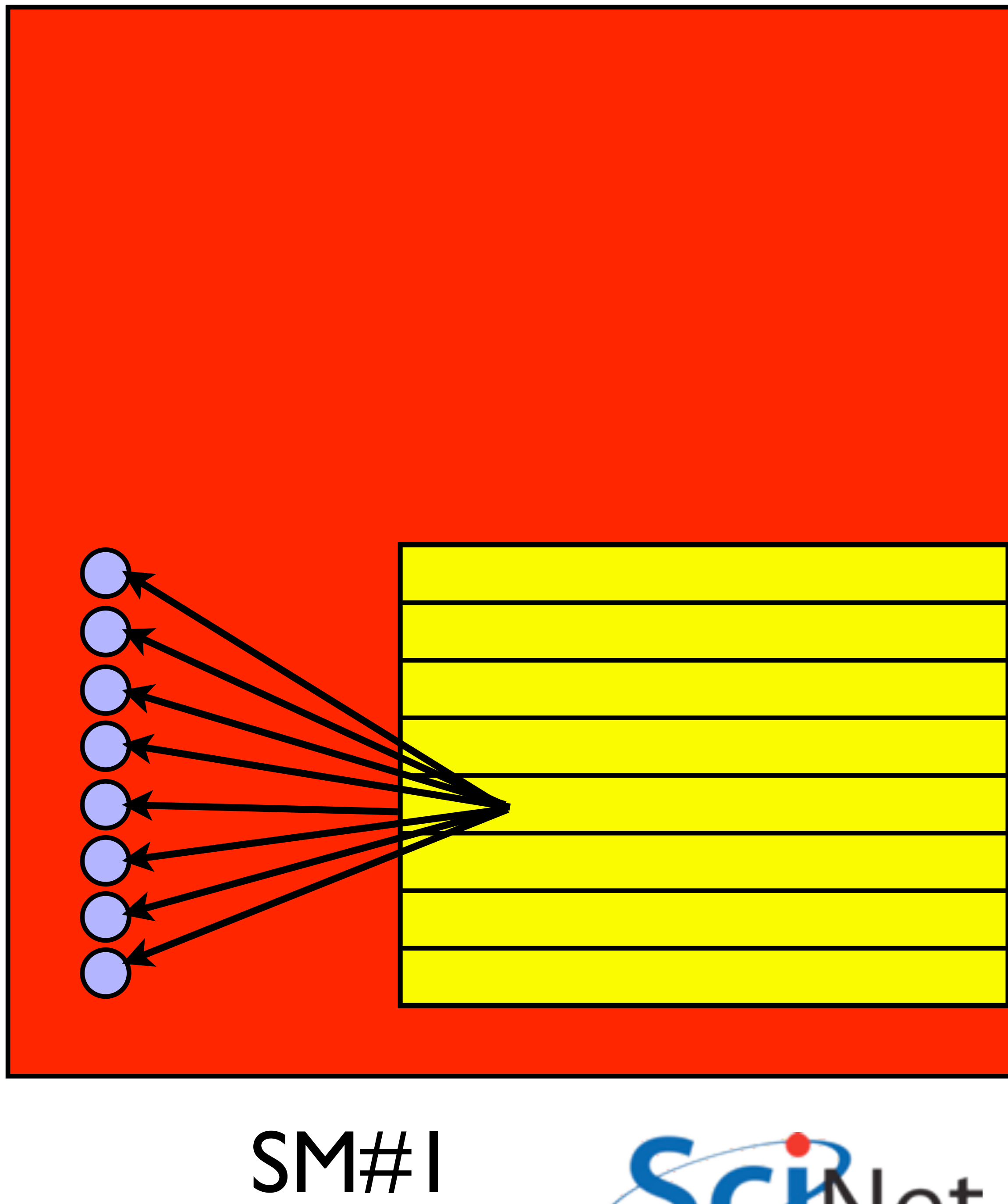
- Each thread in warp accesses different bank: no problem.



SciNet

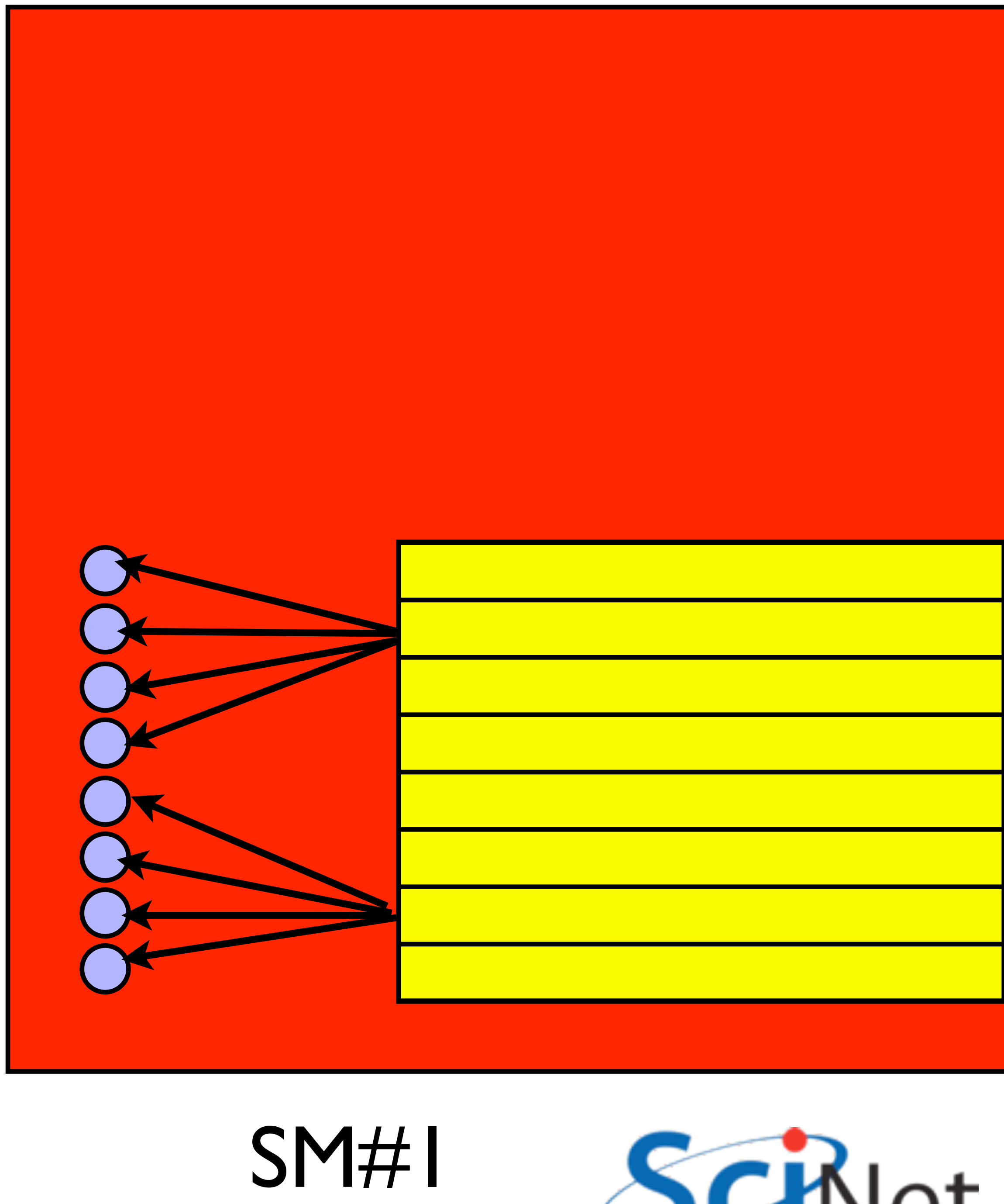
Shared Memory Bank Conflicts

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



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.



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

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

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

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

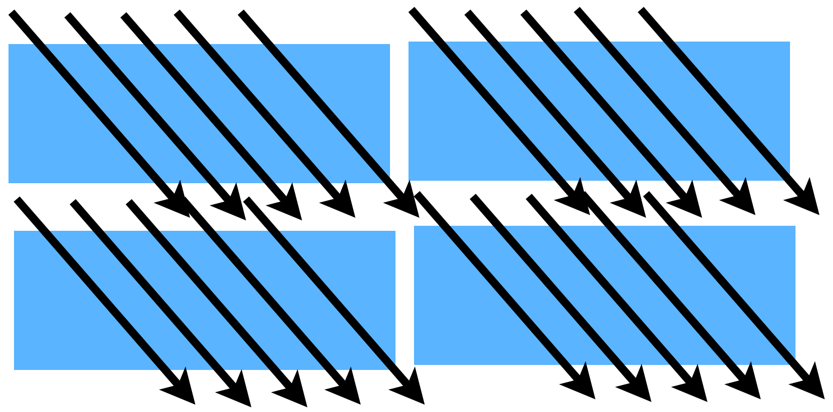
Shared Memory Bank Conflicts

- 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

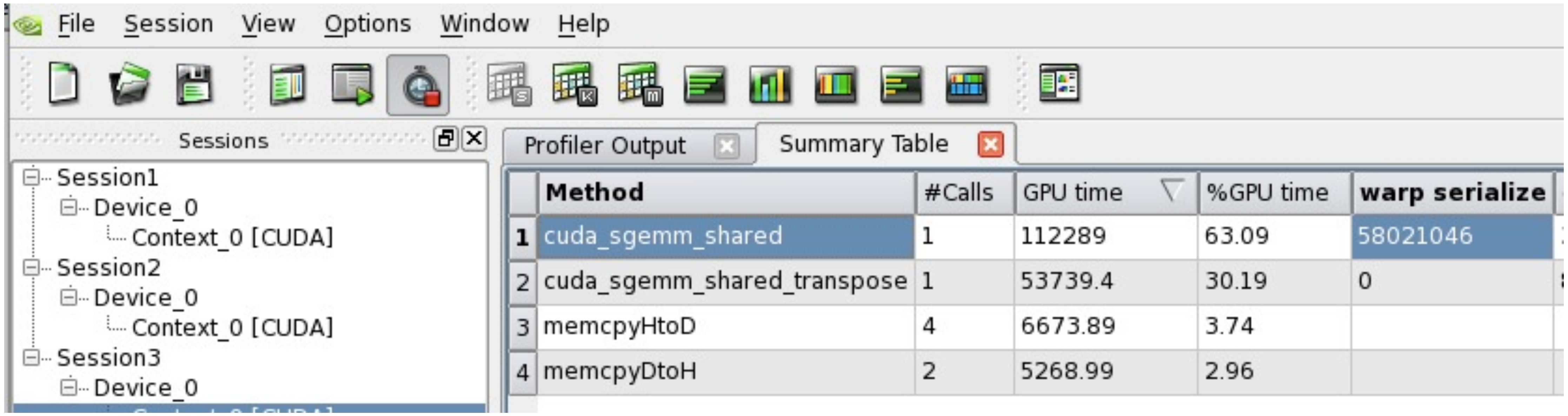
- Easy to see how warps are assigned in 1-d block:
  - First 32 = warp0
  - Next 32 = warp1..
- How done in 2d block?
- C ordering: x first, then y
- blockDim.x = 32:
  - warp 0 : blockDim.y = 0
  - warp 1: blockDim.y = 1..



```
__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



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

	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

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.
```

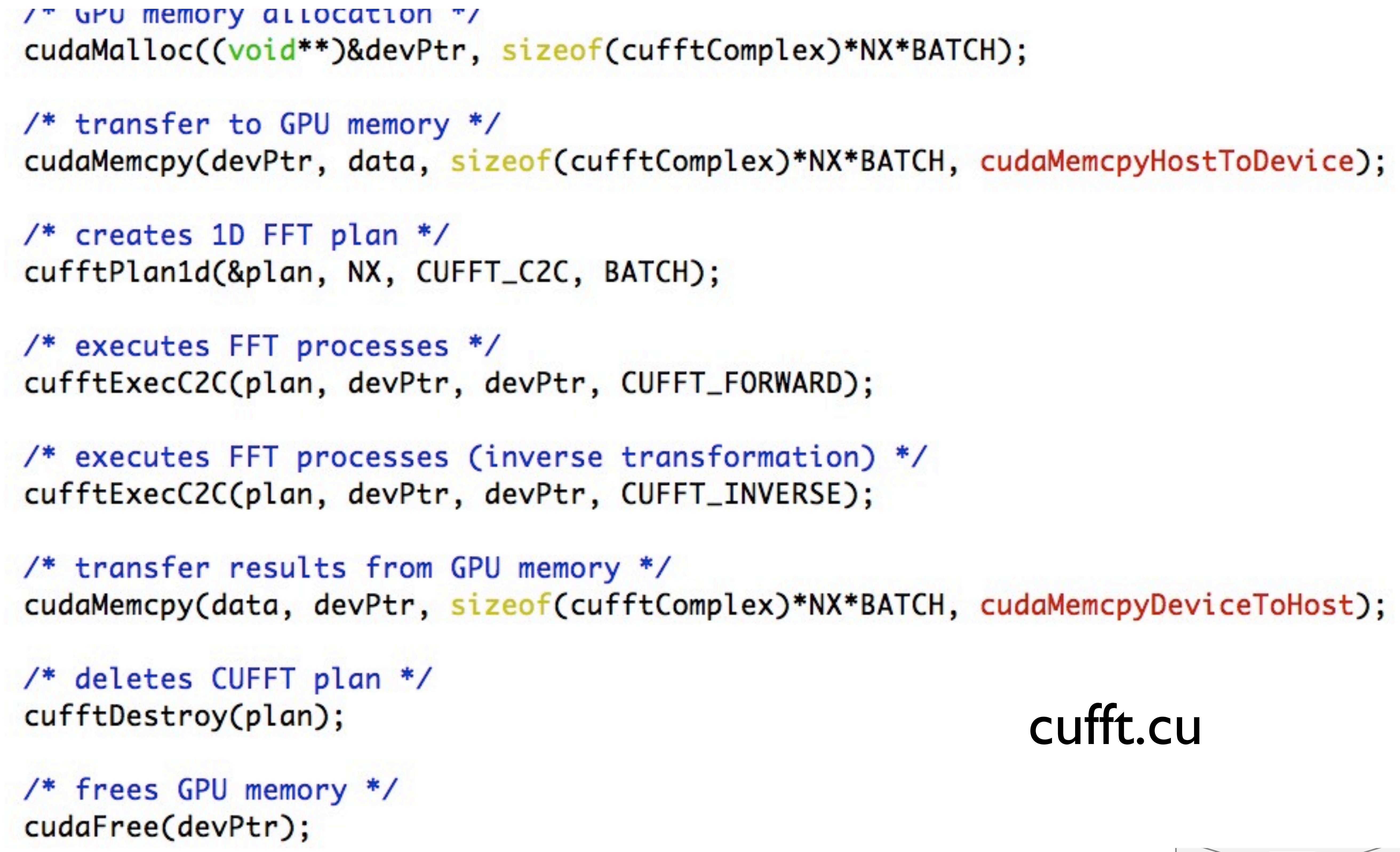
- Use tuned code where available (this is still much slower than CUBLAS, MAGMA!)

CUBLAS

```
cublasInit();
CHK_CUBLAS( cublasAlloc(n*n, sizeof(float), (void**)&ad) );
cublasAlloc(n*n, sizeof(float), (void**)&bd);
cublasAlloc(n*n, sizeof(float), (void**)&cd);

tick(&gputimer);

CHK_CUBLAS( cublasSetMatrix(n, n, sizeof(float),
                            a, n, ad, n) );
CHK_CUBLAS( cublasSetMatrix(n, n, sizeof(float),
                            b, n, bd, n) );
cublasSgemm ('n', 'n', n, n, n, 1.0, ad, n, bd, n, 0.0, cd, n);
CHK_CUBLAS( cublasGetError() );

CHK_CUBLAS( cublasGetMatrix (n, n, sizeof(float),
                             cd, n, ccuda, n) );
gputime = tock(&gputimer);

CHK_CUBLAS( cublasFree( ad ) );
CHK_CUBLAS( cublasFree( bd ) );
CHK_CUBLAS( cublasFree( cd ) );
cublasShutdown();
```

cublas.cu

CUFFT



```
/* GPU memory allocation */
cudaMalloc((void**)&devPtr, sizeof(cufftComplex)*NX*BATCH);

/* transfer to GPU memory */
cudaMemcpy(devPtr, data, sizeof(cufftComplex)*NX*BATCH, cudaMemcpyHostToDevice);

/* creates 1D FFT plan */
cufftPlan1d(&plan, NX, CUFFT_C2C, BATCH);

/* executes FFT processes */
cufftExecC2C(plan, devPtr, devPtr, CUFFT_FORWARD);

/* executes FFT processes (inverse transformation) */
cufftExecC2C(plan, devPtr, devPtr, CUFFT_INVERSE);

/* transfer results from GPU memory */
cudaMemcpy(data, devPtr, sizeof(cufftComplex)*NX*BATCH, cudaMemcpyDeviceToHost);

/* deletes CUFFT plan */
cufftDestroy(plan);

/* frees GPU memory */
cudaFree(devPtr);
```

cufft.cu