Other GPGPU Programming Environments

Tim Mattson

Intel Corp.

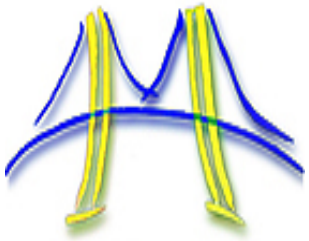
timothy.g.mattson@ intel.com

1

Disclaimer

READ THIS … its very important

- The views expressed in this talk are those of the speakers and not their employer.
- This is an academic style talk and does not address details of any particular Intel product. You will learn nothing about Intel products from this presentation.
- This was a team effort, but if we say anything really stupid, it's our fault … don't blame our collaborators.



Slides marked with this symbol were produced-with Kurt Keutzer and his team for CS194 … A UC Berkeley course on Architecting parallel applications with Design Patterns.

Programming models for SIMT platforms

- Proprietary solutions based on CUDA and OpenACC.
- But there are Open Standard responses (supported to varying degrees by all major vendors)

OpenCL

SIMT programming for CPUs, GPUs, DSPs, and FPGAs. Basically, an Open Standard that generalizes the SIMT platform pioneered by our friends at NVIDIA®

OpenMP™

OpenMP 4.0 added target and device directives … Based on the same work that was used to create OpenACC. Therefore, just like OpenACC, you can program a GPU with OpenMP!!!

*third party names are the property of their owners

Programming models for SIMT platforms

- Proprietary solutions based on CUDA and OpenACC.
- But there are Open Standard responses (supported to varying degrees by all major vendors)

We've used OpenCL to cover key GPGPU concepts. Let's briefly consider the other GPGPU models.

OpenCL

SIMT programming for CPUs, GPUs, DSPs, and FPGAs. Basically, an Open Standard that generalizes the SIMT platform pioneered by our friends at NVIDIA®

OpenMP

OpenMP 4.0 added target and device directives ... Based on the same work that was used to create OpenACC. Therefore, just like OpenACC, you can program a GPU with OpenMP!!!

*third party names are the property of their owners

Introduction to CUDA

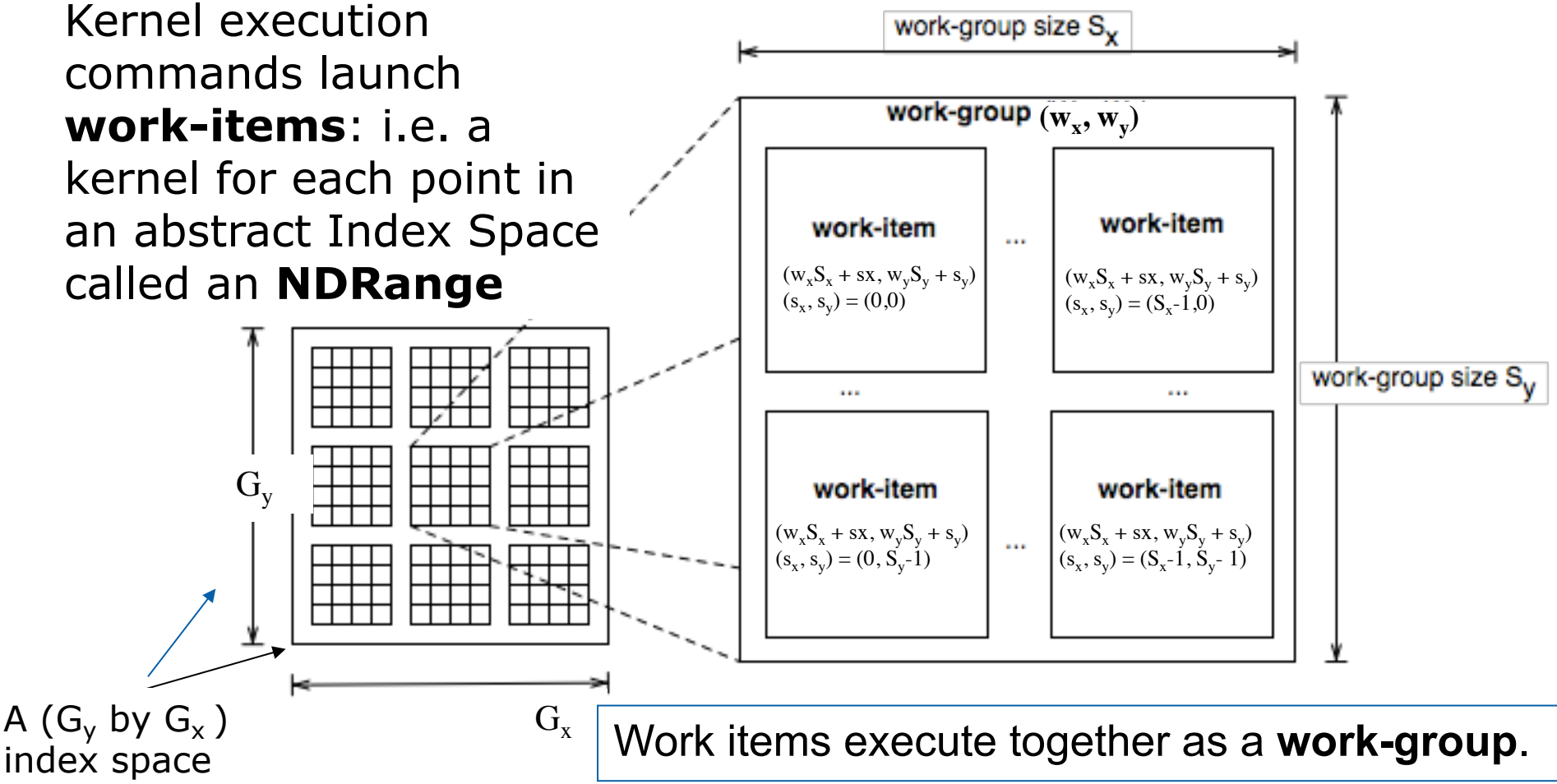
Acknowledgements: CUDA content from comes from David Sheffield, Michael Anderson, Kurt Keutzer, Mark Murphy, Bryan Catanzaro of UC Berkeley

5

Recall the OpenCL Execution Model

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

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



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

OpenCL vs. CUDA Terminology

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

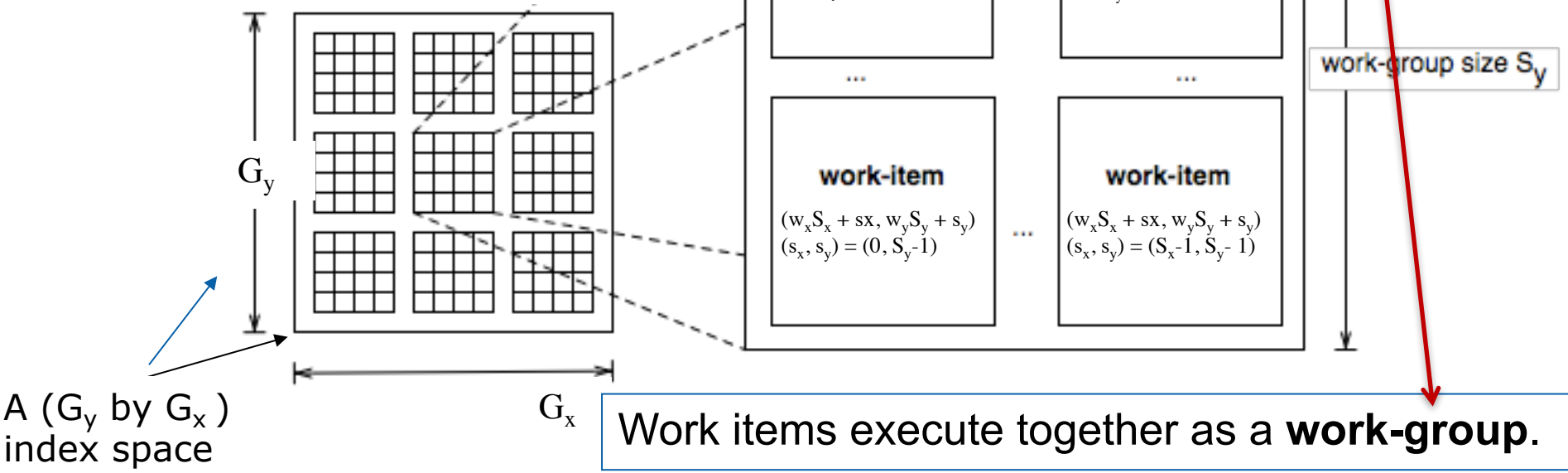
CUDA Stream

Thread Block

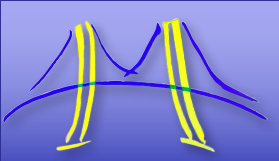
Kernel execution commands launch **work-items**: i.e. a kernel for each point in an abstract Index Space called an **NDRange**

Threads

Grid



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

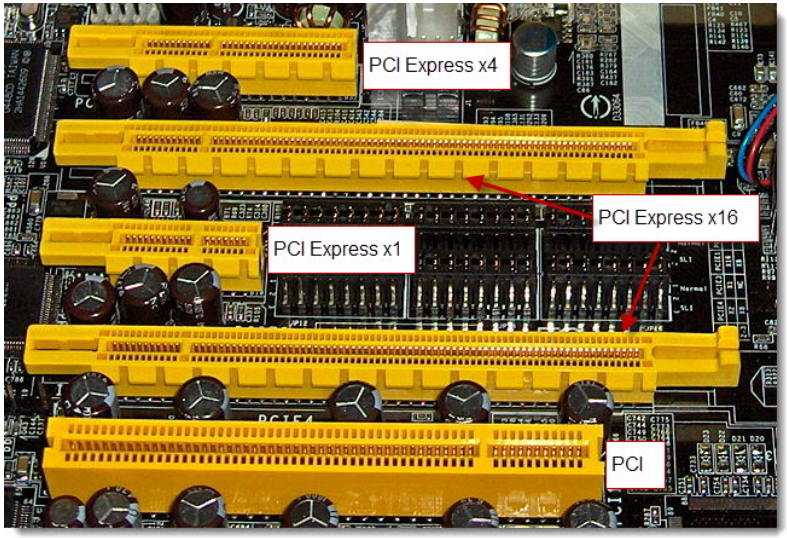


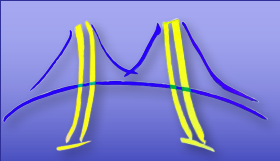
What is a CUDA warp?

- What CUDA pretends to give you:
  - A GPU presents the programmer with a “fine grained SPMD” model with independent scalar threads
- What you actually get:
  - A GPU at the lowest level is a vector processor … clusters of PEs are organized into vector units which execute a single stream of instructions.
- These clusters have a fixed size … called a warp. This size is 32.
  - Execution hardware is most efficiently utilized when all threads in a warp execute instructions from the same PC.
  - Identifiable uniquely by dividing the Thread Index by 32
  - If threads in a warp ***diverge*** (execute different PCs), then some execution pipelines go unused
  - If threads in a warp access aligned, contiguous blocks of DRAM, the accesses are ***coalesced*** into a single high-bandwidth access
- A warp is the minimum granularity of efficient SIMD execution, and the maximum hardware SIMD width in a CUDA processor

CUDA Host Runtime Support

- CUDA is a heterogeneous programming model
  - Sequential code runs in the “Host Thread” on a CPU core, and the “Device” code runs on the many cores of the GPU
  - The Host and the Device communicate via a PCI-Express link
  - The PCI-E link is slow (high latency, low bandwidth)
    - Desirable to minimize the amount of data transferred and the number of transfers





CUDA Host Runtime Support

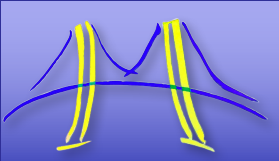
- Allocation/Deallocation of memory on the GPU:
  - **cudaMalloc(void**, int), cudaFree(void*)**
- Memory transfers to/from the GPU:
  - **cudaMemcpy(void*,void*,int, dir)**
  - **dir can be "cudaMemcpyHostToDevice" or "cudaMemcpyDeviceToHost"**

```
int main () {
    int N = (1024*1024);
 // pointers to array on the CPU
    float *h_a = new float[N];
    for(int i=0; i < N; i++) h_a[i] = i;
 // pointers to array on the GPU
    float *g_a;
    cudaMalloc(&g_a, sizeof(float)*N);
    cudaMemcpy(g_a, h_a, sizeof(float)*N,
        cudaMemcpyHostToDevice);
```

Create an array on the host CPU and fill with data

Allocate array on the GPU

Copy data from host CPU upto the GPU

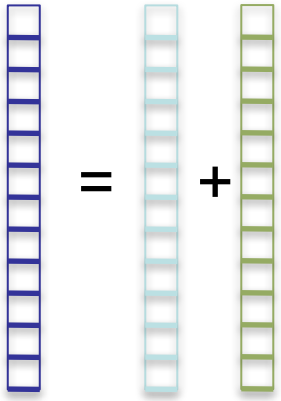


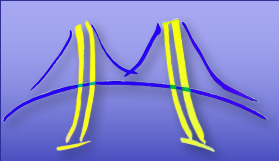
Hello World: Vector Addition (C++)

```
// 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);
}
```

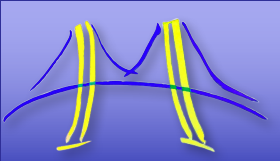




Hello World: Vector Addition (CUDA)

```
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;
    std::vecor<float> h_a[LENGTH]; // host side data
    std::vecor<float> h_c[LENGTH]; // host side result vec
   // Selecte default device
    cudaSetDevice(0);
   // ... allocate arrays, fill with data, copy data to device
    cudaMalloc (&d_a,  sizeof(float) * N);   cudaMemcpy(d_a, &h_a[0],
   sizeof(float)*LENGTH, cudaMemcpyHostToDevice);
   // Use thread blocks with 256 threads each
    vecAdd <<< (N+255)/256, 256 >>> (d_a, d_b, d_c, N);
   // ... copy data back to host
    cudaMemcpy(&h_c[0], d_c, sizeof(float)*LENGTH, cudaMemcpyDeviceToHost);
    cudaFree(d_a);
}
```

Note: this is a partial solution. We don't show all the data allocation and copies for each array (just a and c)



Hello World: Vector Addition (CUDA)

You should test that i<N in case you have extra threads when block size doesn't evenly divide N

```
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;
    std::vecor<float> h_a[LENGTH]; // host side
    std::vecor<float> h_c[LENGTH]; // host side
   // Selecte default device
    cudaSetDevice(0);
   // ... allocate arrays, fill with data, copy
    cudaMalloc (&d_a,  sizeof(float) * N);   cu
   sizeof(float)*LENGTH, cudaMemcpyHostToDevice
   // Use thread blocks with 256 threads each
    vecAdd <<< (N+255)/256, 256 >>> (d_a, d_b, d_c, N);
   // ... copy data back to host
    cudaMemc
    cudaFree(d
}
```

Note: this is a partial solution. We don't show all the data allocation and ch d c)

When you launch a kernel, you must specify two parameters of type dim3. The first specifies the global dimension of the grid (one to three dimensions) and the second species the size of a "thread block".

(N+255)/256) assures that you round up on the integer division and have enough blocks even when N isn't evenly divided by the block size.

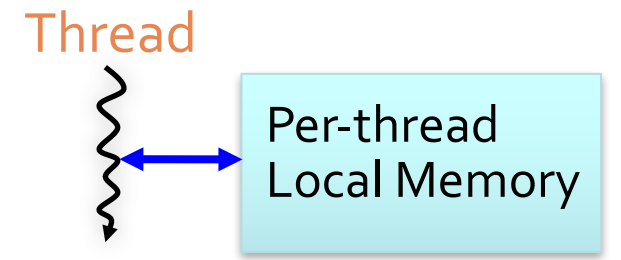
CUDA Exercise 1

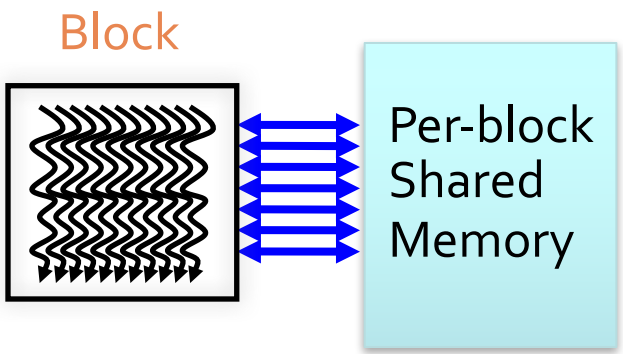
- Goal
  - Verify that you really understand the constructs by playing with the vector add program
- Problem
  - Start with the vector addition program we provide, create a CUDA version of the program. (note: the CUDA compiler requires code to end in .cu).
- Extra work
  - Experiment with different Grid sizes. Find grid sizes that lead to the best performance. Relate what you observe to the size of a CUDA Warp.

Kernel example: void __global__ vfunc(const float *a, float *c, const int N);

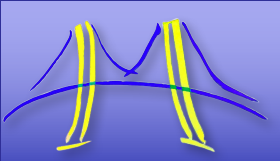
```
int i = blockIdx.x * blockDim.x + threadIdx.x;
cudaSetDevice(0);
cudaMalloc (&a,  sizeof(float) * LEN);
cudaMemcpy(a,&a[0],sizeof(float)*LEN,cudaMemcpyHostToDevice);
vfunc<<<Glob_size,Block_size>>>(a, c, LEN);
cudaMemcpy(&c[0],c, sizeof(float)*LEN,cudaMemcpyDeviceToHost);
cudaFree(a);
```

CUDA memory hierarchy





- Each CUDA thread has private access to a configurable number of registers
  - The 64 KB SM register file is partitioned among all resident threads
  - The CUDA program can trade degree of thread block concurrency for amount of per-thread state
  - Registers, stack spill into “local” DRAM if necessary
- Each thread block has private access to a configurable amount of scratchpad memory
  - Pre-Fermi SM’s have 16 KB scratchpad only
  - The available scratchpad space is partitioned among resident thread blocks, providing another concurrency-state tradeoff



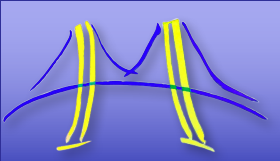
Thread-Block Synchronization

- Intra-block barrier instruction **__syncthreads()** for synchronizing accesses to **__shared__** memory
  - To guarantee correctness threads must **__syncthreads()** before reading values written by other threads
  - All threads in a block must execute the same **__syncthreads()** or the GPU will hang

"extern __shared__" allows the shared memory block to dynamically sized at run-time

```
extern __shared__ float T[];
__device__ void
transpose (float* a, int lda){
    int i = threadIdx.x, j = threadIdx.y;
    T[i + lda*j] = a[i + lda*j];
    __syncthreads();
    a[i + lda*j] = T[j + lda*i];
}
```

The function qualifier "__device__" indicates the function runs on the device, but unlike a kernel a "__device__" function cannot be called by a host … its' called by a kernel running on a device.



Using per-block shared memory

- The per-block shared memory / L1 cache is a crucial resource: without it, the performance of most CUDA programs would be hopelessly DRAM-bound
- Block-shared variables can be declared statically:

```
__shared__ int begin, end;
```

- Software-managed scratchpad memory is allocated statically:

```
__shared__ int scratch[128];
scratch[threadIdx.x] = ... ;
```

- ... or dynamically:

```
extern __shared__ int scratch[];

kernel_call <<< grid_dim, block_dim, scratch_size >>> ( ... );
```

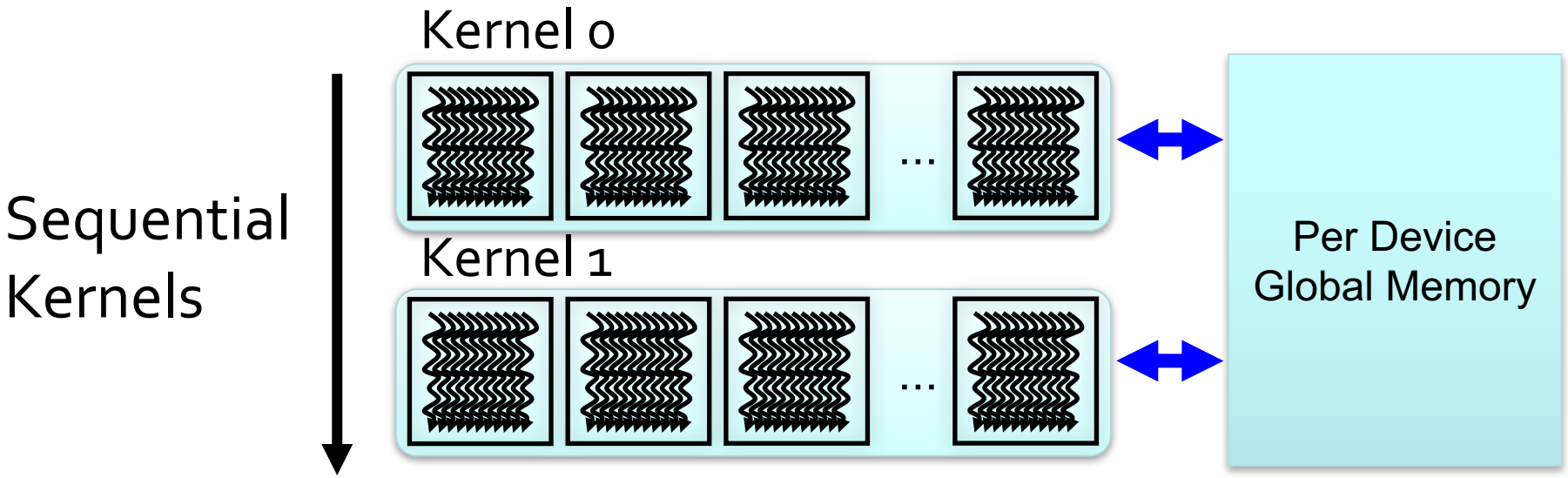
The third argument is optional and gives the size in bytes of per-block shared memory

- Most intra-block communication is via shared scratchpad:

```
scratch[threadIdx.x] = ...;
__syncthreads();
int left = scratch[threadIdx.x - 1];
```

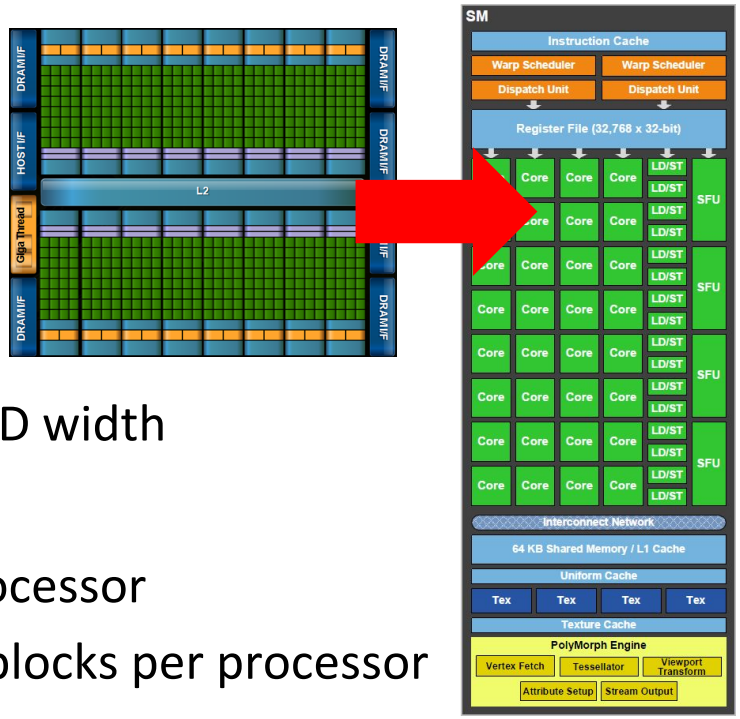
CUDA Memory Hierarchy

- Thread blocks in all Grids share access to a large pool of “Global” memory, separate from the Host CPU’s memory.
  - Global memory holds the application’s persistent state, while the thread-local and block-local memories are ephemeral
  - Global memory is much more expensive than local memories: O(100)x latency, O(1/50)x (aggregate) bandwidth
  - Registers and Cache multiply bandwidth, massive multithreading hides latency



Mapping CUDA to Nvidia GPUs

- CUDA is designed to be functionally forgiving
  - First priority: make things work. Second: get performance.
- However, to get good performance, one must understand how CUDA is mapped to Nvidia GPUs
- Threads:
  - each thread is a SIMD vector lane
- Warps:
  - A SIMD instruction acts on a “warp”
  - Warp width is 32 elements: ***LOGICAL*** SIMD width
- Thread blocks:
  - Each thread block is scheduled onto a processor
  - Peak efficiency requires multiple thread blocks per processor



CUDA Exercise 2

- Goal
  - Work with the CUDA memory hierarchy to optimize matrix multiplication.
- Problem
  - Start with the matrix multiplication program we provide to compute C= A*B in parallel
  - Parallelize with CUDA using the dot product for each element of C(I,j) as a CUDA-thread
  - Optimize performance by (1) putting rows of the A matrix in thread-local memory and (2) putting rows of the B matrix in thread-block shared memory.

Kernel example: void __global__ vfunc(const float *a, float *c, const int N);

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
cudaSetDevice(0);
cudaMalloc (&a,  sizeof(float) * LEN);
cudaMemcpy(a,&a[0],sizeof(float)*LEN,cudaMemcpyHostToDevice);
cudaMemcpy(&c[0],c, sizeof(float)*LEN,cudaMemcpyDeviceToHost);
cudaFree(a);
syncthreads();
extern __shared__ int scratch[];
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);          //A and C are order
dim3 dimGrid(N/ dimBlock.x, N/ dimBlock.y);  // N matrices
vfunc<<<dimGrid, dimBlock, loc_mem_bytes>>>(A, C, N);
```

Programming Your GPU with OpenMP*

Tim Mattson
Intel Corp.
timothy.g.mattson@ intel.com

Simon McIntosh-Smith
University of Bristol
simonm@cs.bris.ac.uk

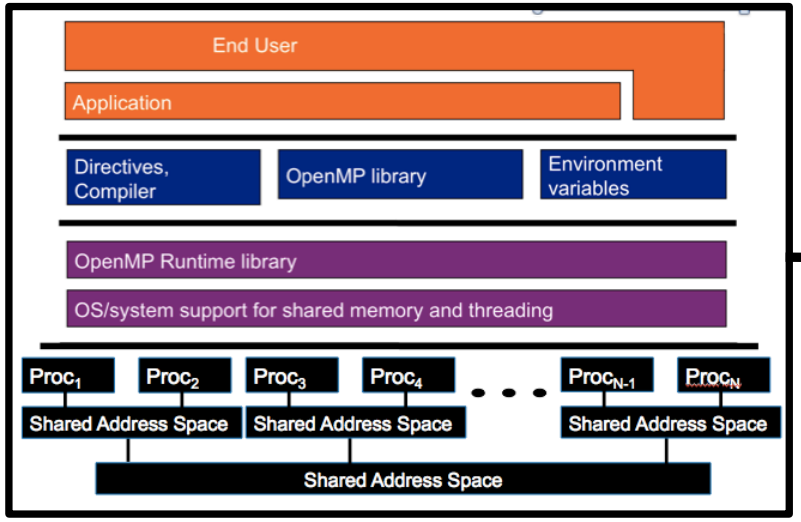
… and the McIntosh-Smith group at the University of Bristol:
Tom Deakin, **Matt Martineau** and **James Price**

* The name “OpenMP” is the property of the OpenMP Architecture Review Board.

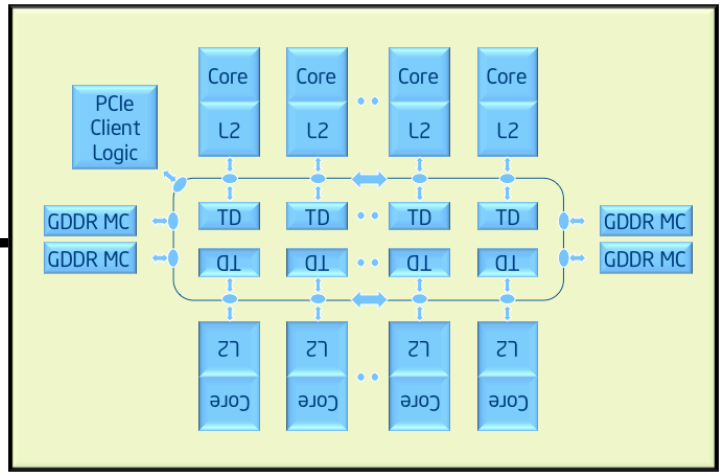
21

OpenMP basic definitions: Target solution stack

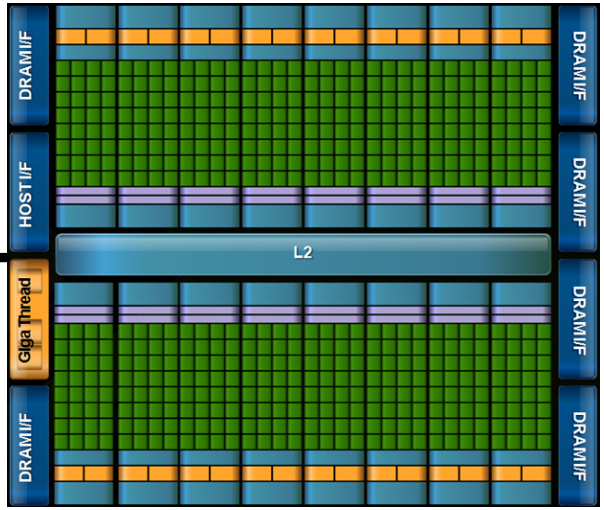
Supported (since OpenMP 4.0) with target, teams, distribute, and other constructs



Host

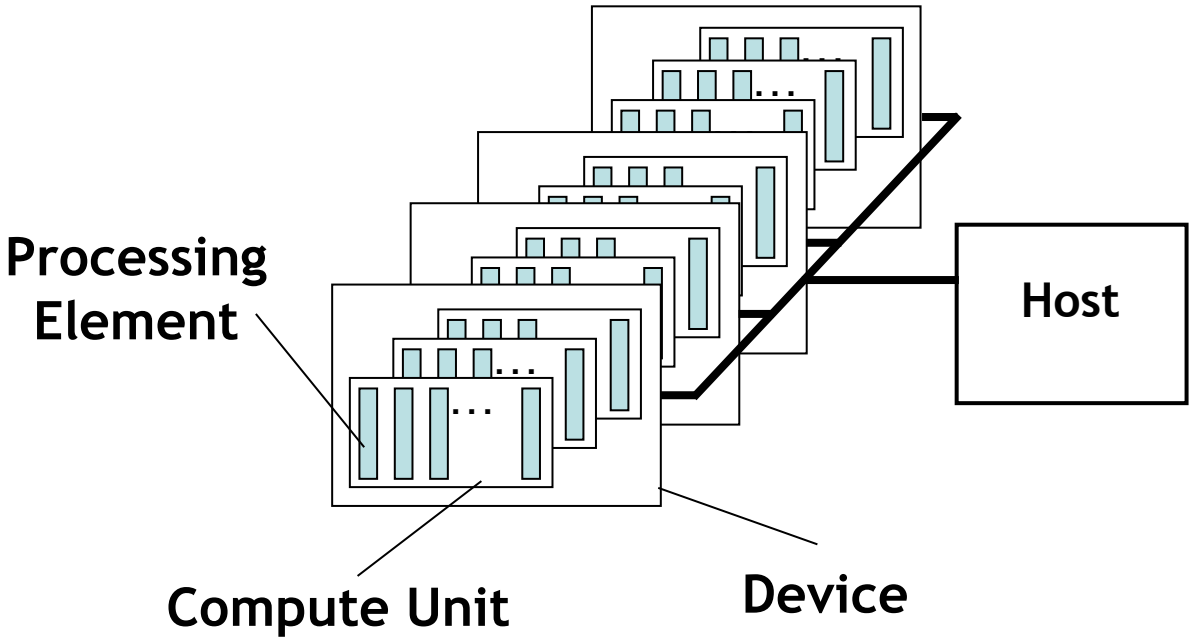


Target Device: Intel® Xeon Phi™ processor



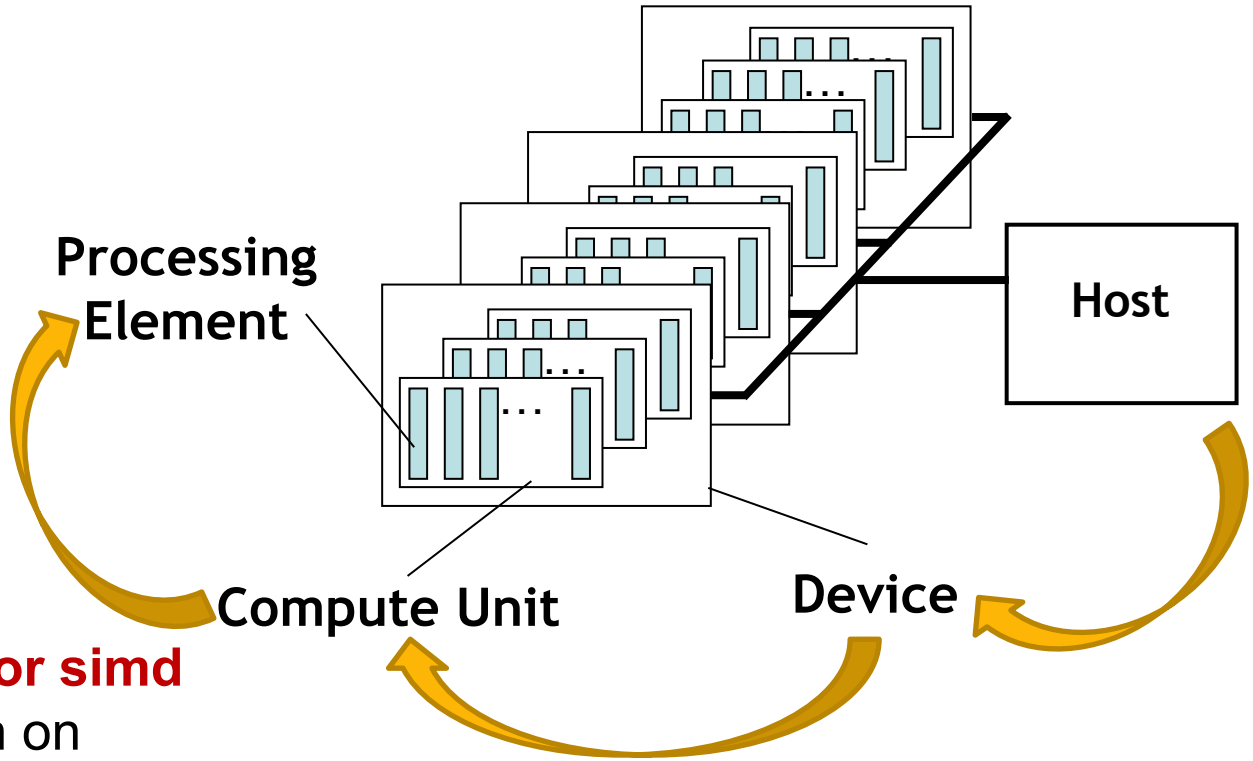
Target Device: GPU

A Generic Host/Device Platform Model



- One ***Host*** and one or more ***Devices***
  - Each Device is composed of one or more ***Compute Units***
    - Each Compute Unit is divided into one or more ***Processing Elements***
- Memory divided into ***host memory*** and ***device memory***

Our host/device Platform Model and OpenMP



Target construct to get onto a device

Parallel for simd to run on processing elements

Teams construct to create a league of teams with one team of threads on each compute unit.

Distribute clause to assign work-groups to teams.

Vadd: with OpenMP

```
// A more compact way to write the VADD code, letting the runtime
//  worry about work-group details

#pragma omp target teams distribute parallel for simd map(to:a,b) map(tofrom:c)

    for (i=0; i<N; i++)
            c[i] += a[i] + b[i];
```

In many cases, you arebetter off to just distribute the parallel loops to the league of teams and leave it to the runtime system to manage the details. This would be more portable code as well.

Controlling data movement

```
int i, a[N], b[N], c[N];
#pragma omp target map(to:a,b) map(tofrom:c)
```

Data movement can be explicitly controlled with the map clause

- The various forms of the map clause
  - **map(to:list)**: *read-only* data on the device. Variables in the list are initialized on the device using the original values from the host.
  - **map(from:list)**: *write-only* data on the device: initial value of the variable is not initialized. At the end of the target region, the values from variables in the list are copied into the original variables.
  - **map(tofrom:list)**: the effect of both a map-to and a map-from
  - **map(alloc:list)**: data is allocated and uninitialized on the device.
  - **map(list)**: equivalent to map(tofrom:list).
- For pointers you must use array notation ..
  - Map(to:a[0:N])

Default Data Sharing

- **Scalar** variables have *implicit* mapping behavior
- OpenMP 4.0 implicitly mapped all scalar variables as **tofrom**
- OpenMP 4.5 implicitly maps scalar variables as **firstprivate**

#pragma omp target teams distribute parallel for ***firstprivate(a)***

If **a** is a scalar, this is equivalent to:

#pragma omp target teams distribute parallel for

This generally means explicitly mapping **scalar** variables is unnecessary

Default Data Sharing

WARNING: Make sure not to confuse the implicit mapping of pointer variables with the data that they point to

```
int main(void) {
  int A = 0;
  int* B = malloc(sizeof(int)*N);
  #pragma omp target teams distribute parallel for
  for(int ii = 0; ii < N; ++ii) {
    // A, B, N and ii all exist here
    // B is a pointer variable! The data that B points to DOES NOT exist here!
  }
}
```

If you want to access the data that is pointed to by **B**, you will need to perform an explicit mapping using the **map** clause

Exercise: Jacobi solver, parallel for and target

- Start from the provided jacobi_solver program. Verify that you can run it serially.
- Parallelize for a CPU using the *parallel for* construct on the major loops
- Use the target directive to run on a GPU.
  - #pragma omp target
  - #pragma omp target map(to:list) map(from:list) map(tofrom:list)
  - #pragma omp target teams distribute parallel for simd

Jacobi Solver (serial 1/2)

```
<<< allocate and initialize the matrix A and >>>
<<< vectors x1, x2 and b                     >>>
while((conv > TOL) && (iters<MAX_ITERS))
  {
    iters++;
    xnew = iters % s ? x2 : x1;
    xold   = iters % s ? x1 : x2;

    for (i=0; i<Ndim; i++){
        xnew[i] = (TYPE) 0.0;
        for (j=0; j<Ndim;j++){
            if(i!=j)
              xnew[i]+= A[i*Ndim + j]*xold[j];
        }
        xnew[i] = (b[i]-xnew[i])/A[i*Ndim+i];
    }
```

Jacobi Solver (serial 2/2)

```
    //
    // test convergence
    //
    conv = 0.0;
    for (i=0; i<Ndim; i++){
        tmp  = xnew[i]-xold[i];
        conv += tmp*tmp;
    }
    conv = sqrt((double)conv);

} \\ end while loop
```

Jacobi Solver (Par Targ, 1/2)

```
while((conv > TOL) && (iters<MAX_ITERS))
  {
    iters++;
    xnew = iters % 2 ? x2 : x1;
    xold   = iters % 2 ? x1 : x2;
#pragma omp target map(tofrom:xnew[0:Ndim],xold[0:Ndim]) \
                    map(to:A[0:Ndim*Ndim], b[0:Ndim] ,Ndim)
     #pragma omp teams distribute parallel for simd private(i,j)
    for (i=0; i<Ndim; i++){
        xnew[i] = (TYPE) 0.0;
        for (j=0; j<Ndim;j++){
            if(i!=j)
              xnew[i]+= A[i*Ndim + j]*xold[j];
        }
        xnew[i] = (b[i]-xnew[i])/A[i*Ndim+i];
    }
```

Jacobi Solver (Par Targ, 2/2)

```
    //
    // test convergence
    //
    conv = 0.0;
#pragma omp target map(to:xnew[0:Ndim],xold[0:Ndim]) \
                        map(to:Ndim) map(tofrom:conv)
      #pragma omp teams distribute parallel for simd  \
                        private(i,tmp) reduction(+:conv)
    for (i=0; i<Ndim; i++){
        tmp  = xnew[i]-xold[i];
        conv += tmp*tmp;
    }
    conv = sqrt((double)conv);

} \\ end while loop
```

Jacobi Solver (Par Targ, 2/2)

```
    //
    // test convergence
    //
    conv = 0.0;
#pragma omp target map(to:xnew[0:Ndim],xold[0:Ndim]) \
                    map(to:Ndim) map(tofrom:conv)
  #pragma omp teams distribute parallel for simd  \
                    private(i,tmp) reduction(+:conv)
    for (i=0; i<Ndim; i++){
        tmp  = xnew[i]-xold[i];
        conv += tmp*tmp;
    }
    conv = sqrt((double)conv);

} \\ end while loop
```

This worked but the performance was awful. Why?

System	Implementation	Ndim = 4096
NVIDA® K20X™ GPU	Target dir per loop	131.94 secs

Cray® XC40™ Supercomputer running Cray® Compiling Environment 8.5.3.
Intel® Xeon ® CPU E5-2697 v2 @ 2.70GHz with 32 GB DDR3. NVIDIA® Tesla® K20X, 6GB.

Data movement dominates!!!

```
while((conv > TOLERANCE) && (iters<MAX_ITERS))
  { iters++;
    xnew = iters % s ? x2 : x1;
    xold   = iters % s ? x1 : x2;

    #pragma omp target map(tofrom:xnew[0:Ndim],xold[0:Ndim]) \
                    map(to:A[0:Ndim*Ndim], b[0:Ndim] ,Ndim)
     #pragma omp teams distribute parallel for simd private(i,j)
     for (i=0; i<Ndim; i++){
        xnew[i] = (TYPE) 0.0;
        for (j=0; j<Ndim;j++){
           if(i!=j)
             xnew[i]+= A[i*Ndim + j]*xold[j];
        }
        xnew[i] = (b[i]-xnew[i])/A[i*Ndim+i];
     }
  // test convergence
    conv = 0.0;
    #pragma omp target map(to:xnew[0:Ndim],xold[0:Ndim]) \
                          map(to:Ndim) map(tofrom:conv)
      #pragma omp teams distribute parallel for private(i,tmp) reduction(+:conv)
      for (i=0; i<Ndim; i++){
         tmp  = xnew[i]-xold[i];
         conv += tmp*tmp;
      }
    conv = sqrt((double)conv);
  }
```

Typically over 4000 iterations!

For each iteration, copy to device $(3*Ndim+Ndim^2)*sizeof(TYPE)$ bytes

For each iteration, copy from device 2*Ndim*sizeof(TYPE) bytes

For each iteration, copy to device 2*Ndim*sizeof(TYPE) bytes

Target data directive

- The **target data** construct creates a target data region.
- You use the map clauses for explicit data management

```
#pragma omp target data map(to: A,B) map(from: C)
{….}  // a structured block of code
```

- Data copied into the device data environment at the beginning of the directive and at the end
- Inside the **target data** region, multiple **target** regions can work with the single data region

```
#pragma omp target data map(to: A,B) map(from: C)
{
    #pragma omp target
        {do lots of stuff with A, B and C}
    {do something on the host}
    #pragma omp target
        {do lots of stuff with A, B, and C}
}
```

Target update directive

- You can update data between target regions with the target update directive.

```
#pragma omp target data map(to: A,B) map(from: C)
{
    #pragma omp target
        {do lots of stuf with A, B and C}

    #pragma omp  update from(A)

    host_do_something_with(A)

    #pragma omp update to(A)

    #pragma omp target
        {do lots of stuff with A, B, and C}
}
```

Copy A from the device onto the host.

Copy A on the host to A on the device.

Jacobi Solver (Par Targ Data, 1/2)

```
#pragma omp target data map(tofrom:x1[0:Ndim],x2[0:Ndim]) \
                        map(to:A[0:Ndim*Ndim], b[0:Ndim] ,Ndim)
while((conv > TOL) && (iters<MAX_ITERS))
  {  iters++;
     // alternate x vectors.
    xnew  = iters % 2 ? x2 : x1;
    xold   =  iters % 2 ? x1 : x2;

#pragma omp target
     #pragma omp teams distribute parallel for simd private(j)
    for (i=0; i<Ndim; i++){
        xnew[i] = (TYPE) 0.0;
        for (j=0; j<Ndim;j++){
            if(i!=j)
              xnew[i]+= A[i*Ndim + j]*xold[j];
        }
        xnew[i] = (b[i]-xnew[i])/A[i*Ndim+i];
    }
```

Jacobi Solver (Par Targ Data, 2/2)

```
      //
      // test convergence
      //
  conv = 0.0;
#pragma omp target map(tofrom: conv)
{
#pragma omp teams distribute parallel for simd  \
                          private(tmp) reduction(+:conv)
      for (i=0; i<Ndim; i++){
          tmp  = xnew[i]-xold[i];
          conv += tmp*tmp;
      }
} // end target region
  conv = sqrt((double)conv);
} // end while loop
```

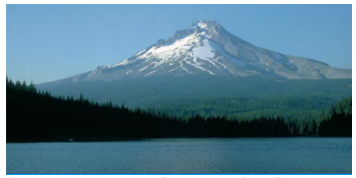
System	Implementation	Ndim = 4096
NVIDA® K20X™ GPU	Target dir per loop	131.94 secs
	Above plus target data region	18.37 secs

OpenACC

40

Directive driven programming of Heterogeneous systems

- Portland group (PGI) introduced proprietary directives for programming GPUs



- OpenMP (with help from PGI) launched a working group to define “accelerator directives” In OpenMP.

OpenMP™

- A subset of the participants grew tired of the cautious, slow and methodical approach in the OpenMP group … and split off to form their own group (OpenACC)
  - NVIDIA, Cray, PGI, and CAPS

OpenACC®
DIRECTIVES FOR ACCELERATORS

- They launched the OpenACC directive set in November of 2011 at SC11.
- At SC12:
  - The OpenACC group released a review draft of OpenACC 2.0
  - The OpenACC and OpenMP groups stated their intent to rejoin the 2 efforts.
- Summer 2013 … OpenMP released OpenMP 4.0 which includes accelerator directives for functionality analogous to OpenACC.
- Fall’2013 … gcc annouces work on OpenACC support. Should be ready soon.

OpenACC
DIRECTIVES FOR ACCELERATORS

Source: John Levesque of Cray

- A common directive programming model for today's GPUs
  - Announced at SC11 conference
  - Offers portability between compilers
    - Drawn up by: NVIDIA, Cray, PGI, CAPS
    - Multiple compilers offer portability, debugging, permanence
  - Works for Fortran, C, C++
    - Standard available at www.OpenACC-standard.org
    - Initially implementations targeted at NVIDIA GPUs
- Current version: 2.0 (November 2012)
- Compiler support:
  - Cray CCE: complete support in 2012
  - PGI Accelerator: released product in 2012
  - CAPS: released product in Q1 2012

CAPS CRAY THE SUPERCOMPUTER COMPANY NVIDIA The Portland Group

OpenACC: Core concepts

- Pragmas direct the compiler to generate code to run on the host and the GPU. Basic form of an OpenACC directive:

  #pragma acc construct [clause(s)]

- Two core constructs define work that runs on the accelerator (coarse grained, gang parallelism):
- Parallel construct:
  - Tells compiler to create a single kernel to accelerate the code.
  - This is explicit, analogous to the parallel region in OpenMP.
  - Clauses can be used with the parallel construct

    num_gang, num_worker, vector_length
  - Kernel construct:
    - Tells the compiler to accelerate the code with one or more kernels.
    - This is NOT explicit. If the compiler deems the code unsafe for execution on the GPU, it will not execute on the accelerator.

```
#pragma acc parallel
for(i=0;i<N;i++)
    A[i] = B[i]+C[i];
```

Code in block defines a kernel which runs in **gang-redundant mode** … one worker and one vector lane per gang redundantly executes the code.

OpenACC loop construct

- Typically programmers want to split loops between gangs (gang partitioned mode). The loop construct does this:

```
#pragma acc parallel loop
for(i=0;i<N;i++)
    A[i] = B[i]+C[i];
```

Loop iterations spread out across the units of execution on the accelerator.

- Can have multiple loop constructs in a single parallel region

```
#pragma acc parallel
{
#pragma acc loop
for(i=0;i<N;i++)
    A[i] = 2*A[i];

#pragma acc loop
For(i=0;i<N;i++)
    A[i] = B[i]+C[i];
}
```

Warning: there is NO IMPLIED barrier between loop constructs (i.e. acts like “nowait” in OpenMP)

Loop clauses

- Clauses can be used to direct loop scheduling … the parallel region defines gangs, workers and vectors and these clauses to the loop construct split up iterations at the indicated level

  Gang, worker, vector

- Connections between OpenACC scheduling clauses and CUDA

OpenACC	CUDA
gang	A thread block
worker	A warp (32 threads)
vector	Threads within a warp

- Reduction clause … same as OpenMP reduction

  reduction(op:vars)

OpenACC in a real program: the "vadd" program

- Let's add two vectors together …. C = A + B

```
void vadd(int n,
          const float *a,
          const float *b,
          float *restrict c)
{
  int i;
 #pragma acc parallel loop
  for (i=0; i<n; i++)
    c[i] = a[i] + b[i];
}
int main(){
float *a, *b, *c;  int n = 10000;
// allocate and fill a and b

    vadd(n, a, b, c);

}
```

Assure the compiler that c is not aliased with other pointers

Turn the loop into a kernel, move data to a device, and launch the kernel.

Host waits here until the kernel is done. Then the output array c is copied back to the host.

46

Exercise 1

- Goal
  - Verify that you can build and run an OpenACC program.
- Problem
  - Write your own simple vector add program
  - Insert the OpenACC construct and run on the GPU
  - Time the program … do you see any speedup relative to running the code on the CPU?
- Extra work
  - Experiment with the different scheduling clauses

```
#include <openacc.h>
#pragma acc parallel loop
#pragma acc parallel loop vector_length(64)
#pragma acc parallel loop reduction (+:var)
*restrict
```

The OpenACC data environment

- Data is moved as needed by the compiler on entry and exit from a parallel or kernel region.
- Data copy overhead can kill performance.
- Solution?
  - A data region to explicitly control data movement.

    #pragma acc data
  - Data movement is explicit …. Compiler no longer moves data for you.
  - Key clauses
    - **Copy, copyin, copyout**: move indicated list of variables between host and device on entry/exit form data region
    - **Create**: create the data on the accelerator.
    - **Private, firstprivate**: same meaning as with OpenMP …. Scalars are made private by default.

A more complicated example:
Jacobi iteration: OpenACC (GPU)

Turn the loop into a kernel, move data to a device, and launch the kernel.

Host waits here until the kernel is done.

```
while (err>tol && iter < iter_masx){
   err = 0.0;
   #pragma acc parallel loop reduction(max:err)
   for(int j=1; j< n-1; j++){
      for(int i=1; i<M-1; i++){
         Anew[j][i] = 0.25* (A[j][i+1] + A[j][i-1]+
                             A[j-1][i] + A[j+1][i]);
         err = max(err,abs(Anew[j][i] - A[j][i]));
       }
   }
    #pragma acc parallel loop
    for(int j=1; j< n-1; j++){
      for(int i=1; i<M-1; i++){
         A[j][i] = Anew[j]i];
       }
    }
    iter ++;
}
```

Source: based on Mark Harris of NVIDIA®, “Getting Started with OpenACC”, GPU technology Conf., 2012

A more complicated example:
Jacobi iteration: OpenACC (GPU)

A, and Anew copied between the host and the GPU on each iteration

```
while (err>tol && iter < iter_max){
   err = 0.0;
   #pragma acc parallel loop reduction(max:err)
   for(int j=1; j< n-1; j++){
      for(int i=1; i<M-1; i++){
         Anew[j][i] = 0.25* (A[j][i+1] + A[j][i-1]+
                             A[j-1][i] + A[j+1][i]);
         err = max(err,abs(Anew[j][i] - A[j][i]));
      }
   }
   #pragma acc parallel loop
   for(int j=1; j< n-1; j++){
      for(int i=1; i<M-1; i++){
         A[j][i] = Anew[j]i];
      }
   }
   iter ++;
}
```

Performance was poor due to excess memory movement overhead

Source: based on Mark Harris of NVIDIA®, “Getting Started with OpenACC”, GPU technology Conf., 2012

A more complicated example:
Jacobi iteration: OpenACC (GPU)

```
#pragma acc data copy(A), create(Anew)
while (err>tol && iter < iter_max){
   err = 0.0;
   #pragma acc parallel loop reduction(max:err)
   for(int j=1; j< n-1; j++){
      for(int i=1; i<M-1; i++){
         Anew[j][i] = 0.25* (A[j][i+1] + A[j][i-1]+
                             A[j-1][i] + A[j+1][i]);
         err = max(err,abs(Anew[j][i] - A[j][i]));
       }
    }
    #pragma acc parallel loop
    for(int j=1; j< n-1; j++){
      for(int i=1; i<M-1; i++){
         A[j][i] = Anew[j]i];
       }
    }
    iter ++;
}
```

Create a data region on the GPU. Copy A once onto the GPU, and create Anew on the device (no copy from host)

Copy A back out to host … but only once

Source: based on Mark Harris of NVIDIA®, "Getting Started with OpenACC", GPU technology Conf., 2012

OpenACC vs. OpenMP

- OpenACC suffers from a form of the CUDA™ problem … it is focused on GPUs only … and mostly those from NVIDIA.
- With the purchase of PGI by NVIDIA, the chances of long term cross platform support is reduced.
- OpenACC is an open standard (which is great) but its only a small subset of the industry … not the broad coverage of OpenMP or OpenCL.
- The OpenMP 4.0 accelerator directives:
  - Mesh with the OpenMP directives so you can use both in a single program.
  - They are designed to support many core CPUs (such as MIC), multicore CPUs, and GPUs.
  - Support a wider range of algorithms (though OpenACC 2.0 closes this gap).
- So … OpenMP directive set will hopefully displace the OpenACC directives as they are finalized and deployed into the market.

A more complicated example:

Jacobi iteration: OpenMP accelerator directives

```
#pragma omp target data map(A, Anew)
while (err>tol && iter < iter_max){
   err = 0.0;
   #pragma target
   #pragma omp parallel for reduction(max:err)
   for(int j=1; j< n-1; j++){
      for(int i=1; i<M-1; i++){
         Anew[j][i] = 0.25* (A[j][i+1] + A[j][i-1]+
                             A[j-1][i] + A[j+1][i]);
         err = max(err,abs(Anew[j][i] - A[j][i]));
       }
    }
    #pragma omp target
    #pragma omp parallel for
    for(int j=1; j< n-1; j++){
      for(int i=1; i<M-1; i++){
         A[j][i] = Anew[j]i];
       }
    }
    iter ++;
}
```

Create a data region on the GPU. Map A and Anew onto the target device

Uses existing OpenMP constructs such as parallel and for

Copy A back out to host … but only once

Conclusion

- If you want to program a GPU, do so with an open standard:
  - OpenCL
  - OpenMP
- Certain vendors, however, put more energy into their own solutions … which are VERY well supported and work VERY well
  - OpenACC
  - CUDA
- Don’t reward bad behavior by using proprietary solutions, and remember ….