Application Optimization
Using CUDA
Development Tools
Optimization: CPU and GPU

**CPU**
- A few cores
- Good memory bandwidth
- Best at serial execution

**GPU**
- Hundreds of cores
- Great memory bandwidth
- Best at parallel execution
Optimization: Maximize Performance

- Take advantage of strengths of both CPU and GPU
- Entire application does not need to be ported to GPU
Application Optimization Process and Tools

- Identify Optimization Opportunities
  - gprof
  - Intel VTune
- Parallelize with CUDA, confirm functional correctness
  - cuda-gdb, cuda-memcheck
  - Parallel Nsight Memory Checker, Parallel Nsight Debugger
  - 3rd party: Allinea DDT, TotalView
- Optimize
  - NVIDIA Visual Profiler
  - Parallel Nsight
  - 3rd party: Vampir, Tau, PAPI, …
1D Stencil: A Common Algorithmic Pattern

- 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
  - Our example will use weighted arithmetic mean
Serial Algorithm

(radius = 3)

in

out

\( \Rightarrow \) = Thread
Serial Algorithm

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

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

\[ \text{Repeat for each element} \]
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);
}
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) {
    float [1] = 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 and initialize
Apply stencil
Cleanup
```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);
    }
}
```

For each element...

Weighted mean over radius
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>
Parallel Algorithm

Serial: 1 element at a time

Parallel: many elements at a time

= Thread
```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);
    float *d_weights; cudaMalloc(&d_weights, wsize);
    float *d_in; cudaMalloc(&d_in, wsize);
    float *d_out; cudaMalloc(&d_out, wsize);
    cudaMemcpy(d_weights,weights,wsize,cudaMemcpyHostToDevice);
    cudaMemcpy(d_in, in, wsize, cudaMemcpyHostToDevice);
    applyStencil1D<<<N/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
    cudaMemcpy(out, d_out, wsize, cudaMemcpyDeviceToHost);
    //free resources
    free(weights); free(in); free(out);
    cudaFree(d_weights); cudaFree(d_in); cudaFree(d_out);
}
```
Parallel Implementation With CUDA

```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);
    float *d_weights; cudaMalloc(&d_weights, wsize);
    float *d_in; cudaMalloc(&d_in, wsize);
    float *d_out; cudaMalloc(&d_out, wsize);
    cudaMemcpy(d_weights, weights, wsize, cudaMemcpyHostToDevice);
    cudaMemcpy(d_in, in, wsize, cudaMemcpyHostToDevice);
    applyStencil1D<<<N/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
    cudaMemcpy(out, d_out, wsize, 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 Idx = blockIdx.x*blockDim.x + threadIdx.x;
    if (Idx < eIdx) {
        float 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
Parallel Implementation With CUDA

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);
    float *d_weights; cudaMalloc(&d_weights, wsize);
    float *d_in; cudaMalloc(&d_in, wsize);
    float *d_out; cudaMalloc(&d_out, wsize);

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

Copy inputs to GPU
Copy results from GPU
Parallel Implementation With CUDA

```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);
    float *d_weights; cudaMalloc(&d_weights, wsize);
    float *d_in; cudaMalloc(&d_in, wsize);
    float *d_out; cudaMalloc(&d_out, wsize);
    cudaMemcpy(d_weights, weights, wsize, cudaMemcpyHostToDevice);
    cudaMemcpy(d_in, in, wsize, cudaMemcpyHostToDevice);
    applyStencil1D<<<N/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
    cudaMemcpy(out, d_out, wsize, 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;
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
```

**Indicates GPU kernel**

**Launch a thread 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);
    float *d_weights;
    cudaMalloc(&d_weights, wsize);
    float *d_in;
    cudaMalloc(&d_in, wsize);
    float *d_out;
    cudaMalloc(&d_out, wsize);
    cudaMemcpy(d_weights, weights, wsize, cudaMemcpyHostToDevice);
    cudaMemcpy(d_in, in, wsize, cudaMemcpyHostToDevice);
    applyStencil1D<<<N/512, 512>>>(RADIUS, N - RADIUS, d_weights, d_in, d_out);
    cudaMemcpy(out, d_out, wsize, 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];
        }
    } else {
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}

Parallel Implementation With CUDA

Get the array index for each thread.

Each thread executes kernel.
But our first run returns an error!

```bash
$ stencil1d
Segmentation fault
```

Debugging Tools:
- cuda-memcheck (memory checker)
- cuda-gdb (debugger)
- printf
Memory Checker: cuda-memcheck

$ cuda-memcheck stencil1d

======== CUDA-MEMCHECK
======== Invalid __global__ read of size 4
======== at 0x000000240 in stencil1d.cu:60:applyStencil1D
======== by thread (31,0,0) in block (0,0,0)
======== Address 0x20020047c is out of bounds
========
======== ERROR SUMMARY: 1 error
Debugger: cuda-gdb

$ cuda-gdb stencil1d

(cuda-gdb) set cuda memcheck on

(cuda-gdb) run
[Launch of CUDA Kernel 0
applyStencil1D<<<(32768,1,1),(512,1,1)>>>
on Device 0]
Program received signal CUDA_EXCEPTION_1, Lane Illegal Address.
applyStencil1D<<<(32768,1,1),(512,1,1)>>>
at stencil1d.cu:60

(cuda-gdb) cuda thread
thread (31,0,0)

__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);
    }
}

Reach the failure point
__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);
  }
}

Debugger: cuda-gdb

(cuda-gdb) print &weights[j+RADIUS]
(const float *) 0x20020003c

(cuda-gdb) print &in[i+j]
(float *) 0x20020047c

(cuda-gdb) print i+j
31

Found the bad array access
Debugger: cuda-gdb

float *d_weights; cudaMalloc(&d_weights, wsize);
float *d_in; cudaMalloc(&d_in, wsize);
float *d_out; cudaMalloc(&d_out, wsize);
cudaMemcpy(d_weights, weights, wsize, ...);
cudaMemcpy(d_in, in, wsize, ...);
applyStencil1D<<<N/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
cudaMemcpy(out, d_out, wsize, ...);

Switch to the CPU thread
Switch to the frame where the allocation occurred
Found bad allocation size
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);
    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);
    }
}
Parallel Nsight for Visual Studio

Debugger stops at the failure location

Detailed information

High level message of the access violation
OutOfRangeLoad
GPU Exception
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);
    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);
    }
}

<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>

*4 cores + hyperthreading
Printf

Commonly used for debugging, available on GPU

```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);
        if (i < 128)
            printf("out[%d] = %f\n", i, out[i]);
    }
}
```

$ stencil1d$
out[15] = 0.263680
out[31] = 0.276422
out[16] = 0.274778
out[32] = 0.227698
out[17] = 0.280459
out[18] = 0.263378
out[19] = 0.276602
out[20] = 0.248153
...

2x Performance In 2 Hours

In just a couple of hours we...
- Used CUDA to parallelize our application
- Used cuda-memcheck and cuda-gdb to detect and correct some bugs
- Got 2.2x speedup over parallelized and optimized CPU code

We used CUDA-C/C++, but other options available...
- Libraries (NVIDIA and 3rd party)
- Directives
- Other CUDA languages (Fortran, Java, …)
Application Optimization Process (Revisited)

- Identify Optimization Opportunities
  - 1D stencil algorithm

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

- Optimize
  - ?
Optimize

- Can we get more performance?

- Visual Profiler
  - Visualize CPU and GPU activity
  - Identify optimization opportunities
  - Automated analysis
NVIDIA Visual Profiler

![NVIDIA Visual Profiler Interface](image)

- Process: 8058
- Thread: 2127574912
- Runtime API: cudaMemcpy, cudaMemcpy
- Driver API: cudaMemcpy, cudaMemcpy
- Context 1 (CUDA): MemCpy (HtoD), MemCpy (DtoH)
- Compute: apply...
- Streams: Stream 1
- MemCpy Hto...
- MemCpy DtoH [sync]

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

- Name
  - Start: 69.628 ms
  - Duration: 8.177 ms
  - Grid Size: [32768,1,1]
  - Block Size: [512,1,1]
  - Registers/Thread: 20
  - Shared Memory/Block: 0 bytes
  - Occupancy: 100%
  - Theoretical: 100%
  - L1 Cache Configuration: 48 KB
  - Shared Memory Request: 48 KB

**GPU**:
- 0.058926 seconds, 2.27773 GBytes/s, 0.284716 GElements/s
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
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
  - CUDA Programming Guide

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 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.
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 Implementation

CPU allocations use pinned memory to enable fast memcpy

No other changes
Pinned CPU Memory Result

![CPU Memory Analysis](image)

- **Process**: 8527
- **Thread**: -1613166720
- **Runtime API**: cudaMemcpy
- **Driver API**: cudaMemcpy
- **Device**: Tesla C2075

### 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>

**Analysis**: 

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>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

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 2 chunks

chunk 1

chunk 2

in

out
Data Partitioning Example

chunk 1

memcpy
compute
memcpy

chunk 2

in

memcpy
compute
memcpy

out
Data Partitioning Example

In the diagram, the process is divided into two chunks (chunk 1 and chunk 2). Data is transferred from 'in' to 'chunk 1' via `memcpy`, then processed by 'compute'. After the computation, the result is transferred to 'chunk 2' via `memcpy`. Finally, from 'chunk 2', the data is copied to 'out' again using `memcpy`.

The diagram illustrates the sequential execution of data movements and computations.
### Overlapped Compute/Memcpy

<table>
<thead>
<tr>
<th>Process: 8689</th>
<th>0.1 s</th>
<th>0.105 s</th>
<th>0.11 s</th>
<th>0.115 s</th>
<th>0.12 s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread: 812144512</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Runtime API</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Driver API</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0] Tesla C2075</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Context 1 (CUDA)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MemCpy (HtoD)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MemCpy (DtoH)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compute</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.7% [16] applyS...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Streams</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stream 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stream 6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stream 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stream 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stream 9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **cudaDeviceSynchronize**
Overlapped Compute/Memcpy

- Compute time completely “hidden”
- Exploit dual memcpy engines
## Overlapped Compute/Memcpy 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>
<tr>
<td>Tesla C2075</td>
<td>Overlap</td>
<td>935</td>
<td>7.2x</td>
</tr>
</tbody>
</table>

*4 cores + hyperthreading
Parallel Nsight For Visual Studio

Timeline
CPU & GPU
Correlation Details
Correlation Hierarchy
Parallel Nsight For Visual Studio

Instruction, Branch, Memory and Other Analysis
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 with CUDA
- Optimize
Optimization Summary

- Initial CUDA parallelization and functional correctness
  - 1-2 hours
  - 2.2x speedup
- Optimize memory throughput
  - 1-2 hours
  - 4.3x speedup
- Overlap compute and data movement
  - 1-2 days
  - 7.2x speedup
Summary

CUDA accelerates compute-intensive parts of your application

Tools are available to help with:
- Identifying optimization opportunities
- Functional correctness
- Performance optimization

Get Started
- Join the community: [developer.nvidia.com/join](http://developer.nvidia.com/join)
- Check out the booth demo stations, experts table
- See Parallel Nsight at the Microsoft booth (#1601 – 4th floor bridge)
- Get stencil example: [developer.nvidia.com/sc11](http://developer.nvidia.com/sc11)
Questions?