Performance Optimization Using the NVIDIA Visual Profiler
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

- CPU Memory: 32 GB/s
- GPU Memory: 177 GB/s
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)

\[ \sim \sim = \text{Thread} \]
Serial Algorithm

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

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

Allocate and initialize

Apply stencil

Cleanup
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, 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>
Parallel Algorithm

<table>
<thead>
<tr>
<th></th>
<th>Serial: 1 element at a time</th>
<th>Parallel: many elements at a time</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td><img src="image1" alt="Serial: 1 element at a time" /></td>
<td><img src="image2" alt="Parallel: many elements at a time" /></td>
</tr>
<tr>
<td>out</td>
<td><img src="image1" alt="Serial: 1 element at a time" /></td>
<td><img src="image2" alt="Parallel: many elements at a time" /></td>
</tr>
</tbody>
</table>

※ = Thread
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, 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 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, 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);
}
```

Allocate GPU memory
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, 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);
    }
}
```

Copy inputs to GPU

```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, 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 Implementation With CUDA

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, 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);
}

__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 Implementation With CUDA

Get the array index for each thread.

Each thread executes kernel.
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, 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);
    }
}
```

Copy results from GPU
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
2x Performance In 2 Hours

- In just a couple of hours we...
  - Used CUDA to parallelize our application
  - 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

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
    - 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
Several types of analysis are provided.

Analysis pointing out low memcpy throughput:

- LowMemcpy/Compute Overlap [0 ns / 8.176 ms = 0%]
- The amount of time performing compute is low relative to the amount of time required for memcpy.

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

- LowMemcpy Overlap [0 ns / 15.79 ms = 0%]
Online Optimization Help

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

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

Each analysis has link to Best Practices documentation
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

GPU PINNED: 0.0297912 seconds, 4.56528 GBytes/s, 0.563158 GEElements/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

in

chunk 1

memcpy

compute

chunk 2

memcpy

out
Data Partitioning Example

chunk 1

memcpy
compute
memcpy

chunk 2

memcpy
compute
memcpy

in

out
Overlapped Compute/Memcpy

![Diagram showing overlapped compute and memcpy operations](image_url)
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>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>

Note: *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)
  - 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
Visual Profiler Demo
Summary

- CUDA accelerates compute-intensive parts of your application
- Visual profiler helps in performance analysis and optimization

Get Started
- Join the community: [developer.nvidia.com/join](http://developer.nvidia.com/join)
Questions?
Performance optimization strategies

- Maximize parallel execution to achieve maximum utilization
- Optimize memory usage to achieve maximum memory throughput
- Optimize instruction usage to achieve maximum instruction throughput
Performance Optimization Process

- Use appropriate performance metric for each kernel
- Determine what limits kernel performance
  - Memory throughput
  - Instruction throughput
  - Latency
  - Combination of the above
- Address the limiters in the order of importance
  - Determine how close to the HW limits the resource is being used
  - Analyze for possible inefficiencies
  - Apply optimizations
3 Ways to Assess Performance Limiters

- **Algorithmic**
  - Based on algorithm’s memory and arithmetic requirements
  - Least accurate: undercounts instructions and potentially memory accesses

- **Profiler**
  - Based on profiler-collected memory and instruction counters
  - More accurate, but doesn’t account well for overlapped memory and arithmetic

- **Code modification**
  - Based on source modified to measure memory-only and arithmetic-only times
  - Most accurate, however cannot be applied to all codes