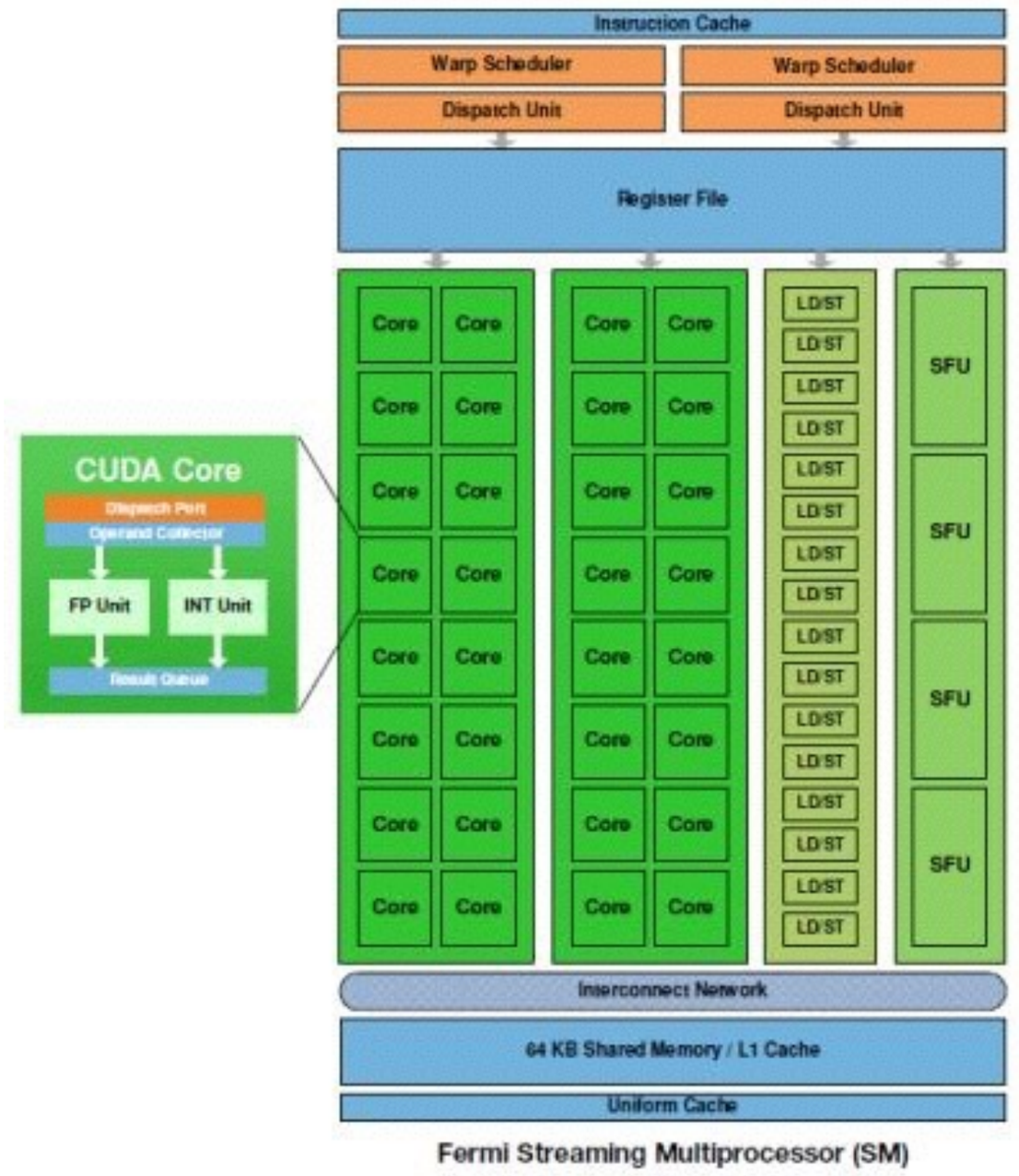
Sixth I.N.F.N. International School on
"Architectures, tools and methodologies for developing efficient large scale scientific computing applications"
Font size Bigger Reset Smaller
ESC14
Ce.U.B. - Bertinoro (FC) 19 - 25 October 2014

CUDA and GPU Performance Tuning Fundamentals: A hands-on introduction

Francesco Rossi
University of Bologna and INFN

A Streaming Multiprocessor (SM) is a parallel compute unit that dispatches instructions to *warps*, vectors of 32 lightweight *threads* (SIMT lanes).

- GPUs have ~16 SMs.
- Warps are capable of SIMT (Single Instruction, Multiple Thread) execution:
  - Virtually, each thread has its own execution context (registers, program counters, …).
  - But serialization occurs if threads take different execution paths.
  - SIMT~ SPMD (single program multiple data) instruction set, vectorized/dispatched by hardware to SIMD vector lanes (single instruction multiple data)*.



Fermi Streaming Multiprocessor (SM)

* Using this terminology since you've already heard of SIMD and SPMD at this school

<CUDA Program: SPMD Instructions>

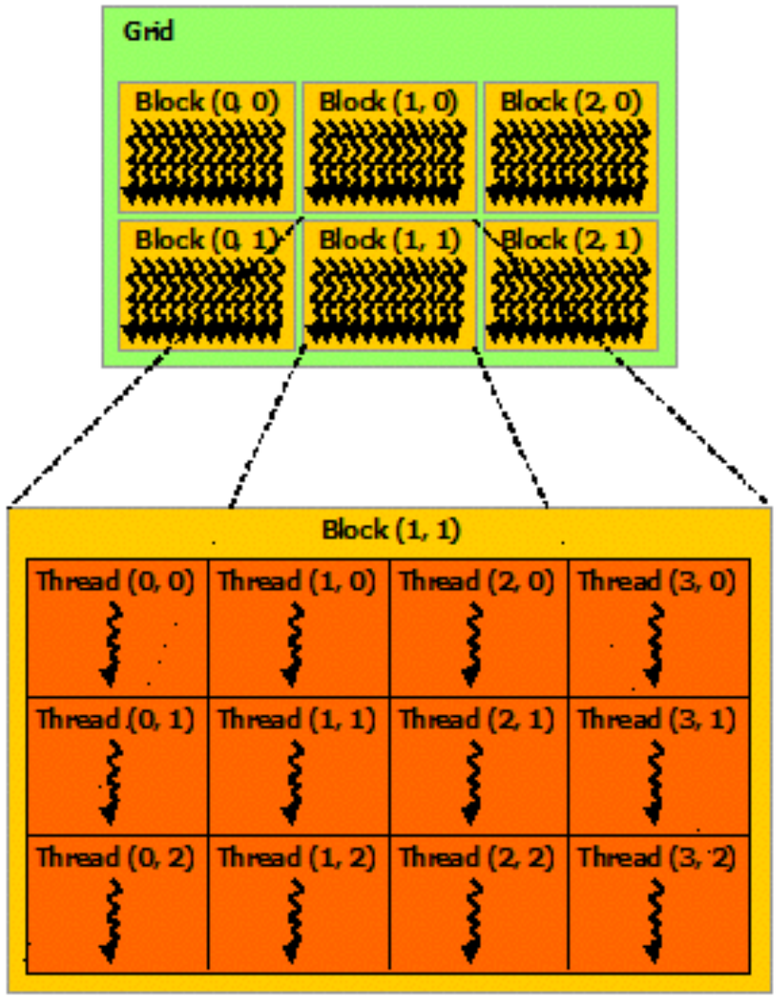
Dispatcher that vectorizes

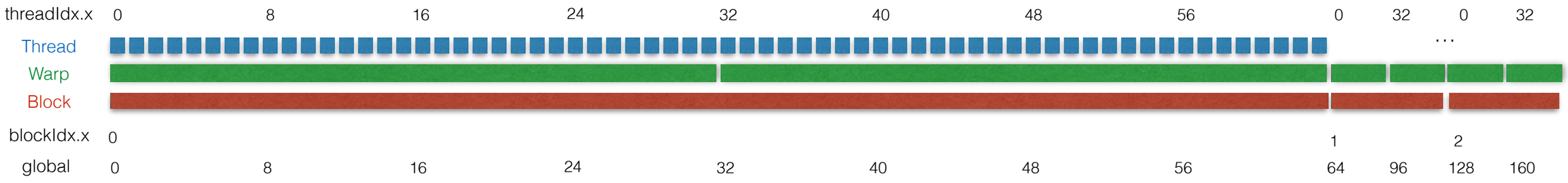
SIMD Hardware

Fermi Streaming Multiprocessor (SM)

The CUDA programming model is an abstraction of the multiprocessor.

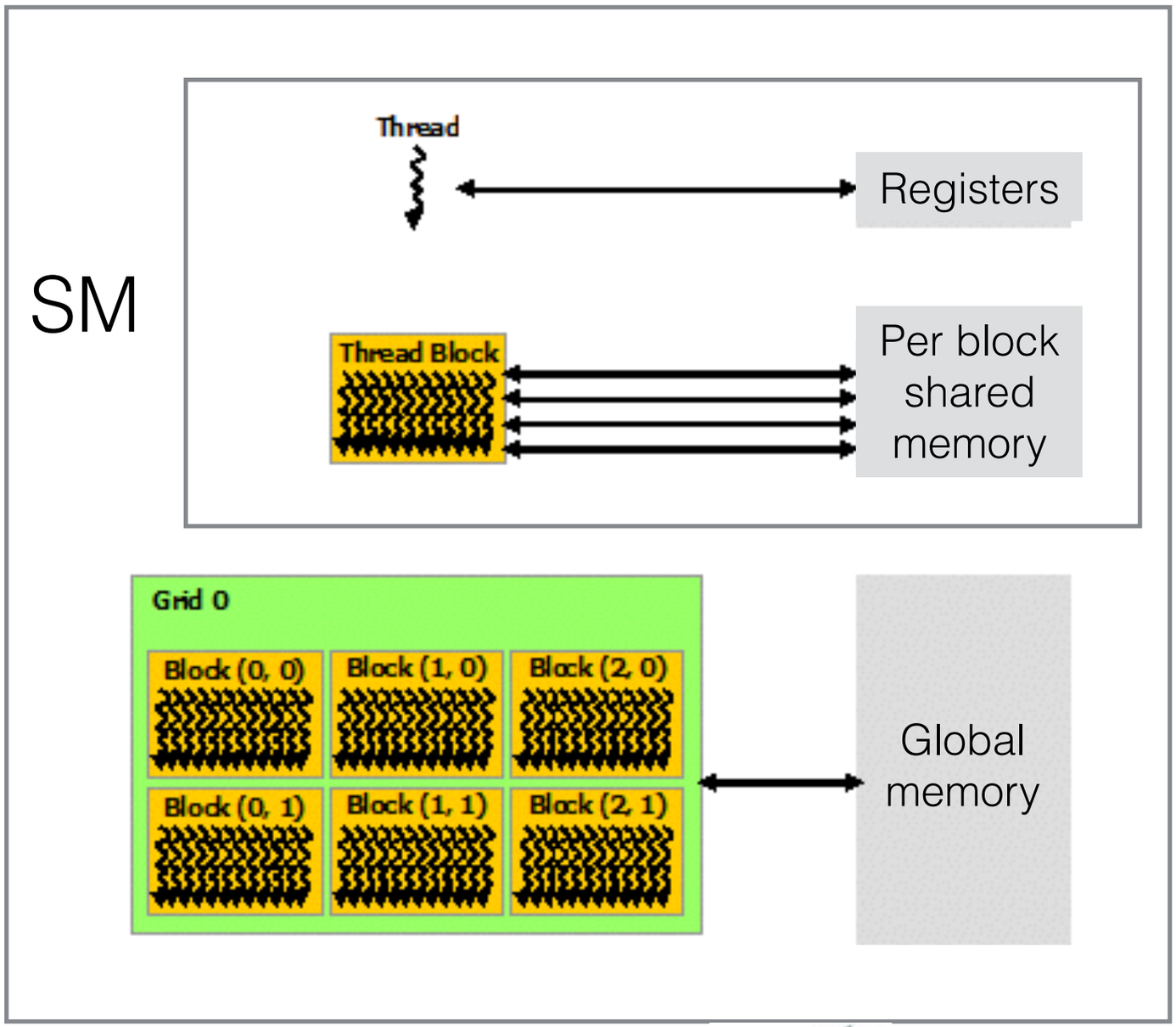
- A *block* is a group of a variable number of threads (warps).
- A streaming multiprocessor is assigned a few blocks.
- Blocks are arranged on a *grid*.
- Each thread is identified inside its block by the integer triplet *threadIdx.{x,y,z}* and each block with *blockIdx.{x,y,z}.*
- Execution of warps (blocks) happens in parallel and no order is specified, as they are dynamically dispatched.

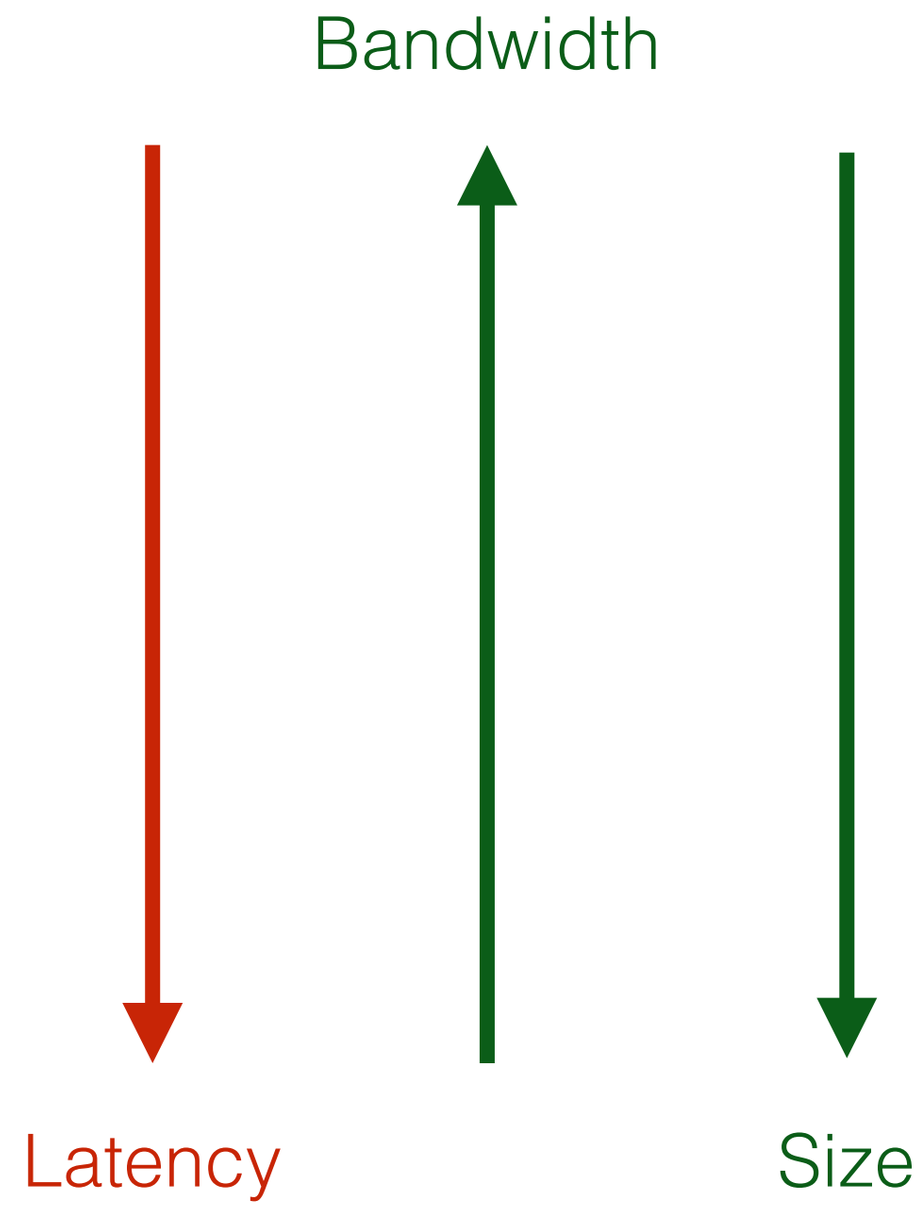




(BlockDim.x=64, 1D grid)

The memory hierarchy reflects the parallelism model hierarchy.





OpenCL vs. CUDA Terminology

Third party names are the property of their owners.

- Host defines a **command queue** and associates it with a context (devices, kernels, memory, etc).
- Host enqueues commands to the command queue

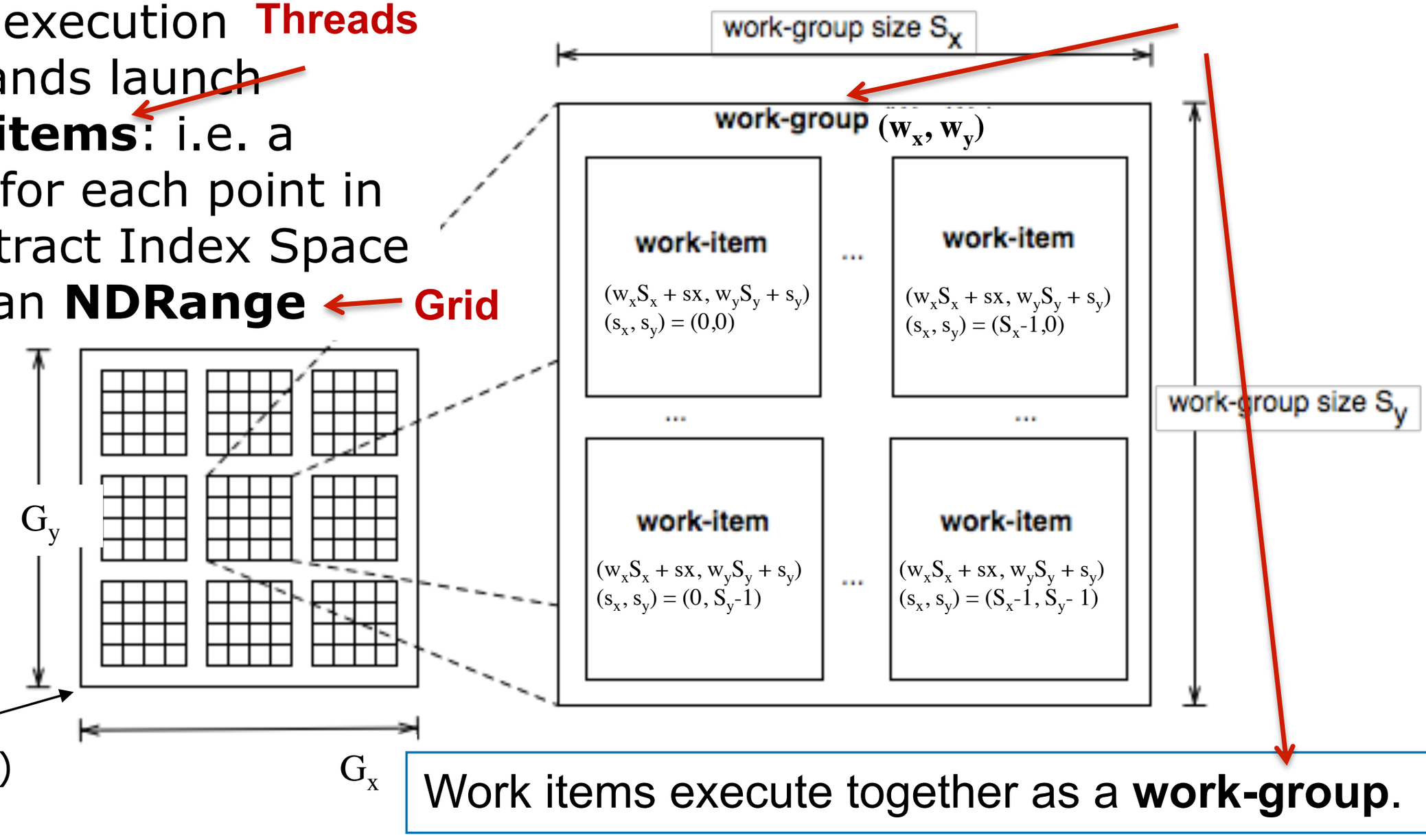
CUDA Stream

Thread Block

Threads

Kernel execution commands launch **work-items**: i.e. a kernel for each point in an abstract Index Space called an **NDRange**

Grid



Work items execute together as a **work-group**.

Thanks to Tim Mattson @ Intel for this slide

```
// Compute sum of length-N vectors: C = A + B
void
vecAdd (float* a, float* b, float* c, int N) {
    for (int i = 0; i < N; i++)
        c[i] = a[i] + b[i];
}

int main () {
    int N = ... ;
    float *d_a, *d_b, *d_c;
    d_a = new float[N];
  // ... allocate other arrays, fill with data


    vecAdd (d_a, d_b, d_c, N);
}
```

Serial Code

OpenCL

```
__kernel void vadd(__global float* a,__global float* b,
    __global float* c,const unsigned int count)
{
   int i = get_global_id(0);
   if(i < count)  {
       c[i] = a[i] + b[i];
   }
}
...
d_a   = cl::Buffer(context, begin(h_a), end(h_a), true);
d_b   = cl::Buffer(context, begin(h_b), end(h_b), true);
d_c  = cl::Buffer(context, CL_MEM_WRITE_ONLY, sizeof(float) * LENGTH);

vadd(cl::EnqueueArgs( queue, cl::NDRange(count)),
    d_a, d_b, d_c, count);

queue.finish();

cl::copy(queue, d_c, begin(h_c), end(h_c));
...
```

CUDA

```
void __global__
vecAdd (float* a, float* b, float* c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) c[i] = a[i] + b[i];
}

int main () {
    int N = ... ;
    float *d_a, *d_b, *d_c;
    // ... allocate host  arrays h_a,h_b,h_c, fill with data
    cudaMalloc (&d_a,  sizeof(float) * N);
    cudaMalloc (&d_b,  sizeof(float) * N);
    cudaMalloc (&d_c,  sizeof(float) * N);
    cudaMemcpy(d_a,h_a,sizeof(float) * N,cudaMemcpyHostToDevice);

    // Use thread blocks with 256 threads each
    vecAdd <<< (N+255)/256, 256 >>> (d_a, d_b, d_c, N);
    cudaMemcpy(d_a,h_a,sizeof(float) * N,cudaMemcpyDeviceToHost);
}
```

Three performance questions.

Question: why does this kernel does not achieve the GPU bandwidth achieved by cudaMemcpy if stride > 1?

```
template<int stride>
__global__ void strideCopy(float *odata, float* idata, int N)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    while(i < N)
    {
        int xid = (i)*stride;
        odata[xid] = idata[xid];

        i += gridDim.x*blockDim.x;
    }
}
```

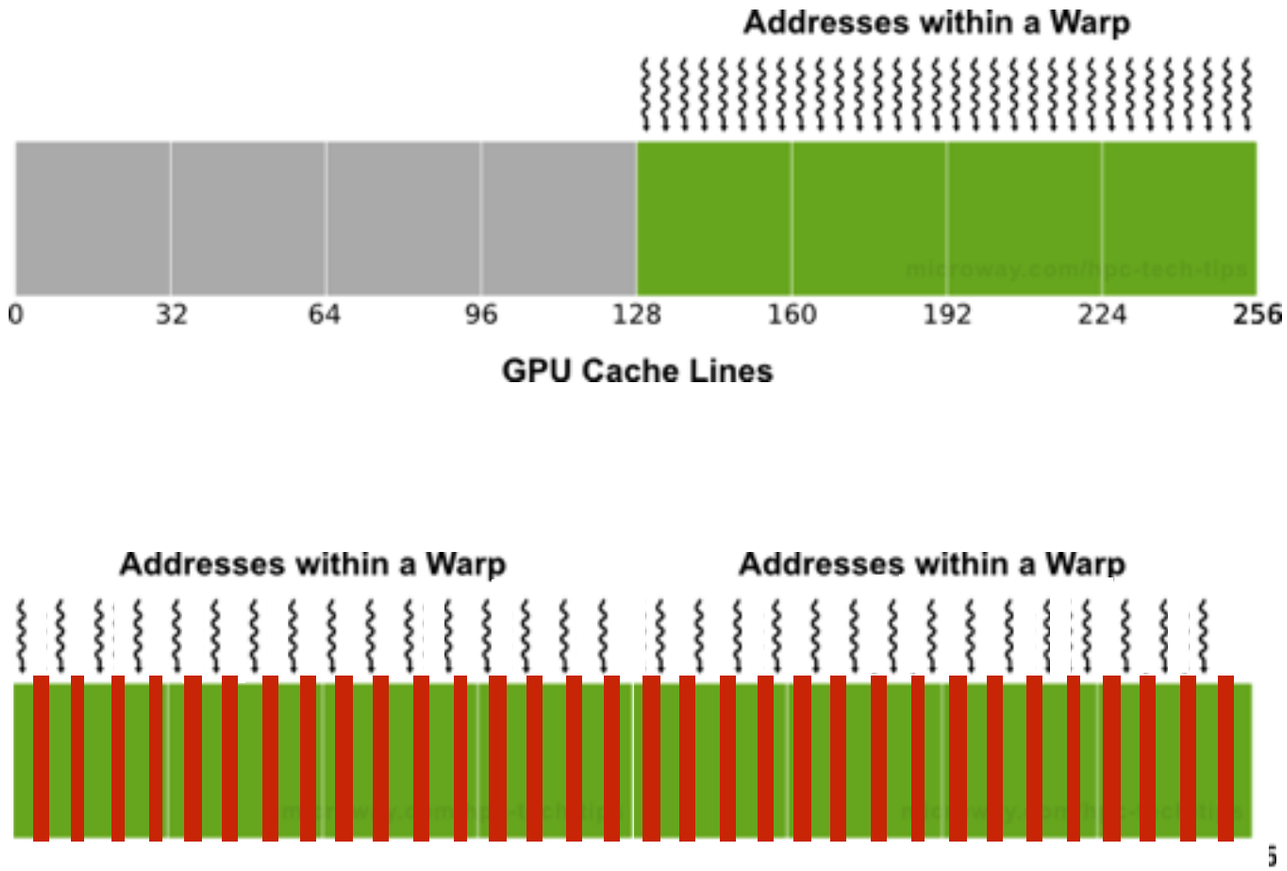
Hands on: Compile and run bw.cu

A: Warps should access global memory in a coalesced manner.

- For example, on Kepler only contiguous chunks of global memory (in caching mode, cache lines) can be moved from/to the warps.

- A stride of two (four) would require two (four) lines of cache to be moved.

- See this presentation for more details http://on-demand.gputechconf.com/gtc-express/2011/presentations/cuda_webinars_GlobalMemory.pdf



Question: why does the second kernel achieves roughly ~50% of the floating point operation throughput compared to the first?

- Compile and run diverge.cu and fix the performance bug.

- Exercise: fix the performance issue of the second kernel while keeping the same result.

- On Fermi, you should be able to see the floating point throughput fully utilized:

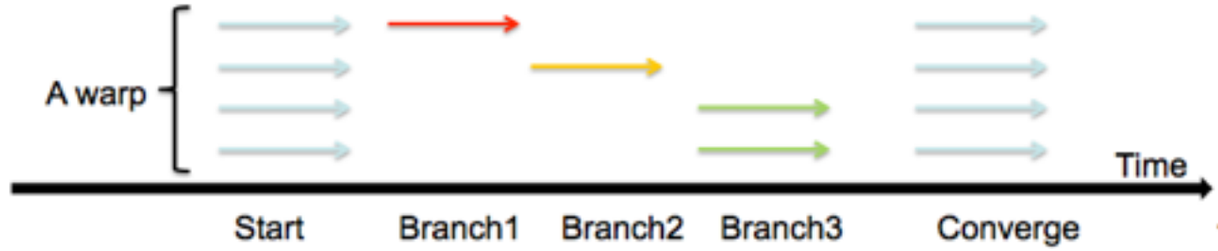
```
$ ./divergence
divergence: time=1.0965s throughput=501.34 GFLOP/s
no divergence:time=0.5513s throughput=997.16 GFLOP/s
```

```
template<int ITERATIONS>
__global__ void a_lot_of_operations(float *odata, int N)
{
    int tid = blockIdx.x*blockDim.x + threadIdx.x;

    while(tid < N)
    {
        float x = tid*1.f/N;


        #pragma unroll 256
        for(int i=0; i<ITERATIONS; i++)
        {
            x = x*0.9999f+1e-8f;
        }

        odata[tid] = x;
        tid += blockDim.x*gridDim.x;
    }
}

template<int ITERATIONS>
__global__ void big_problem_of_divergence(float *odata, int N)
{
    int tid = blockIdx.x*blockDim.x + threadIdx.x;

    while(tid < N)
    {
        float x = tid*1.f/N;;
        if(tid % 2 == 0)
        {
            #pragma unroll 256
            for(int i=0; i<ITERATIONS; i++)
            {
                x = x*0.9999f+1e-8f;
            }
        }
        else
        {
            #pragma unroll 256
            for(int i=0; i<ITERATIONS; i++)
            {
                x = x*1.0001f+1e-8f;
            }
        }

        odata[tid] = x;
        tid += blockDim.x*gridDim.x;
    }
}
```

A: If threads in a warp take different code paths then the execution paths are serialized.

- A warp (32 threads) can be seen as a large, flexible SIMD vector unit, programmable to take internal branches (virtually, a SPMD) but with performance cost (serialization).

- A single instruction can be dispatched only to an entire warp.




```cpp
template<int ITERATIONS>
__global__ void big_problem_of_divergence_solution(float *odata, int N, float p)
{
    int tid = blockIdx.x*blockDim.x + threadIdx.x;

    while(tid < N)
    {
        float x = tid*1.f/N*p;
        float a[2] = {0.9999f, 1.0001f};//easy solution

        #pragma unroll 256
        for(int i=0; i<ITERATIONS; i++)
        {
            x = x*a[tid%2]+1e-8f; //use so called local memory
        }

        odata[tid] = x;

        tid += blockDim.x*gridDim.x;
    }
}
```

- <- Lazy and generally bad solution. Homework: a more robust solution.

- Constants array is stored in the “local” memory space (see CUDA programming guide) and the load is *optimized away to happen just once* per thread, see ptx (nvcc -ptx diverge.cu)

```
. . .
 ld.local.f32  %f8, [%rd10+0];
. . .
```

Question: How can we use parallel caches to help this kernel loading *idata* just once per grid?

```
// A five point stencil. Why does it not achieve the theoretcal GPU bandwidth limit?
__global__ void laplacian(float *odata, float* idata, int N)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    while(i < N)
    {
        if(i > 1 && i < N-2)
        {
            odata[i] = -idata[i-2]+16.f*idata[i-1]-30.0f*idata[i]+16.f*idata[i+1]-idata[i+2];
        }

        i += gridDim.x*blockDim.x;
    }
}
```

The memory access pattern is fully coalesced, and the bandwidth is fully used, yet we can optimize the performance avoiding to load data twice using caches.

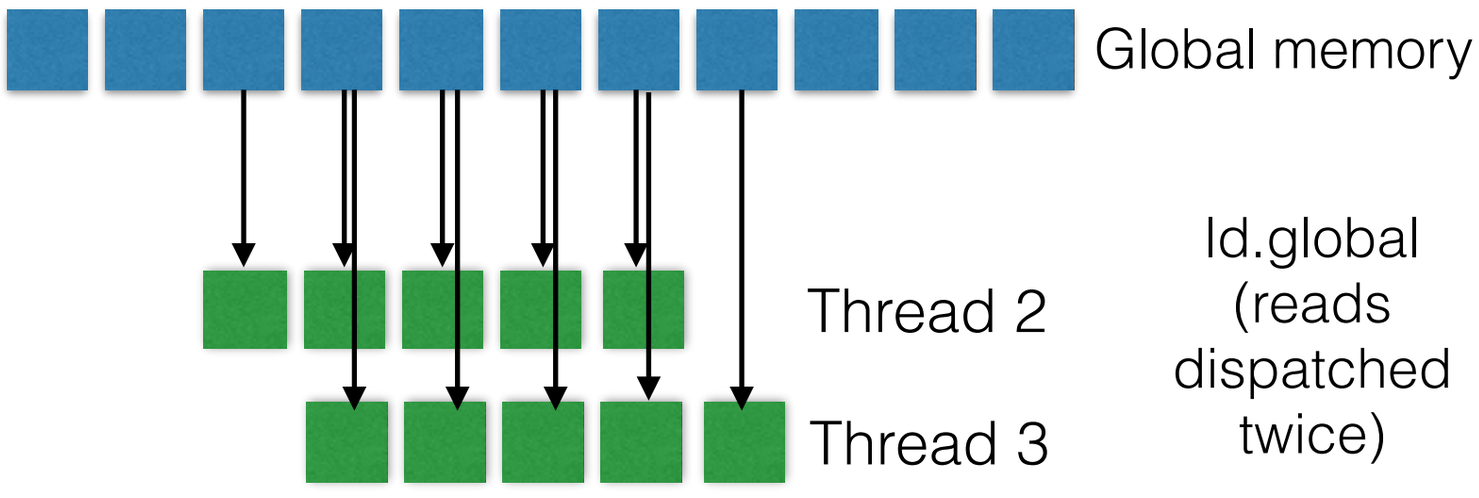
Compile and run stencil.cu

The shared memory cache can be manually programmed to share the different values read by threads.

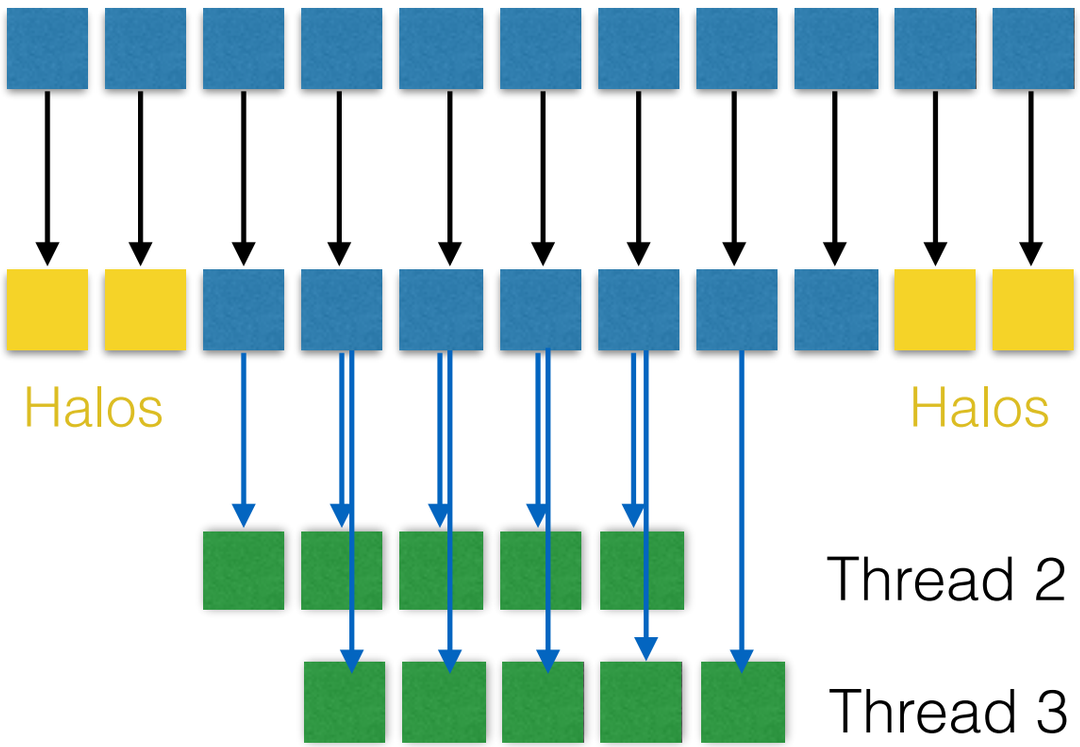
- Caches on CPUs are used to optimize the latencies for a single thread.
- Caches on the GPU are used to share data amongst parallel units.
- Shared memory is a programmable cache to share data amongst the threads in a block.

```
// Caching using shared memory
template<int BLOCK_SIZE>
__global__ void laplacian_shared(float * odata, float* idata, int N)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    __shared__ float mycache[BLOCK_SIZE+4];
    const int kernel_radius = 2;
    const int si = threadIdx.x+kernel_radius;

    while(i < N)
    {
        mycache[si] = idata[i];

        if(threadIdx.x < kernel_radius)
        {
            mycache[si-kernel_radius] = 0;
            mycache[si+BLOCK_SIZE] = 0;

            if(i-kernel_radius >= 0)
            {
                mycache[si-kernel_radius] = idata[i-kernel_radius];
            }
            if(i+BLOCK_SIZE < N)
            {
                mycache[si+BLOCK_SIZE] = idata[i+BLOCK_SIZE];
            }
        }
        __syncthreads();

        odata[i] = -mycache[si-2]+16.f*mycache[si-1]-
            30.0f*mycache[si]+16.f*mycache[si+1]-mycache[si+2];

        i += gridDim.x*blockDim.x;
    }
}
```

The shared memory cache can be manually programmed to share the different values read by threads.



ld.global
(reads
dispatched
twice)



ld.shared
(much faster)

```
mycache[si] = idata[i];

if(threadIdx.x < kernel_radius)
{
    mycache[si-kernel_radius] = 0;
    mycache[si+BLOCK_SIZE] = 0;

    if(i-kernel_radius >= 0)
    {
        mycache[si-kernel_radius] = idata[i-kernel_radius];
    }
    if(i+BLOCK_SIZE < N)
    {
        mycache[si+BLOCK_SIZE] = idata[i+BLOCK_SIZE];
    }
}
__syncthreads();

odata[i] = -mycache[si-2]+16.f*mycache[si-1]-
    30.0f*mycache[si]+16.f*mycache[si+1]-mycache[si+2];
```

Make sure that everyone sees everything in shared memory.

On Kepler, the read-only texture memory cache can be used very easily with the __restrict__ keyword and the __ldg instruction.

```cpp
// Using the __restrict__ keyword to help the compile cache idata, using texture memory path. Kepler chips+
__global__ void laplacian_restrict( float * __restrict__ odata, const float* __restrict__ idata, int N)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    while(i < N)
    {
        if(i > 1 && i < N-2)
        {
            odata[i] = -idata[i-2]+16.f*idata[i-1]-30.0f*idata[i]+16.f*idata[i+1]-idata[i+2];
        }

        i += gridDim.x*blockDim.x;
    }
}

// Manual texture cachgin using the ldg intrinsic, using texture memory path. Kepler chips+
__global__ void laplacian_ldg( float * odata, const float* idata, int N)
{
#if __CUDA_ARCH__ >= 350
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    while(i < N)
    {
        if(i > 1 && i < N-2)
        {
            odata[i] = -__ldg(idata+i-2)+16.f*__ldg(idata+i-1)-30.0f*__ldg(idata+i)+
                       16.f*__ldg(idata+i+1)-__ldg(idata+i-2);
        }

        i += gridDim.x*blockDim.x;
    }
#endif
}
```

Final Tips

- Read the CUDA programming guide to understand the architecture: http://docs.nvidia.com/cuda/cuda-c-programming-guide/

- Learn about how GPUs hide latencies by keeping many transaction in flight (a very important topic not explicitly covered here, but covered in Tim's lecture).

- Use the NVIDIA visual profiler to identify performance issues.

    - In the SIMT paradigm, hardware dispatches/vectorizes the code, dynamic performance analysis is even more important.