Before We Get Started…

- **Last time**
  - Debugging CUDA applications
    - `cuda-gdb`
    - `cuda-memcheck`

- **Today**
  - Profiling CUDA code to improve performance
  - Examples: debugging and profiling of 1D stencil code

- **Other issues**
  - HW5 due tonight at 11:59 PM
  - HW6 posted online, available in Mercurial as well (due March 8)
  - Submit a half page proposal (due March 8) for your Midterm Project
    - It’ll have to do with solving a banded linear system unless you want to tie it into your research
Premature Optimization is the Root of All Evil. Yet,…

“Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.”

Donald Knuth
In “Structured Programming With Go To Statements”
Computing Surveys, Vol. 6, No. 4, December 1974
Available on class website.
Regarding Code Optimization…

- In 99% of the cases, “Code Optimization” is not about writing 40 lines of convoluted code to save two additions and one multiplication at the price of no able human being being able understand what you did there [see, I just tried to optimize this statement]

- For all purposes, especially when it comes to GPU computing, you can basically forget about the math overhead (+, -, *, /, reciprocal, square root, sin, cos, etc.)
  - Pretend they don’t exist

- Focus on the operands: what you are left with once you get rid of the math

- Example:
  - Suppose you have this:
    ```
    >> c[i] = mypi*sin(a[i])+2.f ;
    ```
  - Then, concentrate on this:
    - Where are c[i], mypi, a[i] coming from?
    - Maybe also, “Can I use Single Precision or should I use Double Precision?”
Regarding Code Optimization…

[Cntd.]

- Why you shouldn’t probably worry about math
  - One global memory transaction requires 400-600 cycles

- Math operations (1.3 architecture, C1060):
  - 4 clock cycles for an integer or single-precision floating-point arithmetic instruction
  - 16 clock cycles for a single-precision floating-point transcendental instruction
  - 2.0 architecture is even better
Example: 1 Tflops GPU needs a lot of data to reach peak rate

- Assume that you want to add *different* numbers and reach 1 Tflops: 1E12 ops/second

- You need to feed 2E12 operands per second…

- If each number is stored using 4 bytes (float), then you need to fetch 2E12*4 bytes in a second. This is 8E12 B/s, which is 8 TB/s…

- The memory bandwidth on the GPU is in the neighborhood of 0.15 TB/s, about 50 times less than what you need (and you haven’t taken into account that you probably want to send back the outcome of the operation that you carry out)
GPU Computing: Putting Things in Perspective [Cntd.]

- Another example: quick back-of-the-envelope computation to illustrate the crunching number power of a modern GPU
  - Suppose it takes 4 microseconds (4E-6) to launch a kernel (more about this later…)
  - Suppose you use a 1 Tflops (1E12) Fermi-type GPU to add (in 4 cycles) floats
  - Then, to break even with the amount of time it took you to invoke execution on the GPU in the first place you’d have to carry out about 1 million floating point ops on the GPU
    - [if everything was in registers and basically the only thing you did was crunch numbers]
Important Point

- In GPU computing, memory transactions are perhaps most relevant in determining the overall efficiency (performance) of your code.

<table>
<thead>
<tr>
<th>Memory Space</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register memory</td>
<td>≈ 8,000 GB/s</td>
</tr>
<tr>
<td>Shared memory</td>
<td>≈ 1,600 GB/s</td>
</tr>
<tr>
<td>Global memory</td>
<td>≈ 177 GB/s</td>
</tr>
<tr>
<td>Mapped memory</td>
<td>≈ 8 GB/s</td>
</tr>
</tbody>
</table>

Source: Rob Farber
“CUDA Application Design and Development”
Next, the discussion focuses on tools you can use to find that 3% of the code worth optimizing...
Code Timing/Profiling

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

- Advanced approach: use NVIDIA’s \texttt{nvvp} Visual Profiler
  - Visualize CPU and GPU activity
  - Identify optimization opportunities
  - Allows for automated analysis
  - \texttt{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]:

```bash
>> 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 fffff6c689a404a8
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=[_Z14applyStencil1DiiPKfFpS1_ ] gputime=[ 47332.9 ] cputime=[ 8.469 ] occupancy=[0.67]
method=[memcpyDtoH ] gputime=[ 3535.648 ] cputime=[ 4555.577 ]
```
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 fffffc689a404a8
method,gputime,cputime,occupancy
method=[ memcpHyToD ] gputime=[ 1001.952 ] cputime=[ 1197.000 ]
method=[ __Zl4applyStencil1DiiPKfPfS1__ ] gputime=[ 166.944 ] cputime=[ 13.000 ] occupancy=[1.0]
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
# CUDADEVICE 0 GeForce GT 130M
# TIMESTAMPFACTOR 12764ee9b183e71e
method,gputime,cputime,occupancy
method=[ memcpHyToD ] gputime=[ 1815.424 ] cputime=[ 2787.856 ]
method=[ __Zl4applyStencil1DiiPKfPfS1__ ] gputime=[ 4733.9 ] cputime=[ 8.469 ] occupancy=[0.67]
method=[ memcpyDtoH ] gputime=[ 3535.648 ] cputime=[ 4555.577 ]
```
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 ]
```

```
> 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 ]
```
nvvp: NVIDIA Visual Profiler

- Available on Euler

- Provides a nice GUI and broad but detailed information regarding your run

- Compared to the “Lazy Man’s approach”, it represents one step up in terms of amount of information and level of detail/resolution

- Many bells & whistles, covering here the basics using 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

\[ \text{in} \rightarrow f \rightarrow \text{out} \]

(radius = 3)

\[ \Rightarrow \text{CPU Thread} \]

NVIDIA [S. Satoor]
Serial Algorithm

\[ \implies = \text{CPU Thread} \]

\[ \text{(radius} = 3) \]

\[ \text{in} \quad \ldots \quad \text{\ldots} \]

\[ \text{out} \quad \ldots \quad \text{\ldots} \]

Repeat for each element
### Serial Implementation

```c
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);
    }
}

Serial Implementation

For each element...

Weighted mean over radius

NVIDIA [S. Satoor]→
```c
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);
    }
}
```

### Serial Implementation

<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

notin

out

\[ \Rightarrow = \text{Thread} \]
__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);
    }
}

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 applyStencill1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    int i = sIdx + blockIdx.x*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
__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);
    }
}
```

Allocate GPU memory

```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;  cudaMemcpy(d_weights, weights, wsize, cudaMemcpyHostToDevice);
    float *d_in;       cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    float *d_out;      cudaMemcpy(d_out, out, 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 memory

```c
Allocate GPU memory
```

The Parallel Implementation

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);
    }
}
```
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);
    }
}
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);
}
```

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

<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>

```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);
    }
}
```
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
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

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

Each analysis has link to Best Practices documentation.

LowMemcpy 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.

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 cudaHostAlloc() 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.

Parent topic: Data Transfer Between Host and Device
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);
    ...
}

CPU allocations use pinned memory to enable fast memcpy
No other changes
Pinned CPU Memory Result

![Image of NVIDIA GPU Profiler output showing pinned memory usage and performance metrics.](image)
Pinned CPU Memory Result

<table>
<thead>
<tr>
<th>Device</th>
<th>Algorithm</th>
<th>MElements/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
Application Optimization Process

[Revisited]

- Identify Optimization Opportunities
  - 1D stencil algorithm

- Parallelize with CUDA, confirm functional correctness
  - Debugger
  - Memory Checker

- Optimize
  - Profiler (pinned memory)
Application Optimization Process

[Revisited]

- Identify Optimization Opportunities
  - 1D stencil algorithm

- Parallelize with CUDA, confirm functional correctness
  - Debugger
  - Memory Checker

Optimize
  - Profiler (pinned memory)
- Advanced optimization
  - Larger time investment
  - Potential for larger speedup
Data Partitioning Example

Partition data into TWO chunks
Data Partitioning Example

chunk 1

memcpy
compute

chunk 2

memcpy

in

out

memcpy

NVIDIA [S. Satoor]→
Data Partitioning Example
Overlapped Compute/Memcpy

[problem broken into 16 chunks]
Overlapped Compute/Memcpy

Compute time completely “hidden”

Exploit dual memcpy engines
# Overlapped Compute/Memcpy

<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>
<tr>
<td>Tesla C2075</td>
<td>Overlap</td>
<td>935</td>
<td>7.2x</td>
</tr>
</tbody>
</table>

ME964: Use of multiple streams covered in 10 days
Application Optimization Process

[Revisited]

- Identify Optimization Opportunities
  - 1D stencil algorithm

- Parallelize with CUDA, confirm functional correctness
  - Debugger
  - Memory Checker

- Optimize
  - Profiler (pinned memory)
  - Profiler (overlap memcpy and compute)
Iterative Optimization

- Identify Optimization Opportunities
- Parallelize
- Optimize
Optimization Summary

[Looking back at this journey…]

- Initial CUDA parallelization
  - Expeditious, kernel is almost word-for-word replica of sequential code
  - 2.2x speedup

- Optimize memory throughput
  - Expeditious, need to know about pinned memory
  - 4.3x speedup

- Overlap compute and data movement
  - More involved, need to know about the inner works of CUDA
  - Problem should be large enough to justify mem-transfer/execution
  - 7.2x speedup
Take Home Message...

- Regard CUDA as a way to accelerate the compute-intensive parts of your application

- Visual profiler (nvvp) helps in performance analysis and optimization
Example: Debugging & Profiling

[1DStencil Code]

```c
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);
    float *cuda_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);
    applyStencil1D_SEQ(RADIUS, N-RADIUS, weights, in, out);
    cudaMemcpy(cuda_out, d_out, size, cudaMemcpyDeviceToHost);

    int nDiffs = checkResults(cuda_out, out, N);
    nDiffs==0? std::cout<<"Looks good.\n": std::cout<<"Doesn't look good: "<< nDiffs << " differences\n";

    //free resources
    free(weights); free(in); free(out); free(cuda_out);
    cudaFree(d_weights); cudaFree(d_in); cudaFree(d_out);
    return 0;
}
```
Example: Debugging & Profiling

[1DStencil Code: Supporting Cast]

```c
void initializeArray(float* arr, int nElements) {
    const int myMinNumber = -5;
    const int myMaxNumber = 5;
    srand(time(NULL));
    for (int i=0; i<nElements; i++)
        arr[i] = (float)(rand() % (myMaxNumber - myMinNumber + 1) + myMinNumber);
}

void initializeWeights(float* weights, int rad) {
    // for now hardcoded for RADIUS=3
    weights[0] = 0.50f;
    weights[1] = 0.75f;
    weights[2] = 1.25f;
    weights[3] = 2.00f;
    weights[4] = 1.25f;
    weights[5] = 0.75f;
    weights[6] = 0.50f;
}

int checkResults(float* cudaRes, float* res, int nElements) {
    int nDiffs=0;
    const float smallVal = 0.000001f;
    for (int i=0; i<nElements; i++)
        if ( fabs(cudaRes[i]-res[i])>smallVal )
            nDiffs++;
    return nDiffs;
}
```
**Example: Debugging & Profiling**

[1DStencil Code: the actual stencil function]

```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);
    }
}

global void applyStencil1D_SEQ(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);
    }
}
```
First Version...

[negrut@euler CodeBits]$ qsub -I -l nodes=1:gpus=1:default -X
[negrut@euler01 CodeBits]$ nvcc -gencode arch=compute_20, code=sm_20 testV1.cu
[negrut@euler01 CodeBits]$ ./testV1
Doesn't look good: 57 differences
[negrut@euler01 CodeBits]$
```c
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);
    float *cuda_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+511)/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
    applyStencil1D_SEQ(RADIUS, N-RADIUS, weights, in, out);
    cudaMemcpy(cuda_out, d_out, size, cudaMemcpyDeviceToHost);
    int nDiffs = checkResults(cuda_out, out, N);
    nDiffs==0? std::cout<<"Looks good.\n": std::cout<<"Doesn't look good: "<< nDiffs << " differences\n";
    //free resources
    free(weights); free(in); free(out); free(cuda_out);
    cudaFree(d_weights); cudaFree(d_in); cudaFree(d_out);
    return 0;
}
```

Example: Debugging & Profiling

[1DStencil Code]
[negrut@euler01 CodeBits]$ nvcc -gencode arch=compute_20, code=sm_20 testV2.cu
[negrut@euler01 CodeBits]$ ./testV2
Doesn't look good: 4 differences
[negrut@euler01 CodeBits]$

- **Reason:** checkResults runs a loop over all 1,000,000 entries. It should exclude the first RADIUS and last RADIUS of them. Those entries are not computed, you pick up whatever was there when memory was allocated on the host and on the device. As such, it gives false positives

- **NOTE:** this problem is not reproducible always (sometimes code runs ok, sometimes gives you a false positive)
Third Version

**testV2.cu**

```c
int checkResults(float* cudaRes, float* res, int nElements) {
    int nDiffs=0;
    const float smallVal = 0.000001f;
    for(int i=0; i<nElements; i++)
        if(fabs(cudaRes[i]-res[i])>smallVal)
            nDiffs++;
    return nDiffs;
}
```

**testV3.cu**

```c
int checkResults(int startElem, int endElem, float* cudaRes, float* res) {
    int nDiffs=0;
    const float smallVal = 0.000001f;
    for(int i=startElem; i<endElem; i++)
        if(fabs(cudaRes[i]-res[i])>smallVal)
            nDiffs++;
    return nDiffs;
}
```
Third Version [V3]...

[negrut@euler01 CodeBits]$ nvcc -gencode arch=compute_20, code=sm_20 testV3.cu
[negrut@euler01 CodeBits]$ ./testV3
Looks good.
[negrut@euler01 CodeBits]$

- Things are good now...
Code Profiling…

- Code looks like running ok, no evident bugs
- Time to profile the code, we’ll use the Lazy Man’s approach
- Profile V3 version
  - Create base results, both for compute capability 1.0 (Tesla) and 2.0 (Fermi)
Lazy Man’s Solution…

```bash
>> nvcc -O3 -gencode arch=compute_20,code=sm_20 testV3.cu -o testV3_20
>> ./testV3_20
```

```cpp
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GTX 480
# CUDA_CONTEXT 1
# TIMESTAMPFACTOR fffff6c689a59e98
method,gputime,cputime,occupancy
method=[ memcpyHtoD ] gputime=[ 1.664 ] cputime=[ 9.000 ]
method=[ memcpyHtoD ] gputime=[ 995.584 ] cputime=[ 1193.000 ]
method=[ __Z14applyStencil1DiiPKfPfS1__ ] gputime=[ 189.856 ] cputime=[ 12.000 ] occupancy=[1.0]
method=[ memcpyDtoH ] gputime=[ 1977.728 ] cputime=[ 2525.000 ]
```

```bash
>> nvcc -O3 -gencode arch=compute_10,code=sm_10 testV3.cu -o testV3_10
>> ./testV3_10
```

```cpp
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GT 130M
# TIMESTAMPFACTOR 12764ee9b1842064
method,gputime,cputime,occupancy
method=[ memcpyHtoD ] gputime=[ 1787.232 ] cputime=[ 2760.139 ]
method=[ __Z14applyStencil1DiiPKfPfS1__ ] gputime=[ 68357.69 ] cputime=[ 8.85 ] occupancy=[0.667]
```
Improving Performance

- Here’s what we’ll be focusing on:

```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);
    }
}
```

- There are several opportunities for improvement to move from V3 to V4:
  - Too many accesses to global memory (an issue if you don’t have L1 cache)
  - You can unroll the 7-iteration loop (it’ll save you some pocket change)
  - You can use shared memory (important if you don’t have L1 cache, i.e., in 1.0)
  - You can use pinned host memory [you have to look into main() to this end]
Improving Performance [V4]

- Version V4: Take care of
  - Repeated access to global memory
  - Loop unrolling

```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) {
        float result = 0.f;
        result += weights[0]*in[i-3];
        result += weights[1]*in[i-2];
        result += weights[2]*in[i-1];
        result += weights[3]*in[i];
        result += weights[4]*in[i+1];
        result += weights[5]*in[i+2];
        result += weights[6]*in[i+3];
        result /= 7.f;
        out[i] = result;
    }
}
```

- Even now there is room for improvement (this is V5 and V6 - part of your Assignment)
  - You can have `weights` and `in` stored in shared memory
  - You can use pinned memory (mapped memory) on the host
Lazy Man’s Profiling: V4

Compute capability 2.0 (Fermi)

```
>> 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 fffffff6c689a404a8
method, gputime, cputime, occupancy
method=[ memcpyHtoD ] gputime=[ 1001.952 ] cputime=[ 1197.000 ]
method=[ __Z14applyStencil1DiiPKfPfS1__ ] gputime=[ 166.944 ] cputime=[ 13.000 ] occupancy=[1.0]
method=[ memcpyDtoH ] gputime=[ 1394.144 ] cputime=[ 2533.000 ]
```

Compute capability 1.0 (Tesla/G80)

```
>> 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 ]
```
Timing Results
[Two Different Approaches (V3, V4) & Two Different GPUs (sm_20, sm_10)]
[each executable was run 7 times; script available on the class website]

<table>
<thead>
<tr>
<th>V4_20</th>
<th>V3_20</th>
<th>V4_10</th>
<th>V3_10</th>
</tr>
</thead>
<tbody>
<tr>
<td>166.752</td>
<td>190.560</td>
<td>47341.566</td>
<td>68611.008</td>
</tr>
<tr>
<td>166.912</td>
<td>190.016</td>
<td>47332.930</td>
<td>68531.875</td>
</tr>
<tr>
<td>166.976</td>
<td>190.208</td>
<td>47391.039</td>
<td>68674.109</td>
</tr>
<tr>
<td>166.368</td>
<td>190.048</td>
<td>47252.734</td>
<td>68679.422</td>
</tr>
<tr>
<td>166.848</td>
<td>189.696</td>
<td>47371.426</td>
<td>68357.695</td>
</tr>
<tr>
<td>166.592</td>
<td>189.856</td>
<td>47250.465</td>
<td>68618.492</td>
</tr>
<tr>
<td>166.944</td>
<td>190.240</td>
<td>47379.902</td>
<td>68687.266</td>
</tr>
</tbody>
</table>

**Averages**

<table>
<thead>
<tr>
<th>V4_20</th>
<th>V3_20</th>
<th>V4_10</th>
<th>V3_10</th>
</tr>
</thead>
<tbody>
<tr>
<td>166.7702857</td>
<td>190.0891429</td>
<td>47331.43743</td>
<td>68594.26671</td>
</tr>
</tbody>
</table>

**Standard Deviations**

<table>
<thead>
<tr>
<th>Slowdown, sm_20</th>
<th>Slowdown, sm_10</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.98262109%</td>
<td>44.92326969%</td>
</tr>
</tbody>
</table>
This is how you should think about code profiling and optimization:

- Would you ever send out your CV right after you completed writing it?
- Probably not, you always go back and spend a bit of time polishing it…