Advanced Computing for Engineering Applications

Dan Negrut

Simulation-Based Engineering Lab
Wisconsin Applied Computing Center
Department of Mechanical Engineering
Department of Electrical and Computer Engineering
University of Wisconsin-Madison

Darmstadt University
06-10 October 2014
What We Covered Yesterday…

- CUDA API
- The GPU computing Memory Ecosystem
- Execution scheduling on the GPU
- Atomics in CUDA
What We’ll Be Covering Today

- Profiling, quick overview
- CUDA optimization issues
- CUDA advanced memory issues (unified memory in CUDA 6.0)
- Getting started with multi-core parallel computing as enabled by OpenMP
Dominik’s Point Was Good: No Race Issue Here

```c
__global__ void testKernel(int *x, int *y) {
    int i = threadIdx.x;
    if (i == 0) *x = 1;
    if (i == 1) *y = *x;
}

int main() {
    int* dArr;
    int hArr[2] = {23, -5};
    cudaMalloc(&dArr, 2 * sizeof(int));
    cudaMemcpy(dArr, hArr, 2 * sizeof(int), cudaMemcpyHostToDevice);
    testKernel <<<1, 2 >>>(dArr, dArr + 1);
    cudaMemcpy(hArr, dArr, 2 * sizeof(int), cudaMemcpyDeviceToHost);
    printf("x = %d\n", hArr[0]);
    printf("y = %d\n", hArr[1]);
    return 0;
}
```
CUDA GPU Programming
~ Resource Management Considerations ~
What Do I Mean By “Resource Management”?  

- The GPU is a resourceful device

- What do you have to do to make sure you capitalize on these resources?  
  - In other words, how can you ensure that all the SPs are busy all the time?

- The three factors that come into play are  
  - How many threads you decide to use in each block  
  - What register requirements end up associated with a thread  
  - How much shared memory you assign to one block of threads
The Bottom Line

- You want to have as many warps resident on one SM as possible
  - 32 warps on Tesla
  - 48 warps on Fermi
  - 64 warps on Kepler and Maxwell

- Why?
  - If you have many warps you stand a better chance of hiding latency of memory accesses with useful execution
  - In other words, you don’t see memory as a bottleneck anymore
Execution Model, Key Observations [1 of 2]

- Each block is executed on *one* Stream Multiprocessor (SM)
  - “block is executed” above means that all the threads that make up that block execute a kernel
  - Each block is split into warps of threads. The warps are executed one at a time by the SPs of the SM (time slicing in warp execution is the norm)
Execution Model, Key Observations [2 of 2]

- A Stream Multiprocessor can execute multiple blocks concurrently
  - If N blocks get executed simultaneously on the same SM, then:
    - The total amount of shared memory available on SM should accommodate the N blocks
    - NOTE: The shared memory is an attribute of the block, not of a thread
  - If N blocks get executed simultaneously on the same SM, then
    - The register files associated with the SM is split amongst the union of threads associated with the N blocks
    - NOTE: The less registers get used per thread, the better (you potentially can squeeze more blocks on a SM) better resource utilization
### Technical Specifications and Features

#### [Short Detour]

<table>
<thead>
<tr>
<th>Technical specifications</th>
<th>Compute capability (version)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
</tr>
<tr>
<td>Maximum dimensionality of grid of thread blocks</td>
<td>2</td>
</tr>
<tr>
<td>Maximum x-, y-, or z-dimension of a grid of thread blocks</td>
<td>65535</td>
</tr>
<tr>
<td>Maximum dimensionality of thread block</td>
<td></td>
</tr>
<tr>
<td>Maximum x- or y dimension of a block</td>
<td>512</td>
</tr>
<tr>
<td>Maximum z-dimension of a block</td>
<td></td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td>512</td>
</tr>
<tr>
<td>Warp size</td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident blocks per multiprocessor</td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident warps per multiprocessor</td>
<td>24</td>
</tr>
<tr>
<td>Maximum number of resident threads per multiprocessor</td>
<td>758</td>
</tr>
<tr>
<td>Number of 32-bit registers per multiprocessor</td>
<td>8 K</td>
</tr>
<tr>
<td>Maximum number of 32-bit registers per thread</td>
<td>128</td>
</tr>
<tr>
<td>Maximum amount of shared memory per multiprocessor</td>
<td>16 KB</td>
</tr>
<tr>
<td>Number of shared memory banks</td>
<td>16</td>
</tr>
<tr>
<td>Amount of local memory per thread</td>
<td>16 KB</td>
</tr>
<tr>
<td>Constant memory size</td>
<td></td>
</tr>
<tr>
<td>Cache working set per multiprocessor for constant memory</td>
<td></td>
</tr>
</tbody>
</table>
Resource Utilization

- There is an “occupancy calculator” that can tell you what percentage of the HW gets utilized by your kernel

- Assumes the form of an Excel spreadsheet

- Requires the following input
  - Threads per block
  - Registers per thread
  - Shared memory per block

- Google “occupancy calculator cuda” to access it
**CUDA GPU Occupancy Calculator**

Click here for detailed instructions on how to use this occupancy calculator.

For more information on NVIDIA CUDA, visit [http://developer.nvidia.com/cuda](http://developer.nvidia.com/cuda)

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

#### Varying Block Size

![Graph showing varying block size](chart1.png)

#### Varying Register Count

![Graph showing varying register count](chart2.png)

#### Varying Shared Memory Usage

![Graph showing varying shared memory usage](chart3.png)

---

Google: “cuda occupancy calculator”
CUDA: Measuring Speed of Execution
[Gauging Greatness]
Code Timing/Profiling

- Lazy man’s solution
  - Do nothing, instruct the executable to register crude profiling info

- Advanced approach: use NVIDIA’s nvvp Visual Profiler
  - Visualize CPU and GPU activity
  - Identify optimization opportunities
  - Allows for automated analysis
  - nvvp is a cross platform tool (linux, mac, windows)
Lazy Man’s Solution…

- Set the right environment variable and run your executable [illustrated on Euler]:

```plaintext
>> nvcc -O3 -gencode arch=compute_20,code=sm_20 testV4.cu -o testV4_20
>> export CUDA_PROFILE=1
>> ./testV4_20
>> cat cuda_profile_0.log
```

```
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GTX 480
# TIMESTAMPFACTOR fffff6c689a404a8
method,gputime,cputime,occupancy
method=[ memcpyHtoD ] gputime=[ 1001.952 ] cputime=[ 1197.000 ]
method=[ memcpyDtoH ] gputime=[ 1394.144 ] cputime=[ 2533.000 ]
```
Lazy Man’s Solution…

>> nvcc -O3 -gencode arch=compute_20,code=sm_20 testV4.cu -o testV4_20
>> ./testV4_20

# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GTX 480
# TIMESTAMPFACTOR ffffffff6c689a404a8
method,gputime,cputime,occupancy
method=[ memcpyHtoD ] gputime=[ 1001.952 ] cputime=[ 1197.000 ]
method=[ memcpyDtoH ] gputime=[ 1394.144 ] cputime=[ 2533.000 ]

>> nvcc -O3 -gencode arch=compute_10,code=sm_10 testV4.cu -o testV4_10
>> ./testV4_10

# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GT 130M
# TIMESTAMPFACTOR 12764ee9b183e71e
method,gputime,cputime,occupancy
method=[ memcpyHtoD ] gputime=[ 1815.424 ] cputime=[ 2787.856 ]
method=[ _Z14applyStencil1DiiPKfPfS1_ ] gputime=[ 47332.9 ] cputime=[ 8.469 ] occupancy=[0.67]
method=[ memcpyDtoH ] gputime=[ 3535.648 ] cputime=[ 4555.577 ]

My HP laptop
Lazy Man’s Solution…

```
>> nvcc -O3 -gencode arch=compute_20,code=sm_20 testV4.cu -o testV4_20
>> ./testV4_20

# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GTX 480
# TIMESTAMPFACTOR fffff6c689a404a8
method,gputime,cputime,occupancy
method=[ memcpyHtoD ] gputime=[ 1001.952 ] cputime=[ 1197.000 ]
method=[ memcpyDtoH ] gputime=[ 1394.144 ] cputime=[ 2533.000 ]
```

Compute capability 2.0 (Fermi)

```
>> nvcc -O3 -gencode arch=compute_10,code=sm_10 testV4.cu -o testV4_10
>> ./testV4_10

# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GT 130M
# TIMESTAMPFACTOR 12764ee9b183e71e
method,gputime,cputime,occupancy
method=[ memcpyHtoD ] gputime=[ 1815.424 ] cputime=[ 2787.856 ]
method=[ _Z14applyStencil1DiPiPKfPfS1_ ] gputime=[ 47332.9 ] cputime=[ 8.469 ] occupancy=[0.67]
method=[ memcpyDtoH ] gputime=[ 3535.648 ] cputime=[ 4555.577 ]
```

Compute capability 1.0 (Tesla/G80)
Lazy Man’s Solution...

```bash
>> nvcc -O3 -gencode arch=compute_20,code=sm_20 testV4.cu -o testV4_20
>> ./testV4_20

# CUDA PROFILE LOG VERSION 2.0
# CUDA DEVICE 0 GeForce GTX 480
# TIMESTAMPFACTOR fffff6c689a404a8
method,gputime,cputime,occupancy
method=[ memcpyHtoD ] gputime=[ 1001.952 ] cputime=[ 1197.000 ]
method=[ memcpyDtoH ] gputime=[ 1394.144 ] cputime=[ 2533.000 ]
```

Compute capability 2.0 (Fermi)

```bash
>> nvcc -O3 -gencode arch=compute_10,code=sm_10 testV4.cu -o testV4_10
>> ./testV4_10

# CUDA PROFILE LOG VERSION 2.0
# CUDA DEVICE 0 GeForce GT 130M
# TIMESTAMPFACTOR 12764ee9b183e71e
method,gputime,cputime,occupancy
method=[ memcpyHtoD ] gputime=[ 1815.424 ] cputime=[ 2787.856 ]
method=[ _Z14applyStencil1DiiPKfPfS1_ ] gputime=[ 47332.9 ] cputime=[ 8.469 ] occupancy=[0.67]
method=[ memcpyDtoH ] gputime=[ 3535.648 ] cputime=[ 4555.577 ]
```

Compute capability 1.0 (Tesla/G80)
nvvp: NVIDIA Visual Profiler

- Available on Euler

- Provides a nice GUI and ample information regarding your run

- Many bells & whistles
  - Covering here the basics through a 1D stencil example

- Acknowledgement: Discussion on nvvp uses material from NVIDIA (S. Satoor).
  - Slides that include this material marked by “NVIDIA [S. Satoor]→” sign at bottom of slide
1D Stencil: A Common Algorithmic Pattern
[Problem Used to Introduce Profiling Tool]

- Applying a 1D stencil to a 1D array of elements
  - Function of input elements within a radius

- Fundamental to many algorithms
  - Standard discretization methods, interpolation, convolution, filtering,…

- This example will use weighted arithmetic mean
Serial Algorithm

(radius = 3)

\( f \)

\( \approx \) = CPU Thread

in \( \ldots \) out \( \ldots \)

NVIDIA [S. Satoor]→
Serial Algorithm

\[ \cdots \]

in \[ \cdots \]

(out \[ \cdots \]

\( f \)

Repeat for each element

\( \Rightarrow \) = CPU Thread

(radius = 3)

NVIDIA [S. Satoor]→
int main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out = (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);

    applyStencil1D(RADIUS, N - RADIUS, weights, in, out);

    //free resources
    free(weights); free(in); free(out);
}

void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    for (int i = sIdx; i < eIdx; i++) {
        out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
int main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    // allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out = (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);

    applyStencil1D(RADIUS, N - RADIUS, weights, in, out);

    // free resources
    free(weights); free(in); free(out);
}

void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    for (int i = sIdx; i < eIdx; i++) {
        out[i] = 0;
        // loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
int main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out = (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    applyStencil1D(RADIUS, N - RADIUS, weights, in, out);
    //free resources
    free(weights); free(in); free(out);
}

void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    for (int i = sIdx; I < eIdx; i++) {
        out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
int main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out = (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    applyStencil1D(RADIUS, N - RADIUS, weights, in, out);

    //free resources
    free(weights); free(in); free(out);
}

void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    for (int i = sIdx; i < eIdx; i++) {
        out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}

<table>
<thead>
<tr>
<th>CPU</th>
<th>MEElements/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>i7-930</td>
<td>30</td>
</tr>
</tbody>
</table>

NVIDIA [S. Satoor]→
Parallel Algorithm

Serial: One element at a time

Parallel: Many elements at a time

 Nvidia [S. Satoor] →
The Parallel Implementation

```c
void main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out = (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights; cudaMalloc(&d_weights, wsize);
    float *d_in; cudaMalloc(&d_in, size);
    float *d_out; cudaMalloc(&d_out, size);

    cudaMemcpy(d_weights, weights, wsize, cudaMemcpyHostToDevice);
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    applyStencil1D<<<N/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

    //free resources
    free(weights); free(in); free(out);
    cudaFree(d_weights); cudaFree(d_in); cudaFree(d_out);
}
```

```c
__global__ void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    int i = sIdx + blockIdx.x*blockDim.x + threadIdx.x;
    if (i < eIdx) {
        out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
```

The GPU kernel

NVIDIA [S. Satoor]→
The Parallel Implementation

```c
void main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out = (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights; cudaMalloc(&d_weights, wsize);
    float *d_in; cudaMalloc(&d_in, size);
    float *d_out; cudaMalloc(&d_out, size);

    cudaMemcpy(d_weights, weights, wsize, cudaMemcpyHostToDevice);
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    applyStencil1D<<<N/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

    //free resources
    free(weights); free(in); free(out);
    cudaFree(d_weights); cudaFree(d_in); cudaFree(d_out);
}
```

Allocate GPU memory
```c
__global__ void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    int i = sIdx + blockIdx.x * blockDim.x + threadIdx.x;
    if (i < eIdx) {
        out[i] = 0;
        // loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
```
The Parallel Implementation

```c
void main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);

    // allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out = (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights; cudaMalloc(&d_weights, wsize);
    float *d_in; cudaMalloc(&d_in, size);
    float *d_out; cudaMalloc(&d_out, size);

    cudaMemcpy(d_weights, weights, wsize, cudaMemcpyHostToDevice);
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    applyStencil1D<<<N/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

    // free resources
    free(weights); free(in); free(out);
    cudaFree(d_weights); cudaFree(d_in); cudaFree(d_out);
}
```

Launch a GPU thread for each element
void main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out = (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights; cudaMalloc(&d_weights, wsize);
    float *d_in; cudaMalloc(&d_in, size);
    float *d_out; cudaMalloc(&d_out, size);

    cudaMemcpy(d_weights, weights, wsize, cudaMemcpyHostToDevice);
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    applyStencil1D<<<N/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

    //free resources
    free(weights); free(in); free(out);
    cudaFree(d_weights); cudaFree(d_in); cudaFree(d_out);
}

__global__ void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    int i = sIdx + blockIdx.x*blockDim.x + threadIdx.x;
    if( i < eIdx ) {
        out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}

Get the array index for each thread.

Each thread executes applyStencil1D kernel.
The Parallel Implementation

```c
void main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    // allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out= (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights; cudaMalloc(&d_weights, wsize);
    float *d_in; cudaMalloc(&d_in, size);
    float *d_out; cudaMalloc(&d_out, size);

    cudaMemcpy(d_weights,weights,wsize,cudaMemcpyHostToDevice);
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    applyStencil1D<<<N/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

    // free resources
    free(weights); free(in); free(out);
    cudaFree(d_weights); cudaFree(d_in); cudaFree(d_out);
}
```

```c
_global_ void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    int i = sIdx + blockIdx.x*blockDim.x + threadIdx.x;
    if( i < eIdx ) {
        out[i] = 0;
        // loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
```
```c
void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    int i = sIdx + blockIdx.x*blockDim.x + threadIdx.x;
    if(i < eIdx) {
        out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
```

```c
void main() {
    int size = N * sizeof(float);
    int wsize = (2*RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out= (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights; cudaMalloc(&d_weights, wsize);
    float *d_in; cudaMalloc(&d_in, size);
    float *d_out; cudaMalloc(&d_out, size);
    cudaMemcpy(d_weights, weights, wsize, cudaMemcpyHostToDevice);
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    applyStencil1D<<<N/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);
    free(weights); free(in); free(out);
    cudaFree(d_weights); cudaFree(d_in); cudaFree(d_out);
}
```

### The Parallel Implementation

<table>
<thead>
<tr>
<th>Device</th>
<th>Algorithm</th>
<th>MEElements/s</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>i7-930*</td>
<td>Optimized &amp; Parallel</td>
<td>130</td>
<td>1x</td>
</tr>
<tr>
<td>Tesla C2075</td>
<td>Simple</td>
<td>285</td>
<td>2.2x</td>
</tr>
</tbody>
</table>

NVIDIA [S. Satoor]→
Application Optimization Process [Revisited]

- Identify Optimization Opportunities
  - 1D stencil algorithm

-Parallelize with CUDA, confirm functional correctness
  - `cuda-gdb`, `cuda-memcheck`

- Optimize
  - …dealing with this next
NVIDIA Visual Profiler

Timeline of CPU and GPU activity

Kernel and memcpy details
NVIDIA Visual Profiler

CUDA API activity on CPU

Memcpy and kernel activity on GPU

NVIDIA [S. Satoor]→
Detecting Low Memory Throughput

- Spend majority of time in data transfer
  - Often can be overlapped with preceding or following computation

- From timeline can see that throughput is low
  - PCIe x16 can sustain > 5GB/s
Visual Profiler Analysis

- How do we know when there is an optimization opportunity?
  - Timeline visualization seems to indicate an opportunity
  - Documentation gives guidance and strategies for tuning
    - CUDA Best Practices Guide – link on the website
    - CUDA Programming Guide – link on the website

- Visual Profiler analyzes your application
  - Uses timeline and other collected information
  - Highlights specific guidance from Best Practices
  - Like having a customized Best Practices Guide for your application
Visual Profiler Analysis

Several types of analysis are provided

Analysis pointing out low memcpy throughput

The amount of time performing compute is low relative to the amount of time required for memcpy.

Low Memcpy/Compute Overlap [ 0 ns / 8.176 ms = 0% ]
The percentage of time when memcpy is being performed in parallel with compute is low.

Low Memcpy Throughput [ 997.19 MB/s avg, for memcpys accounting for 68.1% of a ]
The memory copies are not fully using the available host to device bandwidth.

Low Memcpy Overlap [ 0 ns / 15.79 ms = 0% ]
Online Optimization Help

Low Memcpy Throughput [ 997.19 MB/s avg, for memcpys accounting for 68.1% of all memcpy time ]

The memory copies are not fully using the available host to device bandwidth.

Each analysis has link to Best Practices documentation

Pinned Memory

Page-locked or pinned memory transfers attain the highest bandwidth between the host and the device. On PCIe x16 Gen2 cards, for example, pinned memory can attain greater than 5 GBps transfer rates.

Pinned memory is allocated using the cudaMallocHost() or cudaMemcpyHost() functions in the Runtime API. The bandwidthTest.cu program in the CUDA SDK shows how to use these functions as well as how to measure memory transfer performance.

Pinned memory should not be overused. Excessive use can reduce overall system performance because pinned memory is a scarce resource. How much is too much is difficult to tell in advance, so as with all optimizations, test the applications and the systems they run on for optimal performance parameters.
int main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    // allocate resources
    float *weights; cudaMallocHost(&weights, wsize);
    float *in; cudaMallocHost(&in, size);
    float *out; cudaMallocHost(&out, size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights; cudaMalloc(&d_weights);
    float *d_in; cudaMalloc(&d_in);
    float *d_out; cudaMalloc(&d_out);
    ...
Pinned CPU Memory Result

GPU PINNED: 0.0297912 seconds, 4.50528 GBytes/s, 0.563158 GElements/s
### Pinned CPU Memory Result

<table>
<thead>
<tr>
<th>Device</th>
<th>Algorithm</th>
<th>MEElements/s</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>i7-930*</td>
<td>Optimized &amp; Parallel</td>
<td>130</td>
<td>1x</td>
</tr>
<tr>
<td>Tesla C2075</td>
<td>Simple</td>
<td>285</td>
<td>2.2x</td>
</tr>
<tr>
<td>Tesla C2075</td>
<td>Pinned Memory</td>
<td>560</td>
<td>4.3x</td>
</tr>
</tbody>
</table>

*4 cores + hyperthreading

NVIDIA [S. Satoor]→
CUDA Optimization: Wrap Up…
Performance Optimization
[Wrapping Up…]

- Many rules and ways to write good CUDA code
- Useful to sort this collection of recommendations based on their importance
- Writing CUDA software is a craft/skill that is learned
  - Just like playing a game well: get to know the rules and then practice
  - Recommendations made have high, medium, and low priority
    - For more details, check the CUDA C Best Practices Guide:

Before Diving In

- For GPU computing, you need a large problem that exposes a large amount of fine grain parallelism

- “Large problem” – what is that?
  - You need to process large amounts of data
  - You are in business if the amount of time to process the data takes significantly longer than the sum of the overhead stemming from
    - Moving data to/from device
    - Launching the kernel

- What you’d also like to see: data is organized nicely and can be accessed in an orderly fashion (instead of jumping all over the place to access pieces of the data)
  - Not pretty: large sparse matrix that needs to be factored and you jump all over picking up entries
Writing CUDA Software: High-Priority Recommendations

1. To get the maximum benefit from CUDA, focus first on finding ways to parallelize sequential code.

2. Use the actual bandwidth and flop rate of your computation as a metric when measuring performance and optimization benefits.

3. Minimize data transfer between the host and the device, even if it means running some kernels on the device that do not show performance gains when compared with running them on the host CPU.

Writing CUDA Software: High-Priority Recommendations

4. Strive to have aligned and coalesced global memory accesses

5. Minimize the use of global memory. Prefer shared memory access where possible (consider tiling as a design solution)

6. Avoid different execution paths within the same warp

Writing CUDA Software: Medium-Priority Recommendations

1. Accesses to shared memory should be designed to avoid serializing requests due to bank conflicts.

2. To hide latency arising from register dependencies, maintain sufficient numbers of active threads per multiprocessor (i.e., sufficient occupancy).

3. The number of threads per block should be a multiple of 32 threads, because this provides optimal computing efficiency and facilitates coalescing.

Writing CUDA Software: Medium-Priority Recommendations

4. Use the fast math library whenever speed is very important and you can live with a bit of accuracy loss

Writing CUDA Software:
Low-Priority Recommendations

1. Use shift operations to avoid expensive division and modulo calculations

2. Avoid automatic conversion of doubles to floats

GPU Computing using thrust
3 Ways to Accelerate on GPU

- Libraries
- Directives
- Programming Languages

Easiest Approach → Maximum Performance

Direction of increased performance (and effort)
Acknowledgments

- The **thrust** slides include material provided by Nathan Bell of NVIDIA

- Slightly modified, assuming responsibility for any mistakes
Design Philosophy, **thrust**

- Increase programmer productivity
  - Build complex applications quickly

- Adopt a generic programming angle
  - Leverage a template-based approach

- Should run fast
  - Efficient mapping to hardware
What is thrust?

- A template library for CUDA
  - Mimics the C++ STL

- Containers
  - On host and device

- Algorithms
  - Sorting, reduction, scan, etc.
What is **thrust**?

[Cntd.]

- **thrust** is a header library – all the functionality is accessed by `#include`-ing the appropriate `thrust` header file

- Program is compiled with `nvcc` as per usual, no special tools are required

- Lots of C++ syntax, related to high-level host-side code that you write
  - The concept of execution configuration, shared memory, etc. : it’s all gone
Example: Vector Addition

```c
for (int i = 0; i < N; i++)
    Z[i] = X[i] + Y[i];
```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <iostream>

int main(void) {
    thrust::device_vector<float> X(3);
    thrust::device_vector<float> Y(3);
    thrust::device_vector<float> Z(3);


    thrust::transform(X.begin(), X.end(),
                      Y.begin(),
                      Z.begin(),
                      thrust::plus<float>());

    for (size_t i = 0; i < Z.size(); i++)
        std::cout << "Z[" << i << "] = " << Z[i] << "\n";

    return 0;
}
Example, Vector Addition

[negrut@euler01 CodeBits]$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2011 NVIDIA Corporation
Built on Thu_Jan_12_14:41:45_PST_2012
Cuda compilation tools, release 4.1, V0.2.1221

[negrut@euler01 CodeBits]$ nvcc -O2 exThrust.cu -o exThrust.exe
[negrut@euler01 CodeBits]$ ./exThrust.exe
Z[0] = 25
Z[1] = 55
Z[2] = 40

Note: file extension should be .cu
Example: SAXPY

```c
for (int i = 0; i < N; i++)
    Z[i] = a * X[i] + Y[i];
```
```cpp
struct saxpy
{
    float a;

    saxpy(float a) : a(a) {}

    __host__ __device__
    float operator()(float x, float y)
    {
        return a * x + y;
    }
};

int main(void)
{
    thrust::device_vector<float> X(3), Y(3), Z(3);


    float aVal = 2.0f;

    thrust::transform(X.begin(), X.end(),
                       Y.begin(),
                       Z.begin(),
                       saxpy(aVal));

    for (size_t i = 0; i < Z.size(); i++)
        std::cout << "Z[" << i << "] = " << Z[i] << "\n";

    return 0;
}
```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>

int main(void) {
    // generate about 32M random numbers on the host
    thrust::host_vector<int> h_vec(32 << 20);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);

    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;

    // sort data on the device (846M keys per sec on GeForce GTX 480)
    thrust::sort(d_vec.begin(), d_vec.end());

    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

    return 0;
}
Containers

- Concise and readable code
  - Avoids common memory management errors
    - e.g.: Vectors automatically release memory when they go out of scope

```cpp
// allocate host vector with two elements
thrust::host_vector<int> h_vec(2);

// ... compute entries in h_vec ...

// copy host vector to device
thrust::device_vector<int> d_vec = h_vec;

// write device values
d_vec[0] = 13;
d_vec[1] = 27;

// read device values from the host
std::cout << "sum: " << d_vec[0] + d_vec[1] << std::endl;
```
Containers

- Compatible with STL containers

```cpp
// list container on host
std::list<int> h_list;
h_list.push_back(13);
h_list.push_back(27);

// copy list to device vector
thrust::device_vector<int> d_vec(h_list.size());
thrust::copy(h_list.begin(), h_list.end(), d_vec.begin());

// alternative method using vector constructor
thrust::device_vector<int> d_vec2(h_list.begin(), h_list.end());
```
Namespaces

- Avoid name collisions

```cpp
// allocate host memory
thrust::host_vector<int> h_vec(10);

// call STL sort
std::sort(h_vec.begin(), h_vec.end());

// call Thrust sort
thrust::sort(h_vec.begin(), h_vec.end());

// for brevity
using namespace thrust;

// without namespace
int sum = reduce(h_vec.begin(), h_vec.end());
```
Algorithms

- Elementwise operations
  - for_each, transform, gather, scatter ...

- Reductions
  - reduce, inner_product, reduce_by_key ...

- Prefix Sums [scans]
  - inclusive_scan, inclusive_scan_by_key ...

- Sorting
  - sort, stable_sort, sort_by_key ...
Thrust Example: Sort

```c
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>

int main(void) {
    // generate 16M random numbers on the host
    thrust::host_vector<int> h_vec(1 << 24);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);

    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;

    // sort data on the device (805 Mkeys/sec on GeForce GTX 480)
    thrust::sort(d_vec.begin(), d_vec.end());

    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

    return 0;
}
```
Leveraging Parallel Primitives

- Test: sort 32M keys on each platform
  - Performance measured in millions of keys per second [higher is better]
- Conclusion: Use `sort` liberally, it’s highly optimized

<table>
<thead>
<tr>
<th>data type</th>
<th>std::sort</th>
<th>tbb::parallel_sort</th>
<th>thrust::sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>25.1</td>
<td>68.3</td>
<td>3532.2</td>
</tr>
<tr>
<td>short</td>
<td>15.1</td>
<td>46.8</td>
<td>1741.6</td>
</tr>
<tr>
<td>int</td>
<td>10.6</td>
<td>35.1</td>
<td>804.8</td>
</tr>
<tr>
<td>long</td>
<td>10.3</td>
<td>34.5</td>
<td>291.4</td>
</tr>
<tr>
<td>float</td>
<td>8.7</td>
<td>28.4</td>
<td>819.8</td>
</tr>
<tr>
<td>double</td>
<td>8.5</td>
<td>28.2</td>
<td>358.9</td>
</tr>
</tbody>
</table>
Input-Sensitive Optimizations

Sorting Rate (Mkey/s) vs. Key Bits

NVIDIA [N. Bell]→
# Maximum Value

```cpp
#include <thrust/device_vector.h>
#include <thrust/reduce.h>
#include <thrust/functional.h>
#include <iostream>

int main(void) {
    thrust::device_vector<float> X(3);


    float init = X[0];

    float result = thrust::reduce(X.begin(), X.end(),
                                   init,
                                   thrust::maximum<float>());

    std::cout << "maximum is " << result << "\n";

    return 0;
}
```
Algorithms

- Process one or more ranges

```c++
// copy values to device
device_vector<int> A(10);
device_vector<int> B(10);
device_vector<int> C(10);

// sort A in-place
sort(A.begin(), A.end());

// copy A -> B
copy(A.begin(), A.end(), B.begin());

// transform A + B -> C
transform(A.begin(), A.end(), B.begin(), C.begin(), plus<int>());
```
 Algorithms

- Standard operators

```c++
// allocate memory
device_vector<int> A(10);
device_vector<int> B(10);
device_vector<int> C(10);

// transform A + B -> C
transform(A.begin(), A.end(), B.begin(), C.begin(), plus<int>());

// transform A - B -> C
transform(A.begin(), A.end(), B.begin(), C.begin(), minus<int>());

// multiply reduction
int product = reduce(A.begin(), A.end(), 1, multiplies<int>());
```
Algorithms

- Standard data types

```cpp
// allocate device memory
device_vector<int> i_vec = ...
device_vector<float> f_vec = ...

// sum of integers
int i_sum = reduce(i_vec.begin(), i_vec.end());

// sum of floats
float f_sum = reduce(f_vec.begin(), f_vec.end());
```
struct negate_float2
{
    __host__ __device__
    float2 operator()(float2 a)
    {
        return make_float2(-a.x, -a.y);
    }
};

// declare storage
device_vector<float2> input = ... 
device_vector<float2> output = ... 

// create function object or 'functor'
negate_float2 func;

// negate vectors
transform(input.begin(), input.end(), output.begin(), func);
Custom Types & Operators

```cpp
// compare x component of two float2 structures
struct compare_float2
{
    __host__ __device__
    bool operator()(float2 a, float2 b)
    {
        return a.x < b.x;
    }
};

// declare storage
device_vector<float2> vec = ...;

// create comparison functor
compare_float2 comp;

// sort elements by x component
sort(vec.begin(), vec.end(), comp);
```
// return true if x is greater than threshold
struct is_greater_than
{
    int threshold;

    is_greater_than(int t) { threshold = t; }

    __host__ __device__
    bool operator()(int x) { return x > threshold; }
};

device_vector<int> vec = ...;

// create predicate functor (returns true for x > 10)
is_greater_than pred(10);

// count number of values > 10
int result = count_if(vec.begin(), vec.end(), pred);
Interoperability

- Convert iterators to raw pointers

```cpp
// allocate device vector
thrust::device_vector<int> d_vec(4);

// obtain raw pointer to device vector’s memory
int * ptr = thrust::raw_pointer_cast(&d_vec[0]);

// use ptr in a CUDA C kernel
my_kernel<<< N / 256, 256 >>>(N, ptr);

// use ptr in a CUDA API function
cudaMemcopyAsync(ptr, ... );
```
Interoperability

- Wrap raw pointers with `device_ptr`

```c
// raw pointer to device memory
int * raw_ptr;
cudaMalloc((void **) &raw_ptr, N * sizeof(int));

// wrap raw pointer with a device_ptr
thrust::device_ptr<int> dev_ptr(raw_ptr);

// use device_ptr in thrust algorithms
thrust::fill(dev_ptr, dev_ptr + N, (int) 0);

// access device memory through device_ptr
dev_ptr[0] = 1;

// free memory
cudaFree(raw_ptr);
```
# Algorithms, More Context…

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduce</td>
<td>Sum of a sequence</td>
</tr>
<tr>
<td>find</td>
<td>First position of a value in a sequence</td>
</tr>
<tr>
<td>mismatch</td>
<td>First position where two sequences differ</td>
</tr>
<tr>
<td>inner_product</td>
<td>Dot product of two sequences</td>
</tr>
<tr>
<td>equal</td>
<td>Whether two sequences are equal</td>
</tr>
<tr>
<td>min_element</td>
<td>Position of the smallest value</td>
</tr>
<tr>
<td>count</td>
<td>Number of instances of a value</td>
</tr>
<tr>
<td>is_sorted</td>
<td>Whether sequence is in sorted order</td>
</tr>
<tr>
<td>transform_reduce</td>
<td>Sum of transformed sequence</td>
</tr>
</tbody>
</table>
## General Transformations

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Code</th>
</tr>
</thead>
</table>
| Unary Transformation | `for (int i = 0; i < N; i++)
X[i] = f(A[i]);` |
| Binary Transformation | `for (int i = 0; i < N; i++)
X[i] = f(A[i],B[i]);` |
| Ternary Transformation | `for (int i = 0; i < N; i++)
X[i] = f(A[i],B[i],C[i]);` |
| General Transformation | `for (int i = 0; i < N; i++)
X[i] = f(A[i],B[i],C[i],...);` |

- Like the STL, **thrust** provides built-in support for unary and binary transformations.
- Transformations involving 3 or more input ranges must use a different approach.
General Transformations Preamble:
The Zipping Operation

- Multiple Distinct Sequences
- Unique Sequence of Tuples

NVIDIA [N. Bell]→
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/iterator/zip_iterator.h>
#include <iostream>

struct linear_combo {
  __host__ __device__
  float operator()(thrust::tuple<float,float,float> t) {
    float x, y, z;
    thrust::tie(x,y,z) = t;
    return 2.0f * x + 3.0f * y + 4.0f * z;
  }
};

int main(void) {
  thrust::device_vector<float> X(3), Y(3), Z(3);
  thrust::device_vector<float> U(3);


  thrust::transform
    (thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y.begin(), Z.begin())),
     thrust::make_zip_iterator(thrust::make_tuple(X.end(), Y.end(), Z.end())),
     U.begin(),
     linear_combo());

  for (size_t i = 0; i < Z.size(); i++)
    std::cout << "U[" << i << "] = " << U[i] << "\n";
  return 0;
}
Example: thrust::transform_reduce

```c++
#include <thrust/transform_reduce.h>
#include <thrust/device_vector.h>
#include <thrust/iterator/zip_iterator.h>
#include <iostream>

struct linear_combo {
    __host__ __device__ float operator()(thrust::tuple<float, float, float> t) {
        float x, y, z;
        thrust::tie(x, y, z) = t;
        return 2.0f * x + 3.0f * y + 4.0f * z;
    }
};

int main(void) {
    thrust::device_vector<float> X(3), Y(3), Z(3), U(3);


    thrust::plus<float> binary_op;
    float init = 0.f;

    float myResult = thrust::transform_reduce
        (thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y.begin(), Z.begin())),
         thrust::make_zip_iterator(thrust::make_tuple(X.end(), Y.end(), Z.end())),
         linear_combo(),
         init,
         binary_op);

    std::cout << myResult << std::endl;
    return 0;
}
```
thrust, Efficiency Issues
[fusing transformations]
Performance Considerations
[short detour: 1/3]

- Picture below shows key parameters
  - Peak flop rate
  - Max bandwidth

1030 GFLOP/s
[SinglePrecision]

SMs

144 GB/s

DRAM

Tesla C2050
Arithmetic Intensity
[short detour: 2/3]

Memory bound

Compute bound

FLOP/Byte

SAXPY

FFT

SGEMM

NVIDIA [N. Bell]→
### Arithmetic Intensity

[short detour: 3/3]

<table>
<thead>
<tr>
<th>Kernel</th>
<th>FLOP/Byte*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Addition</td>
<td>1 : 12</td>
</tr>
<tr>
<td>SAXPY</td>
<td>2 : 12</td>
</tr>
<tr>
<td>Ternary Transformation</td>
<td>5 : 20</td>
</tr>
<tr>
<td>Sum</td>
<td>1 : 4</td>
</tr>
<tr>
<td>Max Index</td>
<td>1 : 12</td>
</tr>
</tbody>
</table>

* excludes indexing overhead

<table>
<thead>
<tr>
<th>Hardware**</th>
<th>FLOP/Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeForce GTX 280</td>
<td>~7.0 : 1</td>
</tr>
<tr>
<td>GeForce GTX 480</td>
<td>~7.6 : 1</td>
</tr>
<tr>
<td>Tesla C870</td>
<td>~6.7 : 1</td>
</tr>
<tr>
<td>Tesla C1060</td>
<td>~9.1 : 1</td>
</tr>
<tr>
<td>Tesla C2050</td>
<td>~7.1 : 1</td>
</tr>
</tbody>
</table>

** lists the number of flop per byte of data to reach peak Flop/s rate

“Byte” refers to a Global Memory byte
Fusing Transformations

```
for (int i = 0; i < N; i++)
    U[i] = F(X[i], Y[i], Z[i]);

for (int i = 0; i < N; i++)
    V[i] = G(X[i], Y[i], Z[i]);
```

Loop Fusion

- One way to look at things…
  - Zipping: reorganizes **data** for **thrust** processing
  - Fusing: reorganizes **computation** for efficient **thrust** processing
typedef thrust::tuple<float, float> Tuple2;
typedef thrust::tuple<float, float, float> Tuple3;

struct linear_combo {
    __host__ __device__
    Tuple2 operator()(Tuple3 t) {
        float x, y, z; thrust::tie(x, y, z) = t;
        float u = 2.0f * x + 3.0f * y + 4.0f * z;
        float v = 1.0f * x + 2.0f * y + 3.0f * z;
        return Tuple2(u, v);
    }
};

int main(void) {
    thrust::device_vector<float> X(3), Y(3), Z(3);
    thrust::device_vector<float> U(3), V(3);


    thrust::transform
        (thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y.begin(), Z.begin())),
         thrust::make_zip_iterator(thrust::make_tuple(X.end(), Y.end(), Z.end())),
         thrust::make_zip_iterator(thrust::make_tuple(U.begin(), V.begin())),
         linear_combo());

    return 0;
}
Fusing Transformations

Original Implementation

- GPU
- 12 Bytes
- 4 Bytes
- 12 Bytes
- 4 Bytes

- DRAM

Optimized Implementation

- GPU
- 12 Bytes
- 8 Bytes

- DRAM

- Since the operation is completely memory bound the expected speedup is ~1.6x (=32/20)
Fusing Transformations

```c
for (int i = 0; i < N; i++)
    Y[i] = F(X[i]);

for (int i = 0; i < N; i++)
    sum += Y[i];
```

Loop Fusion

```c
for (int i = 0; i < N; i++)
    sum += F(X[i]);
```
Fusing Transformations

```cpp
#include <thrust/device_vector.h>
#include <thrust/transform_reduce.h>
#include <thrust/functional.h>
#include <iostream>

using namespace thrust::placeholders;

int main(void) {
    thrust::device_vector<float> X(3);


    float result = thrust::transform_reduce
        (X.begin(), X.end(),
         _1 * _1,
         0.0f,
         thrust::plus<float>())

    std::cout << "sum of squares is " << result << "\n";
    return 0;
}
```
Fusing Transformations

Original Implementation

- GPU
- 4 Bytes
- 4 Bytes
- 4 Bytes
- DRAM

Optimized Implementation

- GPU
- 4 Bytes
- DRAM

Try to answer this: how many times will we be able to run faster if we fuse?
typedef thrust::tuple<int, int> Tuple;

struct max_index {
    __host__ __device__
    Tuple operator()(Tuple a, Tuple b) {
        if (thrust::get<0>(a) > thrust::get<0>(b))
            return a;
        else
            return b;
    }
};

int main(void) {
    thrust::device_vector<int> X(3), Y(3);

    X[0] = 10; X[1] = 30; X[2] = 20; // values
    Y[0] = 0; Y[1] = 1; Y[2] = 2; // indices

    Tuple init(X[0], Y[0]);

    Tuple result = thrust::reduce
        (thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y.begin())),
         thrust::make_zip_iterator(thrust::make_tuple(X.end(), Y.end())),
         init,
         max_index());

    int value, index; thrust::tie(value, index) = result;

    std::cout << "maximum value is " << value << " at index " << index << "\n";

    return 0;
}
typedef thrust::tuple<int,int> Tuple;

struct max_index {
    __host__ __device__
    Tuple operator()(Tuple a, Tuple b) {
        if (thrust::get<0>(a) > thrust::get<0>(b))
            return a;
        else
            return b;
    }
};

int main(void) {
    thrust::device_vector<int> X(3);
    thrust::counting_iterator<int> Y(0);

    Tuple init(X[0],Y[0]);

    Tuple result = thrust::reduce
        (thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y)),
         thrust::make_zip_iterator(thrust::make_tuple(X.end(), Y + X.size())),
         init,
         max_index());

    int value, index; thrust::tie(value,index) = result;

    std::cout << "maximum value is " << value << " at index " << index << "\n";

    return 0;
}
Maximum Index (Optimized)

- Try to answer this: how many times will we be able to run faster if we fuse?
Good Speedups Compared to Multi-threaded CPU Execution

- CUDA 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz

NVIDIA [N. Bell]
thrust Wrap-Up

- Significant boost in productivity at the price of small performance penalty
  - No need to know of execution configuration, shared memory, etc.

- Key concepts
  - Functor
  - Zipping data
  - Fusing operations

- Why not always use thrust?
  - There is no “solve multibody dynamics” support in thrust.
    - I need to take care of this
  - Thrust provides support for primitives – up to us to use them as needed
thrust on Google Code

- Quick Start Guide
- Examples
- News
- Documentation
- Mailing List (thrust-users)
thrust in “GPU Computing Gems”

This chapter demonstrates how to leverage the Thrust parallel template library to implement high-performance applications with minimal programming effort. Based on the C++ Standard Template Library (STL), Thrust brings a familiar high-level interface to the realm of GPU Computing while remaining fully interoperable with the rest of the CUDA software ecosystem. Applications written with Thrust are concise, readable, and efficient.

26.1 MOTIVATION

With the introduction of CUDA C++, developers can harness the massive parallelism of the GPU through a standard programming language. CUDA allows developers to make fine-grained decisions about how computations are decomposed into parallel threads and executed on the device. The level of control offered by CUDA C++ (henceforth CUDA C) is an important feature: it facilitates the development of high-performance algorithms for a variety of computationally demanding tasks which (1) merit significant optimization and (2) profit from low-level control of the mapping onto hardware. For this class of computational tasks CUDA C is an excellent solution.

Thrust ([1]) solves a complementary set of problems, namely those that are (1) implemented efficiently without a detailed mapping of work onto the target architecture or those that (2) do not merit or simply will not receive significant optimization effort by the user. With Thrust, developers describe their computation using a collection of high-level algorithms and completely delegate the decision of how to implement the computation to the library. This abstract interface allows programmers to describe what to compute without placing any additional restrictions on how to carry out the computation. By capturing the programmer’s intent at a high level, Thrust has the discretion to make informed

PDF available at http://goo.gl/adj9S
Example, **thrust:** Processing Rainfall Data

Rain situation, end of first day, for a set of five observation stations. Results, summarized over a period of time, reported in the table below.

<table>
<thead>
<tr>
<th>day</th>
<th>[0 0 1 2 5 5 6 6 7 8 ... ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>site</td>
<td>[2 3 0 1 1 2 0 1 2 1 ... ]</td>
</tr>
<tr>
<td>measurement</td>
<td>[9 5 6 3 3 8 2 6 5 10 ... ]</td>
</tr>
</tbody>
</table>

Remarks:
1) Time series sorted by day
2) Measurements of zero are excluded from the time series
Example: Processing Rainfall Data

Given the data above, here’re some questions you might ask:

- Total rainfall at a given site
- Total rainfall between given days
- Total rainfall on each day
- Number of days with any rainfall
Total Rainfall at a Given Site

```cpp
struct one_site_measurement
{
    int siteOfInterest;

    one_site_measurement(int site) : siteOfInterest(site) {}

    __host__ __device__
    int operator()(thrust::tuple<int, int> t)
    {
        if (thrust::get<0>(t) == siteOfInterest)
            return thrust::get<1>(t);
        else
            return 0;
    }
};

template <typename Vector>
int compute_total_rainfall_at_one_site(int siteID, const Vector& site, const Vector& measurement)
{
    return thrust::transform_reduce
        (thrust::make_zip_iterator(thrust::make_tuple(site.begin(), measurement.begin())),
        thrust::make_zip_iterator(thrust::make_tuple(site.end(), measurement.end())),
        one_site_measurement(siteID),
        0,
        thrust::plus<int>())
;
Total Rainfall Between Given Days

```cpp
template <typename Vector>
int compute_total_rainfall_between_days(int first_day, int last_day, 
   const Vector& day, const Vector& measurement)
{
    int first = thrust::lower_bound(day.begin(), day.end(), first_day) - day.begin();
    int last = thrust::upper_bound(day.begin(), day.end(), last_day) - day.begin();

    return thrust::reduce(measurement.begin() + first, measurement.begin() + last);
}
```

For this to fly, you’ll need to include several header files (not all for the code snippet above)

```cpp
#include <thrust/device_vector.h>
#include <thrust/binary_search.h>
#include <thrust/transform.h>
#include <thrust/iterator/zip_iterator.h>
#include <iostream>
```
Number of Days with Any Rainfall

template <typename Vector>
int compute_number_of_days_with_rainfall(const Vector& day)
{
    return thrust::inner_product(day.begin(), day.end() - 1,
                                day.begin() + 1,
                                0,
                                thrust::plus<int>(),
                                thrust::not_equal_to<int>()) + 1;
}

day [0 = 0 ≠ 1 ≠ 2 ≠ 5 = 5 ≠ 6 = 6 ≠ 7 ≠ 8 ... ]

1+ [0 = 1 = 1 = 1 = 0 + 1 + 0 + 1 + 1 + 1 ... ]
template <typename Vector>
void compute_total_rainfall_per_day(const Vector& day, const Vector& measurement,
Vector& day_output, Vector& measurement_output)
{
    size_t N = compute_number_of_days_with_rainfall(day); //see previous slide

day_output.resize(N);
measurement_output.resize(N);

thrust::reduce_by_key(day.begin(), day.end(),
    measurement.begin(),
    day_output.begin(),
    measurement_output.begin());
}
3 Ways to Accelerate on GPU

Application

Libraries
Directives
Programming Languages

Easiest Approach

Maximum Performance

Direction of increased performance (and effort)

NVIDIA [C. Woolley] →
Directives…
OpenACC

- Seeks to become:
  - A standard for directives-based Parallel Programming
  - Provide portability across hardware platforms and compiler vendors
- Promoted by NVIDIA, Cray, CAPS, PGI
OpenACC Specification

- Hardware agnostic and platform independent (CPU only, different GPUs)

- OpenACC is an open standard for directives based computing

- Announced at SC11 [November 2011]

- Caps, Cray, and PGI to ship OpenACC Compilers beginning Q1 2012

- A version 2.0 draft is out for comments, idea is to expand OpenMP with features that enables one to farm out work to the GPUs
The OpenACC Idea

- Host code computes an approximation for $\pi$:

```c++
#include <iostream>
#include <math.h>
using namespace std;

int main( int argc, char *argv[] )
{
    const double PI25DT = 3.141592653589793;

    const int n=1000000;
    double h = 1.0 / (double) n;
    double sum = 0.0;

    for( int i=0; i<=n; i++ ) {
        double x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));
    }
    double mypi = h * sum;

    cout << "Approx. value: " << mypi << endl;
    cout << "Error: " << fabs(mypi-PI25DT) << endl;
    return 0;
}
```
The OpenACC Idea

- Code computes an approximation for $\pi$ [might use multi-core or GPU]

```cpp
#include <iostream>
#include <math.h>
using namespace std;

int main( int argc, char *argv[] )
{
    const double PI25DT = 3.141592653589793238462643;

    const int n=1000000;
    double h = 1.0 / (double) n;
    double sum = 0.0;
    // # pragma acc region for
    for( int i=0; i<=n; i++ ) {
        double x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));
    }
    double mypi = h * sum;

    cout << "Approx. value: " << mypi << endl;
    cout << "Error: " << fabs(mypi-PI25DT) << endl;
    return 0;
}
```

Add one line of code (a directive): provides a hint to the compiler about opportunity for parallelism
OpenACC Target Audience

- OpenACC targets three classes of users:
  - Users with parallel codes, ideally with some OpenMP experience, but less GPU knowledge
  - Users with serial codes looking for portable parallel performance with and without GPUs
  - “Hardcore” GPU programmers with existing CUDA ports
OpenACC Perceived Benefits

- Code easier to maintain
- Helps with legacy code bases
- Portable:
  - Can run same code CPU/GPU
- Very much like OpenMP
- Only small performance loss
  - Cray goal: 90% of CUDA
CUDA: Getting More Info…

- More information on this
  

- CUDA Tools and Ecosystem
  
  - Described in detail on NVIDIA Developer Zone
    
First question you need to ask: is there a GPU library that I can use?

If no library available, just go ahead and do it!
- Avoid “analysis paralysis”
- Adopt a “crawl – walk – run” approach
- Go back and profile/optimize once you have something working
- To “have something working” debug like a pro (cuda-gdb and cuda-memchk)
Libraries…
CUDA Libraries

- Math, Numerics, Statistics
- Dense & Sparse Linear Algebra
- Algorithms (sort, etc.)
- Image Processing
- Signal Processing
- Finance

- In addition to these widely adopted libraries, several less established ones available in the community

cuBLAS: Dense linear algebra on GPUs

- Complete BLAS implementation plus useful extensions
  - Supports all 152 standard routines for single, double, complex, and double complex
  - Levels 1, 2, and 3 BLAS

- New features in CUDA 4.1:
  - New batched GEMM API provides >4x speedup over MKL
  - Useful for batches of 100+ small matrices from 4x4 to 128x128
  - 5%-10% performance improvement to large GEMMs
Speedups Compared to Multi-threaded CPU Execution

- CUDA 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz
cuSPARSE: Sparse linear algebra routines

- Sparse matrix-vector multiplication & triangular solve
  - APIs optimized for iterative methods

- New features in 4.1:
  - Tri-diagonal solver with speedups up to 10x over Intel MKL
  - ELL-HYB format offers 2x faster matrix-vector multiplication

\[
\begin{bmatrix}
y_1 \\
y_2 \\
y_3 \\
y_4 \\
\end{bmatrix} = \alpha \begin{bmatrix}
2 & -1 \\
4 & -1 & 2 \\
5 & 9 & 1 \\
-1 & 8 & 3 \\
\end{bmatrix} \begin{bmatrix}
y \\\ny \\\ny \\\ny \\
\end{bmatrix} + \beta \begin{bmatrix}
2 \\
0 \\
1 \\
-1 \\
\end{bmatrix}
\]
Good Speedups Compared to Multi-threaded CPU Execution

Sparse matrix test cases on following slides come from:
1. The University of Florida Sparse Matrix Collection
   http://www.cise.ufl.edu/research/sparse/matrices/
   http://www.nvidia.com/object/nvidia_research_pub_001.html

- CUDA 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz
cuFFT: Multi-dimensional FFTs

- Algorithms based on Cooley-Tukey and Bluestein
- Simple interface, similar to FFTW
- Streamed asynchronous execution
- 1D, 2D, 3D transforms of complex and real data
- Double precision (DP) transforms
- 1D transform sizes up to 128 million elements
- Batch execution for doing multiple transforms
- In-place and out-of-place transforms

\[ F(x) = \sum_{n=0}^{N-1} f(n)e^{-j2\pi(x^n/N)} \]

\[ f(n) = \frac{1}{N} \sum_{n=0}^{N-1} F(x)e^{j2\pi(x^n/N)} \]
Speedups Compared to Multi-Threaded CPU Execution

- CUDA 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz

NVIDIA [C. Woolley]
cuRAND: Random Number Generation

- Pseudo- and Quasi-RNGs
  - Supports several output distributions
  - Statistical test results reported in documentation

- New RNGs in CUDA 4.1:
  - MRG32k3a RNG
  - MTGP11213 Mersenne Twister RNG

NVIDIA [C. Woolley]→
NPP: NVIDIA Performance Primitives

- Arithmetic, Logic, Conversions, Filters, Statistics, Signal Processing, etc.
- This is where GPU computing shines
- 1,000+ new image primitives in 4.1
Development, Debugging, and Deployment Tools

[Rounding Up the CUDA Ecosystem]
Programming Languages & APIs

- HMPP Compiler
- Python for CUDA
- NVIDIA C Compiler
- CUDA Fortran
- OpenCL
- OpenGL
- PGI Accelerator
- Microsoft DirectX 11
- Microsoft AMP C/C++
Debugging Tools

- NVIDIA Parallel Nsight for Visual Studio
- NVIDIA CUDA-MEMCHECK for Linux & Mac
- Allinea DDT with CUDA Distributed Debugging Tool
- NVIDIA CUDA-GDB for Linux & Mac
- TotalView for CUDA for Linux Clusters
Performance Analysis Tools

- NVIDIA Parallel Nsight for Visual Studio
- Vampir Trace Collector
- TAU Performance System
- Performance API Library
- NVIDIA Visual Profiler for Linux & Mac
- Under Development
MPI & CUDA Support

Platform MPI
- Announced beta at SC2011
- As of OFED 1.5.2

GPUDirect™
- Announced pre-release at SC2011
- Platform MPI
  - Announced beta at SC2011

InfiniBand

NVIDIA [C. Woolley]→
Cluster Management & Job Scheduling

- Platform Computing
  - LSF, HPC, Cluster Manager

- Bright Computing
  - Bright Cluster Manager

- PBS Works
  - PBS Professional

- Ganglia
  - NVML Plugin for GPUs

- Adaptive Computing

- Rocks+ MOAB

- Univa
  - Univa Grid Engine
Multicore Parallel Computing with OpenMP
OpenMP and Symmetric Multi-Processing

- Threads have access to a large pool of shared memory
- Threads can have private data
  - Not accessible by other threads
- Data transfer/access is transparent to the programmer
- Synchronization is implicit but can be made explicit as well
Quick Look at Hardware

- Intel Haswell
  - Released in June 2013
  - 22 nm technology
  - Transistor budget: 1.4 billions
    - Tri-gate, 3D transistors
  - Typically comes in four cores
  - Has an integrated GPU
  - Deep pipeline – 16 stages
  - Sophisticated infrastructure for ILP acceleration
  - Superscalar
  - Supports HTT (hyper-threading technology)

Good source of information for these slides: [http://www.realworldtech.com/](http://www.realworldtech.com/)
Quick Look at Hardware

- Actual layout of the chip

- Schematic of the chip organization
  - LLC: last level cache (L3)
  - Three clocks:
    - A core’s clock ticks at 2.7 to 3.0 GHz but adjustable up to 3.7-3.9 GHz
    - Graphics processor ticking at 400 MHz but adjustable up to 1.3 GHz
    - Ring bus and the shared L3 cache - a frequency that is close to but not necessarily identical to that of the cores
Caches

- **Data:**
  - L1 – 32 KB per core
  - L2 – 512 KB or 1024 KB per core
  - L3 – 8 MB per CPU

- **Instruction:**
  - L0 – room for about 1500 microoperations (uops) per core
    - See H/S primer, online
  - L1 – 32 KB per core

- **Cache is a black hole for transistors**
  - Example: 8 MB of L3 translates into:
    - \(8 \times 1024 \times 1024 \times 8 \text{ (bits)} \times 6 \text{ (transistors per bit, SRAM)} = 402 \text{ million transistors out of } 1.4 \text{ billions}\)

- **Caches are *very* important for good performance**
Haswell Microarchitecture
[30,000 Feet]

- Microarchitecture components:
  - Instruction pre-fetch support (purple)
  - Instruction decoding support (orange)
    - CISC into uops
      - Turning CISC to RISC
  - Instruction Scheduling support (yellowish)
  - Instruction execution
    - Arithmetic (blue)
    - Memory related (green)

- More details - see primer posted online:
  [http://www.realworldtech.com](http://www.realworldtech.com)
  [http://sbel.wisc.edu/Courses/ME964/Literature/primerHW-SWinterface.pdf](http://sbel.wisc.edu/Courses/ME964/Literature/primerHW-SWinterface.pdf)
Moving from HW to SW
Acknowledgements

- Majority of slides used for discussing OpenMP issues are from Intel’s library of presentations for promoting OpenMP
  - Slides used herein with permission

- Credit given where due: IOMPP
  - IOMPP stands for “Intel OpenMP Presentation”
Data vs. Task Parallelism

- **Data parallelism**
  - You have a large amount of data elements and each data element needs to be processed to produce a result
  - When this processing can be done in parallel, we have data parallelism
  - Example:
    - Adding two long arrays of doubles to produce yet another array of doubles

- **Task parallelism**
  - You have a collection of tasks that need to be completed
  - If these tasks can be performed in parallel you are faced with a task parallel job
  - Examples:
    - Reading the newspaper, whistling, and scratching your back
    - The simultaneous breathing of your lungs, beating of your heart, liver function, controlling the swallowing, etc.
Objectives

- Understand OpenMP at the level where you can
  - Implement data parallelism
  - Implement task parallelism
Work Plan: What We’ll Cover

- What is OpenMP?
  Parallel regions
  Work sharing
  Data environment
  Synchronization

- Advanced topics
OpenMP: Target Hardware

- **CUDA**: targeted parallelism on the GPU

- **OpenMP**: targets parallelism on SMP architectures
  - Handy when
    - You have a machine that has 64 cores
    - You have a large amount of shared memory, say 128GB

- **MPI**: targeted parallelism on a cluster (distributed computing)
  - Note that MPI implementation can handle transparently an SMP architecture such as a workstation with two hexcore CPUs that draw on a good amount of shared memory
OpenMP:
What’s Reasonable to Expect

- If you have 64 cores available to you, it is *highly* unlikely to get a speedup of more than 64 (superlinear)

- Recall the trick that helped the GPU hide latency
  - Overcommitting the SMs and hiding memory access latency with warp execution

- This mechanism of hiding latency by overcommitment does not *explicitly* exist for parallel computing under OpenMP beyond what’s offered by HTT
  - It exists implicitly, under the hood, through ILP support
OpenMP: What Is It?

- Portable, shared-memory threading API
  - Fortran, C, and C++
  - Multi-vendor support for both Linux and Windows

- Standardizes task & loop-level parallelism
- Very good at coarse-grained parallelism
- Combines serial and parallel code in single source
- Standardizes ~ 20 years of compiler-directed threading experience

- Current spec is OpenMP 4.0
  - Released in 2013
  - http://www.openmp.org
  - More than 300 Pages
OpenMP Programming Model

- **Master thread** spawns a team of threads as needed
  - Managed transparently on your behalf
  - It relies on low-level thread fork/join methodology to implement parallelism
    - The developer is spared the details

- Parallelism is added incrementally: that is, the sequential program evolves into a parallel program
OpenMP: Library Support

- Runtime environment routines:
  - Modify/check the number of threads
    
    ```
    omp_[set|get]_num_threads()
    omp_get_thread_num()
    omp_get_max_threads()
    ```

  - Are we in a parallel region?
    
    ```
    omp_in_parallel()
    ```

  - How many processors in the system?
    
    ```
    omp_get_num_procs()
    ```

  - Explicit locks
    
    ```
    omp_[set|unset]_lock()
    ```

  - Many more...

https://computing.llnl.gov/tutorials/openMP/
A Few Syntax Details to Get Started

- Picking up the API - header file in C, or Fortran 90 module
  ```
  #include "omp.h"
  use omp_lib
  ```

- Most OpenMP constructs are compiler directives or pragmas
  - For C and C++, the pragmas take the form:
    ```
    #pragma omp construct [clause [clause]...]
    ```
  - For Fortran, the directives take one of the forms:
    ```
    C$OMP construct [clause [clause]...]  
    !$OMP construct [clause [clause]...]  
    *$OMP construct [clause [clause]...]  
    ```
Why Compiler Directive and/or Pragmas?

- One of OpenMP’s design principles: the same code, with no modifications, can run either on an one core machine or a multiple core machine

- Therefore, you have to “hide” all the compiler directives behind Comments and/or Pragmas

- These directives picked up by the compiler only if you instruct it to compile in OpenMP mode
  - Example: Visual Studio – you have to have the /openmp flag on in order to compile OpenMP code
  - Also need to indicate that you want to use the OpenMP API by having the right header included: #include <omp.h>

Step 1: Go here

Step 2: Select /openmp
OpenMP, Compiling Using the Command Line

- Method depends on compiler

- **GCC:**
  
  ```bash
  $ g++ -o integrate_omp integrate_omp.c -fopenmp
  ```

- **ICC:**
  
  ```bash
  $ icc -o integrate_omp integrate_omp.c -openmp
  ```

- **MSVC (not in the express edition):**
  
  ```bash
  $ cl /openmp integrate_omp.c
  ```
Controlling the number of threads
- The default number of threads that a program uses when it runs is the number of processors on the machine

For the C Shell:  
```
setenv OMP_NUM_THREADS number
```

For the Bash Shell:  
```
export OMP_NUM_THREADS=number
```

Timing:
```
#include <omp.h>
stime = omp_get_wtime();
mylongfunction();
etime = omp_get_wtime();
total=etime-stime;
```
Work Plan

- What is OpenMP?
  - Parallel regions
  - Work sharing
  - Data environment
  - Synchronization

- Advanced topics
Parallel Region & Structured Blocks (C/C++)

- Most OpenMP constructs apply to structured blocks
  - **structured block**, definition: a block with one point of entry at the top and one point of exit at the bottom
  - The only “branches” allowed are exit() function calls

A structured block

```
#pragma omp parallel
{
   int id = omp_get_thread_num();
   more: res[id] = do_big_job (id);
   if ( not_conv(res[id]) ) goto more;
} printf("All done\n");
```

Not a structured block

```
if (go_now()) goto more;
#pragma omp parallel
{
   int id = omp_get_thread_num();
   more: res[id] = do_big_job(id);
   if ( conv (res[id]) ) goto done;
   goto more;
} done: if (!really_done()) goto more;
```

There is an implicit barrier at the right “}” curly brace and that’s the point at which the other worker threads complete execution and either go to sleep or spin or otherwise idle.
```c
#include <stdio.h>
#include <omp.h>

int main() {
#pragma omp parallel
{
    int myId = omp_get_thread_num();
    int nThreads = omp_get_num_threads();

    printf("Hello World. I'm thread %d out of %d.
", myId, nThreads);
    for( int i=0; i<2 ;i++)
        printf("Iter:%d\n",i);
}
    printf("All done here...\n");
}
```

Example: Hello World

Here’s my laptop:
Intel Core i5-3210M @ 2.50GHz 3 MB L3 Cache,
TDP 35 Watts, Two-Core Four-Thread Processors
One of the key tenets of OpenMP is that of data independence across parallel jobs.

Specifically, when distributing work among parallel threads it is assumed that there is no data dependency.

Since you place the `omp parallel` directive around some code, it is your responsibility to make sure that data dependency is ruled out.

Compilers are not smart enough and sometimes and they can’t identify data dependency between what might look as independent parallel jobs.
Work Plan

● What is OpenMP?
  Parallel regions
  **Work sharing**
  Data environment
  Synchronization

● Advanced topics
Work Sharing

- **Work sharing** is the general term used in OpenMP to describe distribution of work across threads.

- Three primary avenues for work sharing in OpenMP:
  - "omp for" construct
  - "omp sections" construct
  - "omp task" construct

Each of them automatically divides work among threads.
“omp for” construct

```
// assume N=12
#pragma omp parallel
#pragma omp for
    for(i = 1, i < N+1, i++)
        c[i] = a[i] + b[i];
```

- Threads are assigned an independent set of iterations
- Threads must wait at the end of work-sharing construct
Combining Constructs

- These two code segments are equivalent

```c
#pragma omp parallel
{
    #pragma omp for
    for (int i=0; i< MAX; i++){
        res[i] = huge();
    }
}
```

```c
#pragma omp parallel for
for (int i=0; i< MAX; i++){
    res[i] = huge();
}
```
The Private Clause

- Reproduces the variable for each task
  - By declaring a variable as being private it means that each thread will have a private copy of that variable
    - The value that thread 1 stores in x is different than the value that thread 2 stores in the variable x
  - Variables are un-initialized; C++ object is default constructed

```c
void* work(float* c, int N) {
    float x, y;
    int i;
    #pragma omp parallel for private(x,y)
    for(i=0; i<N; i++) {
        x = a[i]; y = b[i];
        c[i] = x + y;
    }
}
```
The schedule Clause

- The `schedule` clause affects how loop iterations are mapped onto threads

**schedule(static [,chunk])**
- Blocks of iterations of size “chunk” assigned to each thread
- Round robin distribution
- Low overhead, may cause load imbalance

**schedule(dynamic[,chunk])**
- Threads grab “chunk” iterations
- When done with iterations, thread requests next set
- Higher threading overhead, can reduce load imbalance

**schedule(guided[,chunk])**
- Dynamic schedule starting with large block
- Size of the blocks shrink; no smaller than “chunk”
schedule Clause Example

```
#pragma omp parallel for schedule (static, 8)
  for( int i = start; i <= end; i += 2 )
  {
    if ( TestForPrime(i) )  gPrimesFound++;
  }
```

- Iterations are divided into chunks of 8
- If start = 3, then first chunk is

\[ i = \{ 3, 5, 7, 9, 11, 13, 15, 17 \} \]
Example, Static Scheduling

- What is the default value of the chunk size?
  - Depends on the underlying hardware
  - Can figure out writing a short piece of code

```c
#include <stdio.h>
#include <omp.h>

int main() {
  #pragma omp parallel for schedule (static)
  for (int i = 0; i <= 14; i++){
    printf("I'm thread %d working on iteration %d\n", omp_get_thread_num(), i);
  }
  printf("All done here...\n");
}
```
Work Plan

- What is OpenMP?
  Parallel regions
  Work sharing – Parallel Sections
  Data environment
  Synchronization

- Advanced topics
Function Level Parallelism

```c
a = alice();
b = bob();
s = boss(a, b);
k = kate();
printf("%6.2f\n", bigboss(s,k));
```

alice, bob, and kate can be computed in parallel
omp sections

- `#pragma omp sections`
- Must be inside a parallel region
- Precedes a code block containing $N$ sub-blocks of code that may be executed concurrently by $N$ threads
- Encompasses each `omp section`, see below

- `#pragma omp section`
- Precedes each sub-block of code within the encompassing block described above
- Enclosed program segments are distributed for parallel execution among available threads
#pragma omp parallel sections
{
    #pragma omp section
        double a = alice();
    #pragma omp section
        double b = bob();
    #pragma omp section
        double k = kate();
}

double s = boss(a, b);
printf("%6.2fn", bigboss(s, k));
Advantage of Parallel Sections

- Independent sections of code can execute concurrently → reduces execution time

```c
#pragma omp parallel sections
{
#pragma omp section
    phase1();
#pragma omp section
    phase2();
#pragma omp section
    phase3();
}
```

Time (Execution Flow)

Serial

Parallel

The pink and green tasks are executed at no additional time-penalty in the shadow of the blue task.
```
#include <stdio.h>
#include <omp.h>

int main() {
    printf("Start with 2 procs only... \n\n");
    #pragma omp parallel sections num_threads(2)
    {
        #pragma omp section
        {
            printf("Start work 1\n");
            double startTime = omp_get_wtime();
            while( (omp_get_wtime() - startTime) < 2.0);
            printf("Finish work 1\n");
        }
        #pragma omp section
        {
            printf("Start work 2\n");
            double startTime = omp_get_wtime();
            while( (omp_get_wtime() - startTime) < 2.0);
            printf("Finish work 2\n");
        }
        #pragma omp section
        {
            printf("Start work 3\n");
            double startTime = omp_get_wtime();
            while( (omp_get_wtime() - startTime) < 2.0);
            printf("Finish work 3\n");
        }
    }
    return 0;
}
```
sections, Example: 2 threads
```c
#include <stdio.h>
#include <omp.h>

int main() {
    printf("Start with 4 procs\n");
    #pragma omp parallel sections num_threads(4)
    {
        #pragma omp section
        {
            printf("Start work 1\n");
            double startTime = omp_get_wtime();
            while( (omp_get_wtime() - startTime) < 2.0);
            printf("Finish work 1\n");
        }
        #pragma omp section
        {
            printf("Start work 2\n");
            double startTime = omp_get_wtime();
            while( (omp_get_wtime() - startTime) < 6.0);
            printf("Finish work 2\n");
        }
        #pragma omp section
        {
            printf("Start work 3\n");
            double startTime = omp_get_wtime();
            while( (omp_get_wtime() - startTime) < 2.0);
            printf("Finish work 3\n");
        }
    }
    return 0;
    }
```
sections, Example: 4 threads