ECE/ME/EMA/CS 759
High Performance Computing for Engineering Applications

CUDA Profiling and Optimization Issues

October 19, 2015
Quote of the Day

“Weaseling out of things is important to learn. It's what separates us from the animals... except the weasel.”

-- Homer Simpson
Before We Get Started

- Issues covered last time:
  - Wrap up, GPU parallel computing with the thrust library
    - Key concepts covered in the process: functors, zipping, fusing
  - Wrapped up atomic operations

- Today’s topics
  - CUDA profiling and optimization issues

- Assignment:
  - HW05 – due on Oct. 21 at 11:59 PM
Related to Midterm Exam

- Finished grading exam today, grades should be in shortly
  - Ang graded multiple choice problem (1st problem)
  - Dan graded last three problems

- Slightly lower scores than expected
  - Occupancy aspects seemed to cause some confusion
    - We’ll revisit the topic next time

- Ang has the exams, he’ll send out email w/ when/where to pick up
  - I’ll deliver the rest of the exams to you on Friday

- Come and see me during office hours or after class if you think we missed something in your exam
Application Development Process, CUDA

- Identify Optimization Opportunities
- Parallelize with CUDA, confirm functional correctness
  - Debugger – cuda-gdb
  - Memory Checker – cuda-memcheck
- Optimize
  - Use a code profiler
Code Timing/Profiling

- Entry level approach: Lazy man’s solution
  - Do nothing, instruct the runtime 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
  - Notes:
    - nvvp is a cross platform tool (linux, mac, windows)
    - Virtually impossible to use off Euler
Lazy Man’s Solution…

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

```
>> 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
```

```bash
# 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
# CUDADEVICE 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 ]

Euler

My old HP laptop
Lazy Man’s Solution...

```plaintext
>>> 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 ]

# 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 ]
```
### Lazy Man’s Solution...

- **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 fffff6c689a404a8
method,gputime,cputime,occupancy
method=[ memcpyHtoD ] gputime=[ 1001.952 ] cputime=[ 1197.000 ]
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 12764e6b183e71e
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

- Provides a nice GUI and ample information regarding your execution

- 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

- Operation is fundamental to many algorithms
  - Standard discretization methods, interpolation, convolution, filtering,…

- This example will use weighted arithmetic mean
Serial Algorithm

\( f \)

\( \Rightarrow = \text{CPU Thread} \)

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

in

out

NVIDIA [S. Satoor]→
Serial Algorithm

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

Repeat for each element
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>
Application Optimization Process

- Identify Optimization Opportunities
  - 1D stencil algorithm

- Parallelize with CUDA and confirm functional correctness
  - `cuda-gdb`, `cuda-memcheck`
  - Note: `cuda-memcheck` useful for memory debugging
    - Out of bounds accesses
    - Accessing misaligned data
    - Race conditions
    - Memory leaks

- Optimize
  - …dealing with this next, using `nvvp`
Parallel Algorithm

Serial: One element at a time

Parallel: Many elements at a time

\( \downarrow \) = Thread
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);
    }
}
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);
   }
}
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);
}
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);
}

__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);
    }
}
```

Launch a GPU thread for each element
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,wsizesize,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, cudaMemcpy DeviceToHost);

    //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
__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 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>

NVIDIA [S. Satoo]
NVIDIA Visual Profiler

Timeline of CPU and GPU activity

Kernel and memcpy details

<table>
<thead>
<tr>
<th>Process: 8058</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread: 2127574912</td>
<td>cudaMemcpy</td>
<td>cudaMemcpy</td>
</tr>
<tr>
<td>Runtime API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Driver API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0] Tesla C2075</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Context 1 (CUDA)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MemCPy (HtoD)</td>
<td>MemCPy Hto...</td>
<td>MemCPy DtoH</td>
</tr>
<tr>
<td>Compute</td>
<td>apply...</td>
<td>apply...</td>
</tr>
<tr>
<td>Streams</td>
<td>MemCPy Hto...</td>
<td>MemCPy DtoH</td>
</tr>
<tr>
<td>Stream 1</td>
<td>apply...</td>
<td>MemCPy DtoH</td>
</tr>
</tbody>
</table>

**applyStencil1D_gpu(int, int, float ...)**

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>69.628 ms</td>
</tr>
<tr>
<td>Duration</td>
<td>8.177 ms</td>
</tr>
<tr>
<td>Grid Size</td>
<td>[32768,1,1]</td>
</tr>
<tr>
<td>Block Size</td>
<td>[512,1,1]</td>
</tr>
<tr>
<td>Registers/Thread</td>
<td>20</td>
</tr>
<tr>
<td>Shared Memory/Block</td>
<td>0 bytes</td>
</tr>
<tr>
<td>Occupancy</td>
<td>Theoretical</td>
</tr>
<tr>
<td>Theoretical</td>
<td>100%</td>
</tr>
<tr>
<td>L1 Cache Configuration</td>
<td></td>
</tr>
<tr>
<td>Shared Memory Request</td>
<td>48 KB</td>
</tr>
<tr>
<td>Shared Memory Exec</td>
<td>48 KB</td>
</tr>
</tbody>
</table>

GPU: 0.058926 seconds, 2.27773 GBytes/s, 0.284716 GElements/s
NVIDIA Visual Profiler

CUDA API activity on CPU

Memcpy and kernel activity on GPU
Detecting Low Memory Throughput

- Spent 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

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.

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 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

Copyright © 2011 NVIDIA Corporation | www.nvidia.com

NVIDIA [S. Satoor]
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
[side trip]
CUDA: Pageable vs. Pinned Data Transfer

Pageable Data Transfer

Device

DRAM

Host

Pageable Memory

Pinned Memory

Pinned Data Transfer

Device

DRAM

Host

Pinned Memory
Pinned CPU Memory Result

Memcpy DtoH [sync]

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>117.622 ms</td>
</tr>
<tr>
<td>Duration</td>
<td>10.469 ms</td>
</tr>
<tr>
<td>Size</td>
<td>64 MB</td>
</tr>
<tr>
<td>Throughput</td>
<td>5.97 GB/s</td>
</tr>
</tbody>
</table>

<terminated> viper runhandler [Program] /home/david/depot/davidg-linux-sw/sw/pvt/davidg/sc11_example/stencil/run_gpu

GPU PINNED: 0.0297912 seconds, 4.50528 GBytes/s, 0.563158 GEelements/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]→
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

Asynchronous Transfers and Overlapping Transfers with Computation

Data transfers between the host and the device using `cudaMemcpy()` are blocking transfers; that is, control is returned to the host thread only after the data transfer is complete. The `cudaMemcpyAsync()` function is a non-blocking variant of `cudaMemcpy()` in which control is returned immediately to the host thread. In contrast with `cudaMemcpy()`, the asynchronous transfer version requires pinned host memory (see Pinned Memory), and it contains an additional argument, a stream ID. A stream is simply a sequence of operations that are performed in order on the device. Operations in different streams can be interleaved and in some cases overlapped—a property that can be used to hide data transfers between the host and the device.

Asynchronous transfers enable overlap of data transfers with computation in two different ways. On all CUDA-enabled devices, it is possible to overlap host computation with asynchronous data transfers and with device computations. For example, Overlapping computation and data transfers demonstrates how host computation in the
Data Partitioning Example

Partition data into TWO chunks

chunk 1 chunk 2

in

out
Data Partitioning Example

chunk 1

memcpy
compute

chunk 2

memcpy

in

out

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

[problem broken into 16 chunks]

| Process: 8689 |
| Thread: 812144512 |
| Runtime API |
| Driver API |
| [0] Tesla C2075 |
| Context 1 (CUDA) |
| MemCpy (HtoD) |
| MemCpy (DtoH) |
| Compute |
| 3.7% [16] applyS... |
| Streams |
| Stream 1 |
| Stream 6 |
| Stream 7 |
| Stream 8 |
| Stream 9 |

\[\text{cudaDeviceSynchronize}\]
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>

ME759: Use of multiple streams covered in a week
Optimization Summary
[Looking Back at 1D Stencil Example…]

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

- Optimize memory throughput
  - Minimal code change, yet 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
Iterative Optimization

- Identify Optimization Opportunities
- Parallelize
- Optimize
Revisit Stencil Example

● Problem setup
  ● 1,000,000 elements
  ● RADIUS is 3

● Purpose:
  ● Show a typical bug and then one easy way to get some extra performance out of the code
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, with 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
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);
    }
}

__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);
    }
}
```
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]$
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]
Second Version...

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

- 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)
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;
}

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

# 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=[ _Z14applyStencil1DiiPKfPiPKfS1_ ] 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

# 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=[ _Z14applyStencil1DiiPKfPiPKfS1_ ] 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
  - 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

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

Averages

|       | 166.7702857 | 190.0891429 | 47331.43743 | 68594.26671 |

Standard Deviations

|       | 0.132410266 | 0.147947777 | 0.123060609 | 0.171466201 |

Slowdown, sm_20

13.98262109%

Slowdown, sm_10

44.92326969%
Concluding Slide
Profiling & Optimization

- Question:
  - 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…

- Same applies to your code
  - Don’t declare victory as soon as code runs without crashing
  - Debug, profile, optimize
Putting Things in Perspective…

Here’s what we’ve covered so far:
- CUDA execution configuration (grids, blocks, threads)
- CUDA scheduling issues (warps, thread divergence, synchronization, etc.)
- CUDA Memory ecosystem (registers, shared mem, device mem, L1/L2 cache, etc.)
- Practical things: building, debugging, profiling CUDA code

Next: CUDA GPU Programming - Examples & Code Optimization Issues
- Tiling: a CUDA programming pattern
- Example: CUDA optimization exercise in relation to a vector reduction operation
- CUDA Execution Configuration Optimization Heuristics: Occupancy issues
- CUDA Optimization Rules of Thumb