

The logo for GPU Technology Conference is located in the top-left corner. It consists of a green rectangular box with a small triangle pointing downwards on its left side. Inside the box, the text "GPU" is written in a large, bold, white sans-serif font, and "TECHNOLOGY CONFERENCE" is written in a smaller, white sans-serif font to its right.

GPU TECHNOLOGY
CONFERENCE

The background of the slide is a detailed, top-down view of a GPU die. The die is dark, and its intricate circuitry is highlighted with vibrant, glowing lines in various colors including red, orange, yellow, green, cyan, blue, and purple. These lines form a complex grid and pattern across the surface of the chip.

Performance Optimization Using the NVIDIA Visual Profiler

Optimization: CPU and GPU

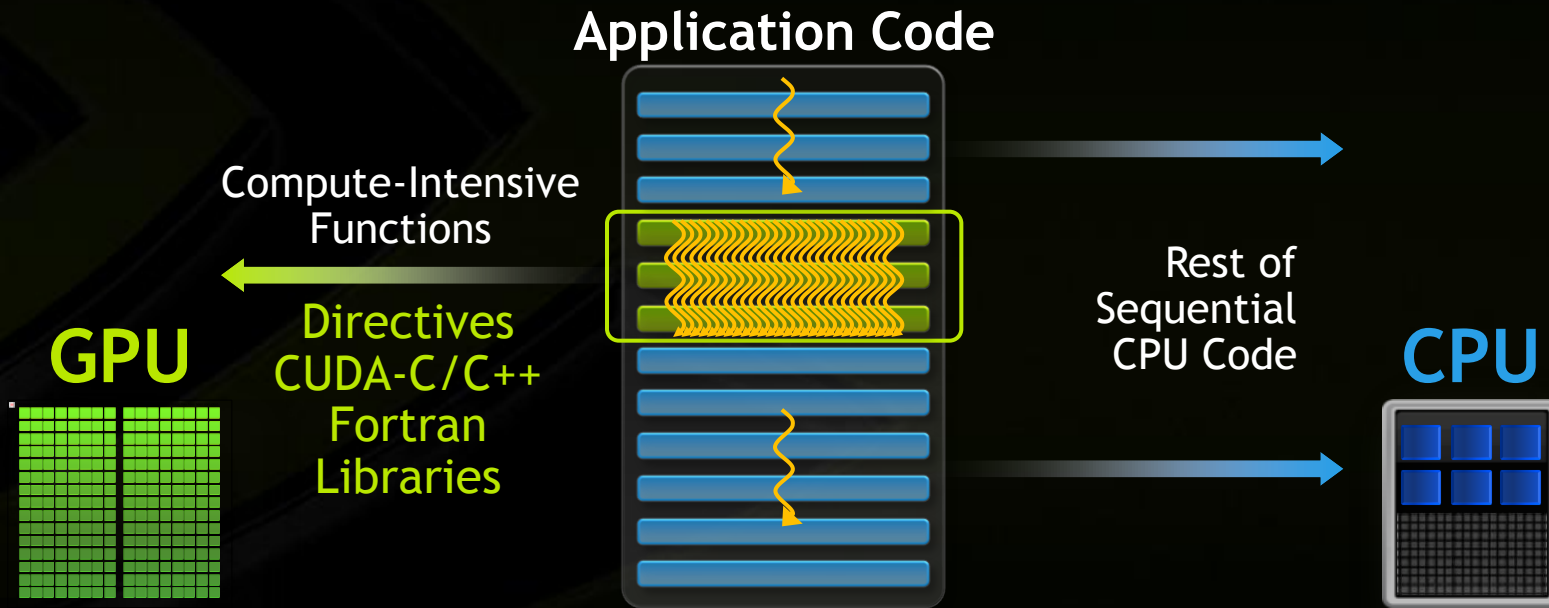


- A few cores
- Good memory bandwidth
- Best at serial execution

- 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

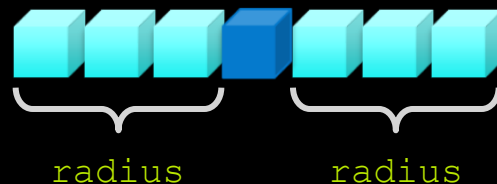


Application Optimization Process and Tools

- 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

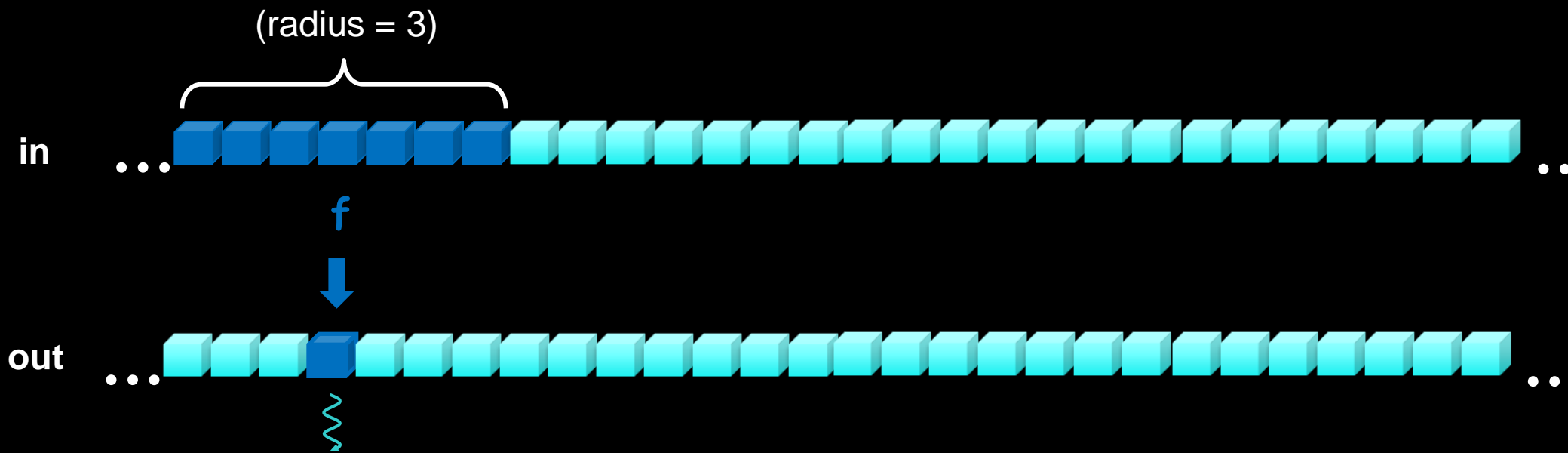
- Applying a 1D stencil to a 1D array of elements
 - Function of input elements within a radius



- Fundamental to many algorithms
 - Standard discretization methods, interpolation, convolution, filtering
- Our example will use weighted arithmetic mean

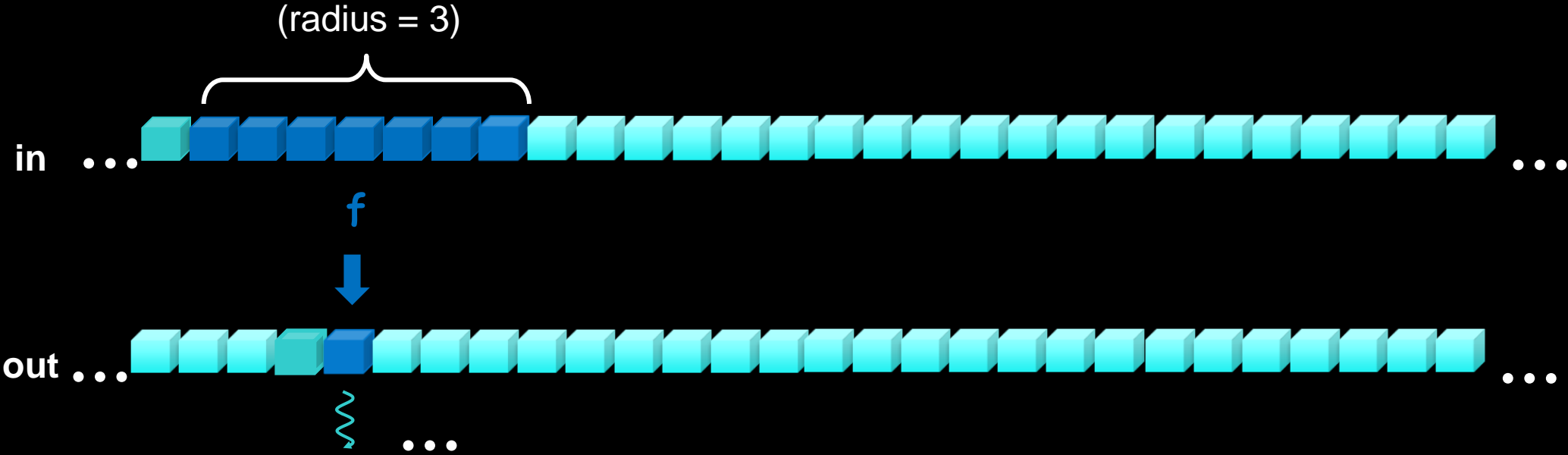
Serial Algorithm

⤿ = Thread



Serial Algorithm

⋈ = Thread



Repeat for each element

Serial Implementation



```
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



```
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);  
}
```

Allocate and initialize

Apply stencil

Cleanup

```
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



```
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, wsize);
    initializeArray(in, N);

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

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

Weighted mean over radius

```
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);
    }
}
```

For each element...

Serial Implementation Performance



```
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, 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);
    }
}
```

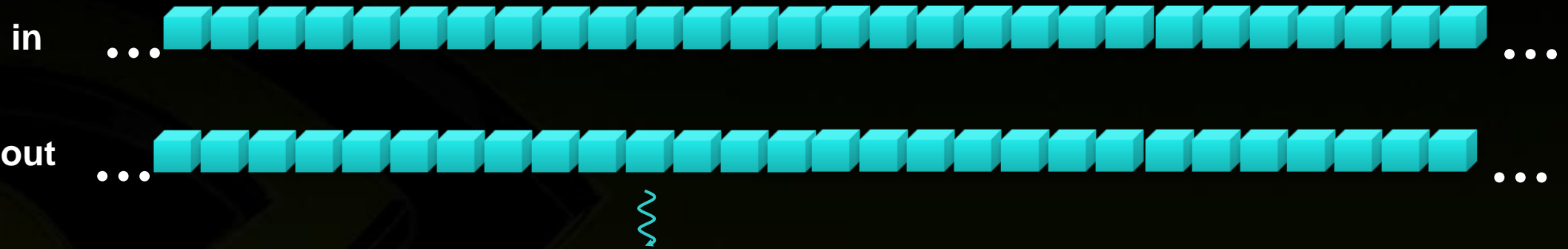
CPU	MElements/s
i7-930	30

Parallel Algorithm

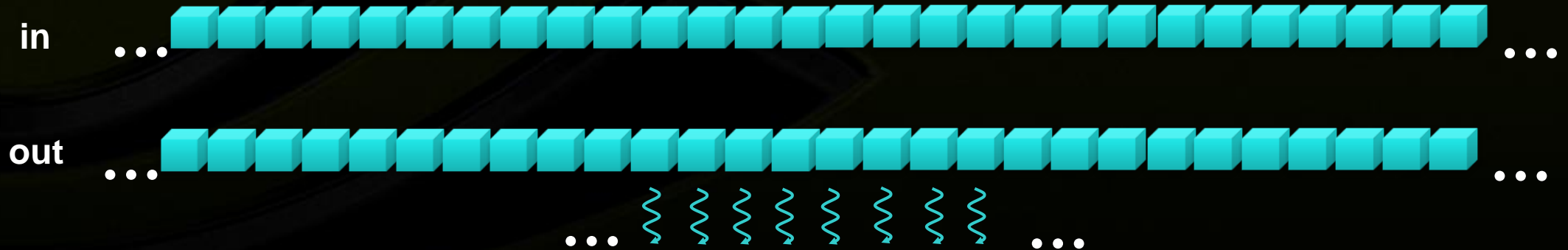


⚡ = Thread

Serial: 1 element at a time



Parallel: many elements at a time



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, 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);
}
```

Indicates
GPU
kernel

```
__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



```
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);
}
```

Allocate GPU memory

```
__global__ void applyStencil1D(int sIdx, int eIdx,
    const float *weights, float *in, float *out) {
    int i = sIdx + blockIdx.x*blockDim.x + threadIdx.x;
    float sum = 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



```
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;
        //for 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

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, size);
    float *d_out;     cudaMalloc(&d_out, size);

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

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

Launch a thread for each element

```
__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



```
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);
}
```

Get the array index for each 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);
    }
}
```

Each thread executes kernel

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, 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);
    }
}
```

Copy results
from GPU

Parallel Implementation Performance



```
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);
    }
}
```

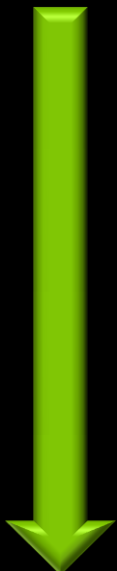
Device	Algorithm	MElements/s	Speedup
i7-930*	Optimized & Parallel	130	1x
Tesla C2075	Simple	285	2.2x

*4 cores + hyperthreading

2x Performance In 2 Hours

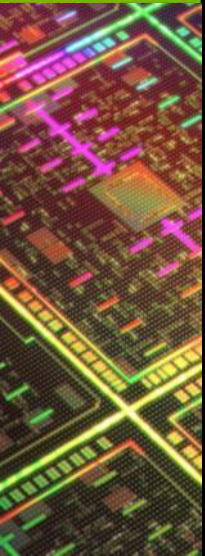
- In just a couple of hours we...
 - Used CUDA to parallelize our application
 - Got 2.2x speedup over parallelized and optimized CPU code
- We used CUDA-C/C++, but other options available...
 - Libraries (NVIDIA and 3rd party)
 - Directives
 - Other CUDA languages (Fortran, Java, ...)

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

File View Run Help

stencil.vp

0.05 s 0.075 s 0.1 s

Process: 8058
Thread: 2127574912
Runtime API: cudaMemcpy
Driver API
[0] Tesla C2075
Context 1 (CUDA)
MemCpy (HtoD): Memcpy Hto...
MemCpy (DtoH): Memcpy DtoH [sync]
Compute: apply...
5.4% [1] applySt...
Streams: Stream 1: Memcpy Hto..., apply..., Memcpy DtoH [sync]

Properties Detail Graphs

applyStencil1D_gpu(int, int, float ...)

Name	Value
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	
Theoretical	100%
L1 Cache Configuration	
Shared Memory Reque	48 KB
Shared Memory Execu	48 KB

Analysis Details Console Settings

<terminated> viper runhandler [Program] /home/david/depot/davidg-linux-sw/sw/pvt/davidg/sc11_exam
GPU: 0.058926 seconds, 2.27773 GBytes/s, 0.284716 GElements/s

Kernel and memcopy details

NVIDIA Visual Profiler



The screenshot displays the NVIDIA Visual Profiler interface for a process named 'stencil.vp'. The main window shows a timeline from 0.05 s to 0.1 s. The left sidebar indicates the process is running on a Tesla C2075 GPU. The main area shows CPU activity for 'Runtime API' with two 'cudaMemcpy' operations. GPU activity includes 'MemCpy (HtoD)', 'MemCpy (DtoH)', and 'Compute' operations. A 'Detail Graphs' panel on the right shows the 'applyStencil1D_gpu' kernel with a duration of 8.177 ms and a grid size of [32768, 1, 1]. A console window at the bottom shows the command path and GPU performance metrics: GPU: 0.058926 seconds, 2.27773 GBytes/s, 0.284716 GElements/s.

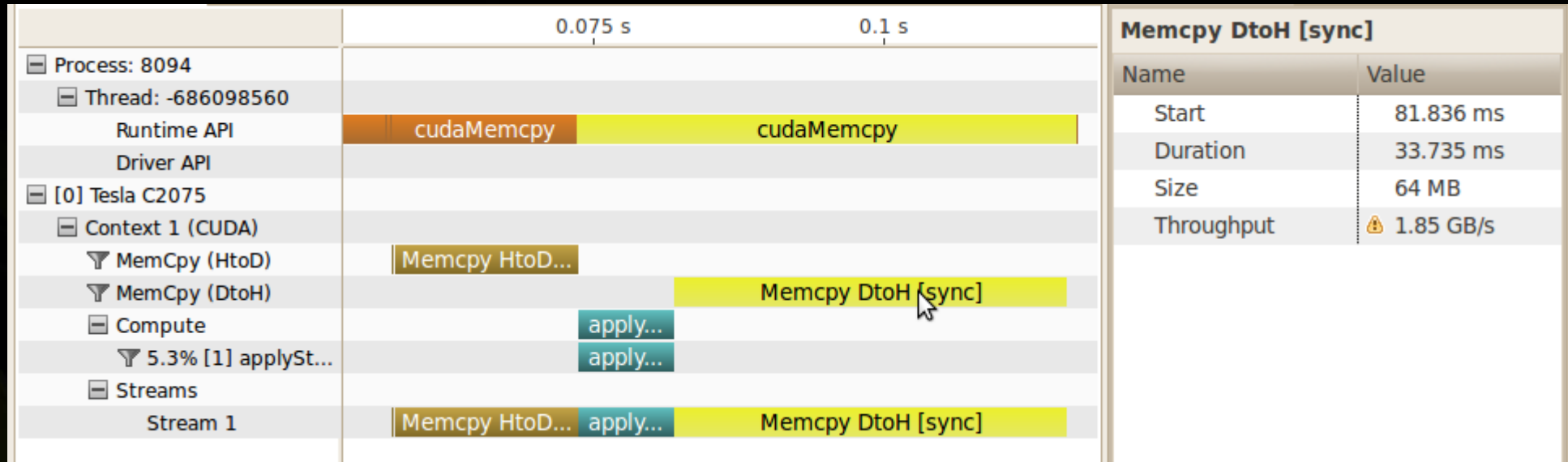
CUDA API activity on CPU

Memcpy and kernel activity on GPU

Name	Value
Start	69.628 ms
Duration	8.177 ms
Grid Size	[32768,1,1]
Block Size	[512,1,1]
Registers/Thread	20
L1 Cache	
Shared Memory Reque	48 KB
Shared Memory Execu	48 KB

```
<terminated> viper runhandler [Program] /home/david/depot/davidg-linux-sw/sw/pvt/davidg/sc11_example/stencil/run_gpu
GPU: 0.058926 seconds, 2.27773 GBytes/s, 0.284716 GElements/s
```


Detecting Low Memory Throughput

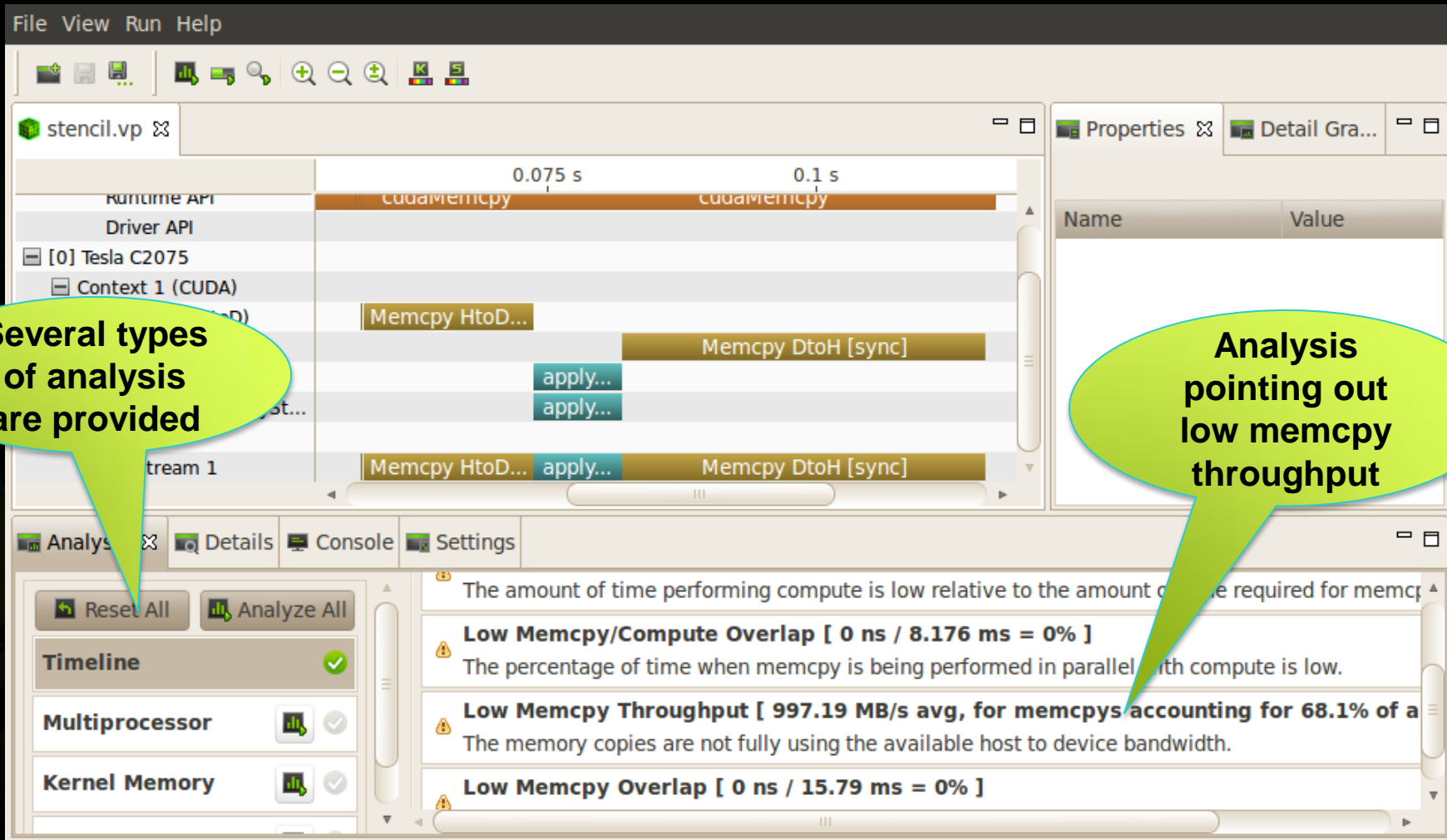


- Spend majority of time in data transfer
 - Often can be overlapped with preceding or following computation
- From timeline can see that throughput is low
 - PCIe x16 can sustain > 5GB/s

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

Visual Profiler Analysis



Several types of analysis are provided

Analysis pointing out low memcopy throughput

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. [More..](#)

The screenshot shows a web browser displaying the NVIDIA Visual Profiler documentation. The left sidebar contains a navigation tree with the following items: Visual Profiler Optimizations, Preface, Parallel Computing with CUDA, Performance Metrics, Memory Optimizations (expanded), Data Transfer Between Host and Device (expanded), Pinned Memory (selected), Asynchronous Transfers, Zero Copy, Device Memory Spaces, Allocation, Execution Configuration Options, Instruction Optimizations, Control Flow, Recommendations and Best Practices, and NVCC Compiler Switches. The main content area is titled "Pinned Memory" and contains the following text: "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." Below the text, it says "Parent topic: [Data Transfer Between Host and Device](#)". At the bottom of the page, there is a copyright notice: "Copyright © 2011 NVIDIA Corporation | www.nvidia.com" and the NVIDIA logo.

Each analysis has link to Best Practices documentation

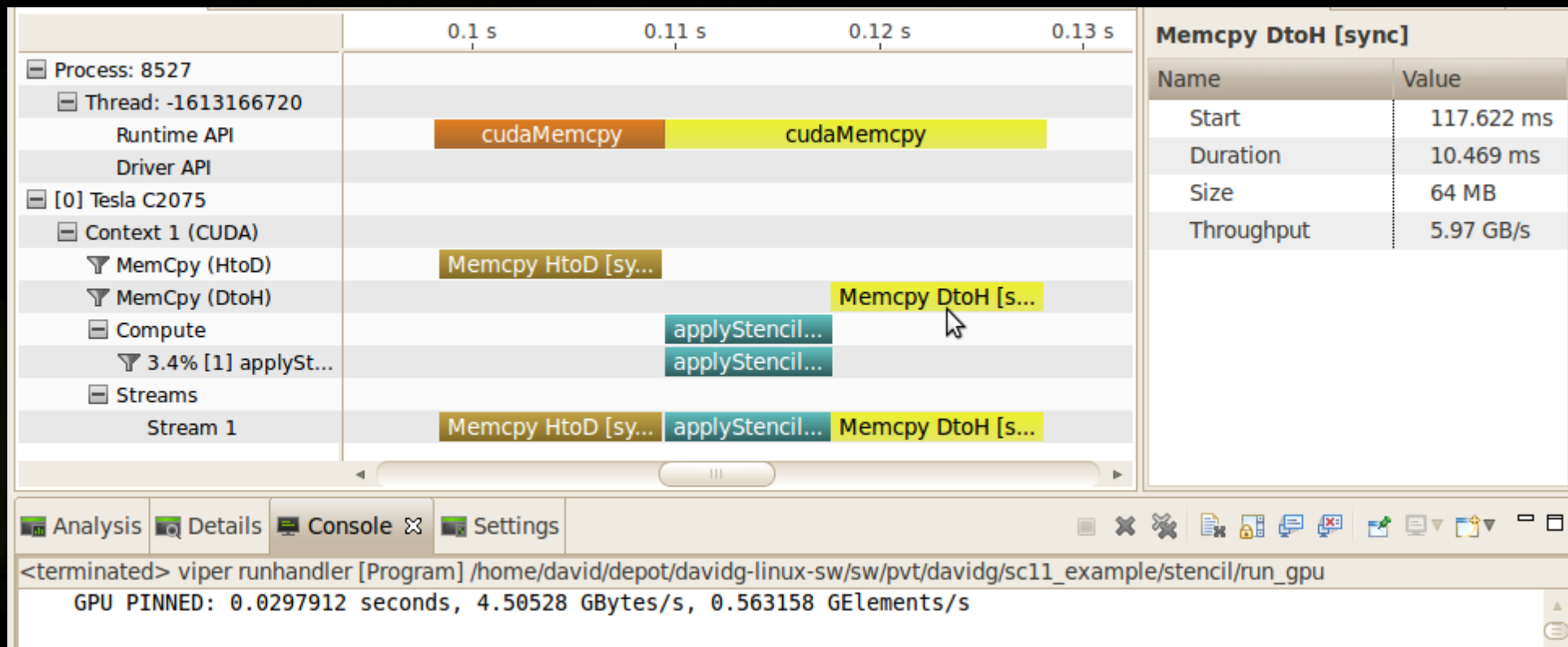
Pinned CPU Memory Implementation

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

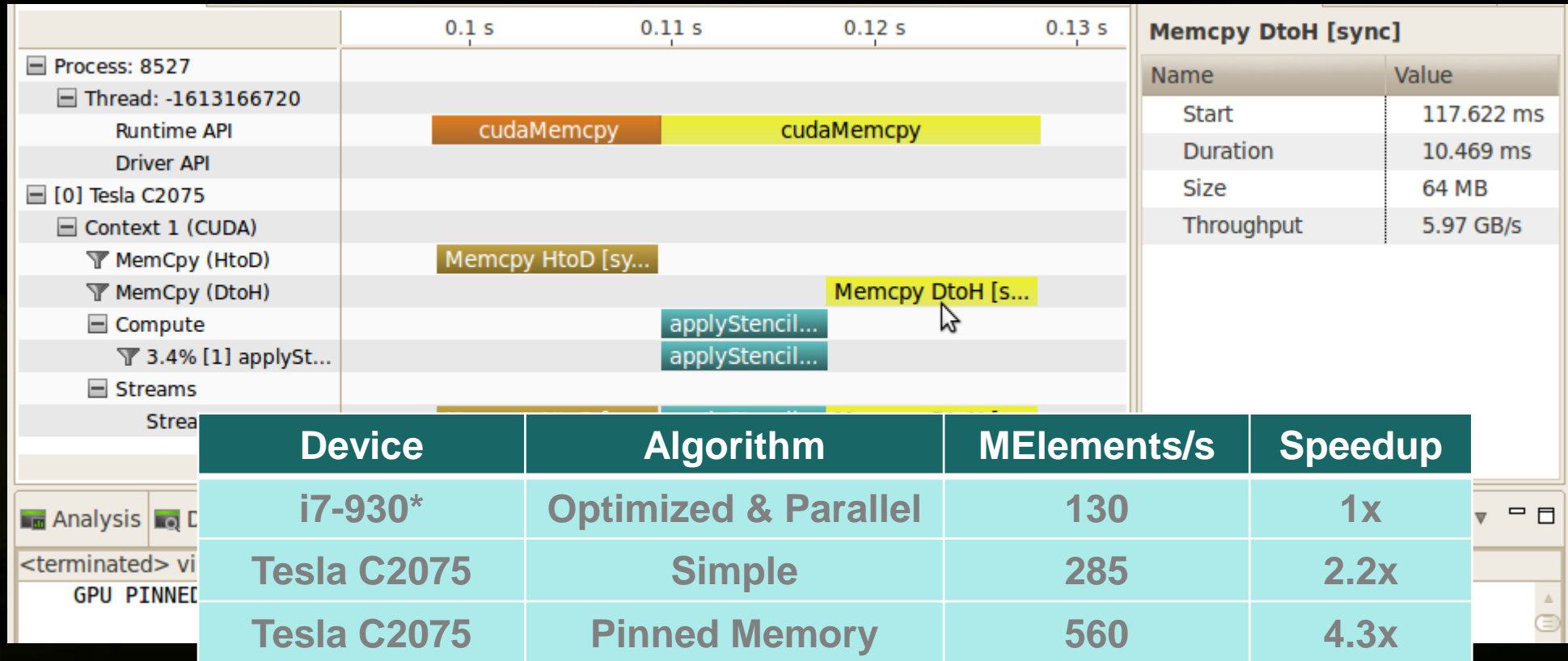
CPU allocations
use pinned
memory to enable
fast memcpy

No other changes

Pinned CPU Memory Result



Pinned CPU Memory Result



*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)

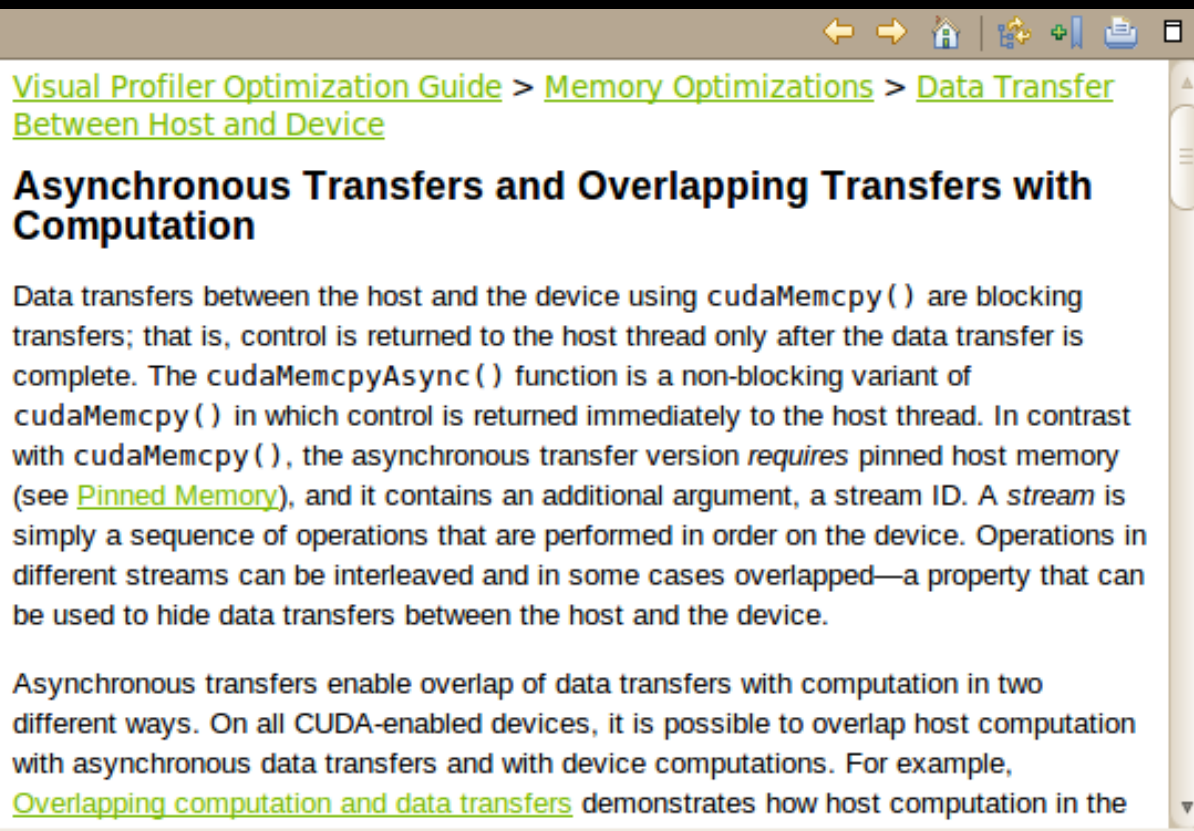


Low Memcpy/Compute Overlap [0 ns / 8.176 ms = 0%]

The percentage of time when memcpy is being performed in parallel with compute is low.

[More...](#)

- Advanced optimization
 - Larger time investment
 - Potential for larger speedup



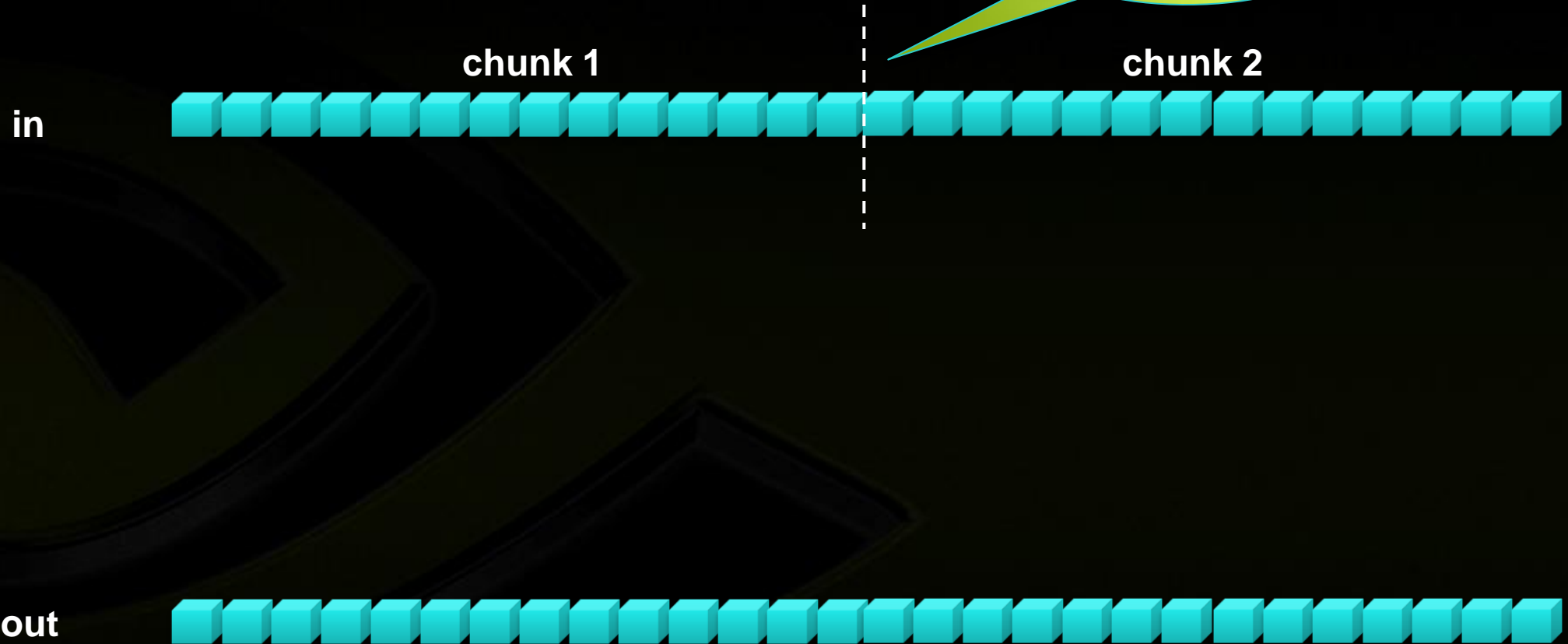
Visual Profiler Optimization Guide > Memory Optimizations > Data Transfer Between Host and Device

Asynchronous Transfers and Overlapping Transfers with Computation

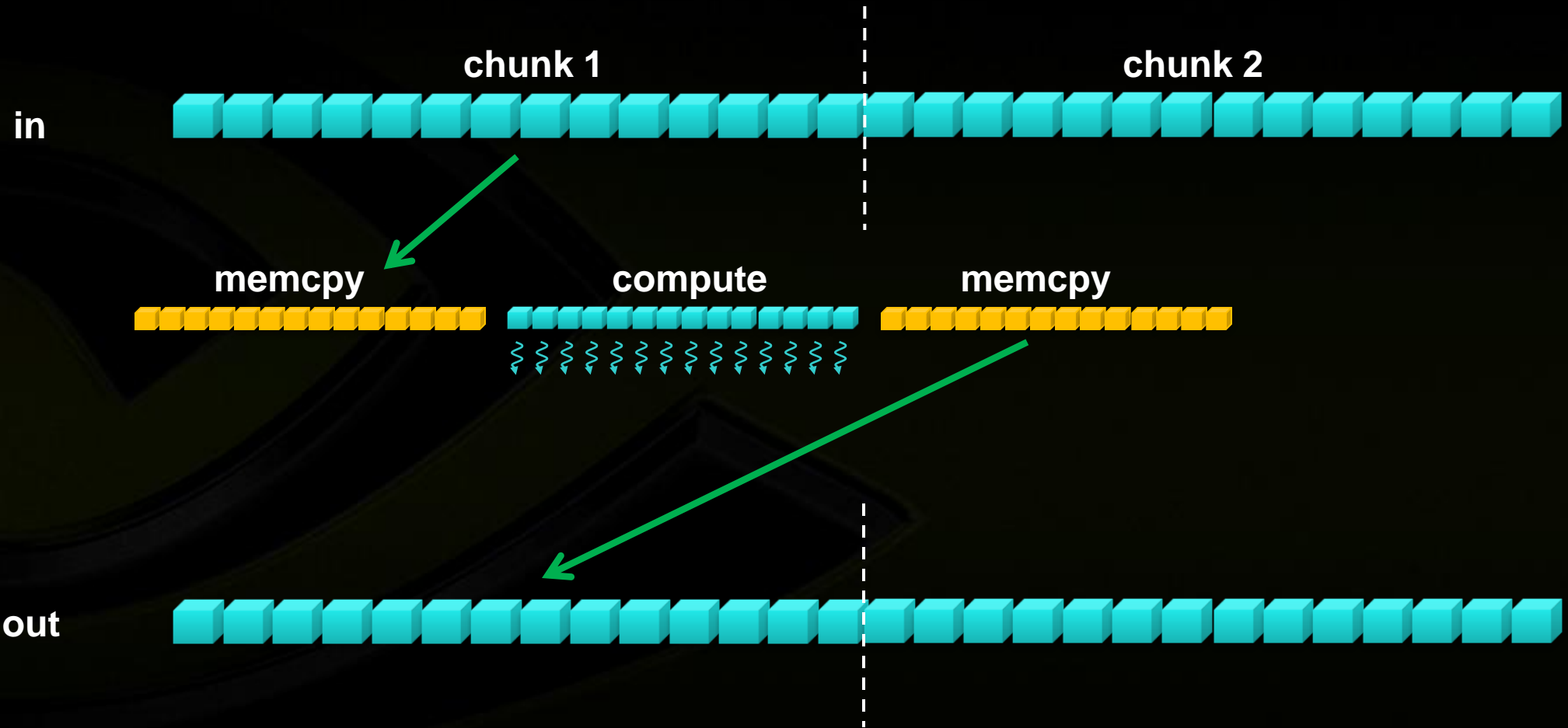
Data transfers between the host and the device using `cudaMemcpy()` are blocking transfers; that is, control is returned to the host thread only after the data transfer is complete. The `cudaMemcpyAsync()` function is a non-blocking variant of `cudaMemcpy()` in which control is returned immediately to 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

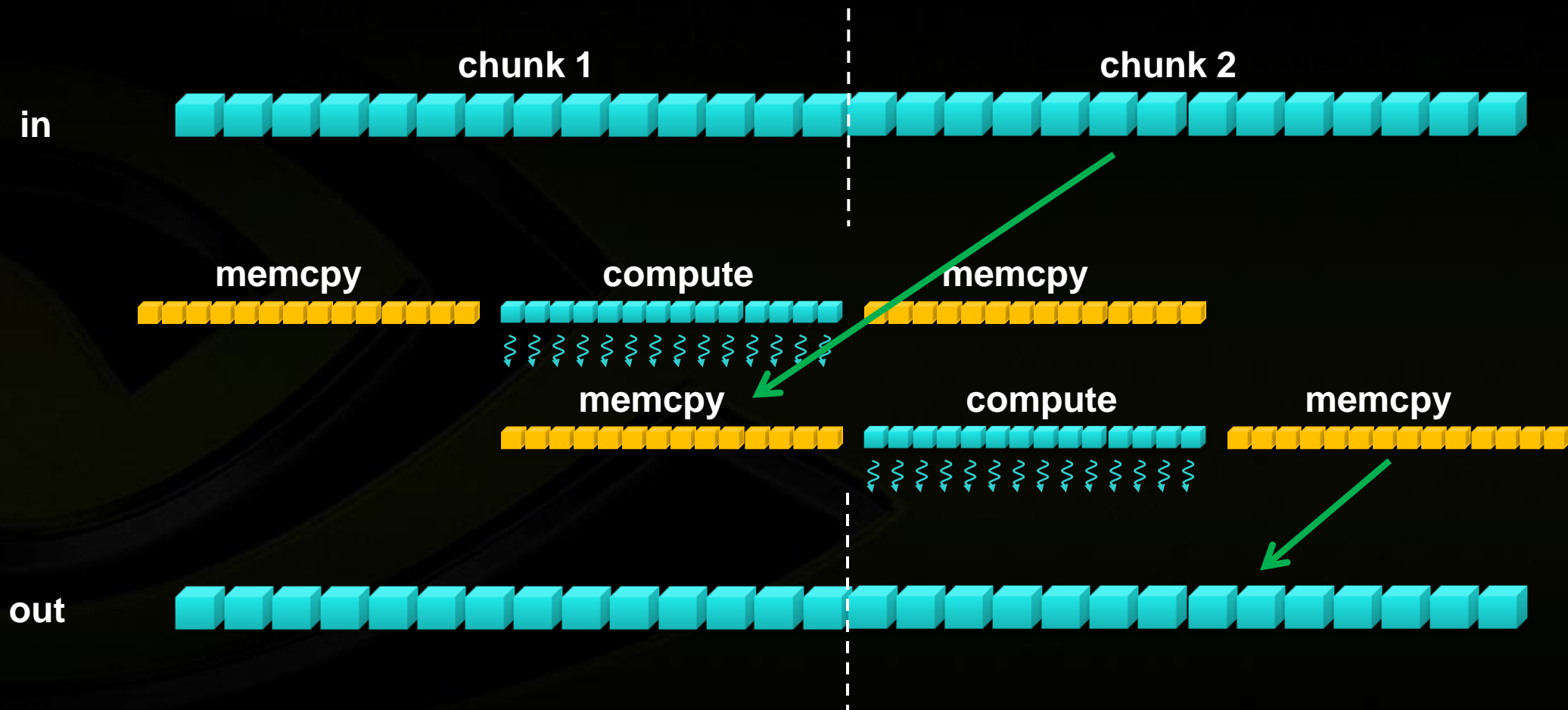
Data Partitioning Example



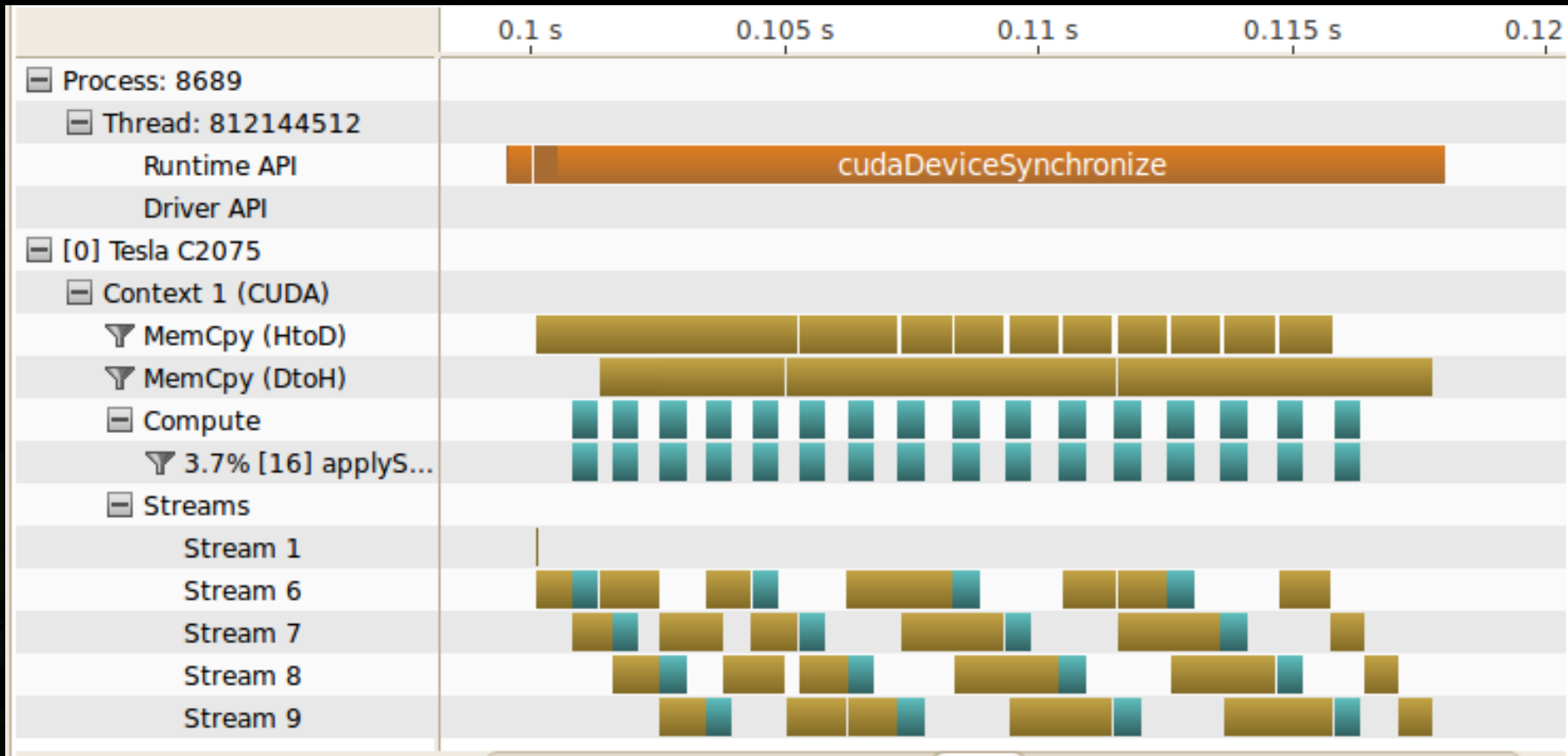
Data Partitioning Example



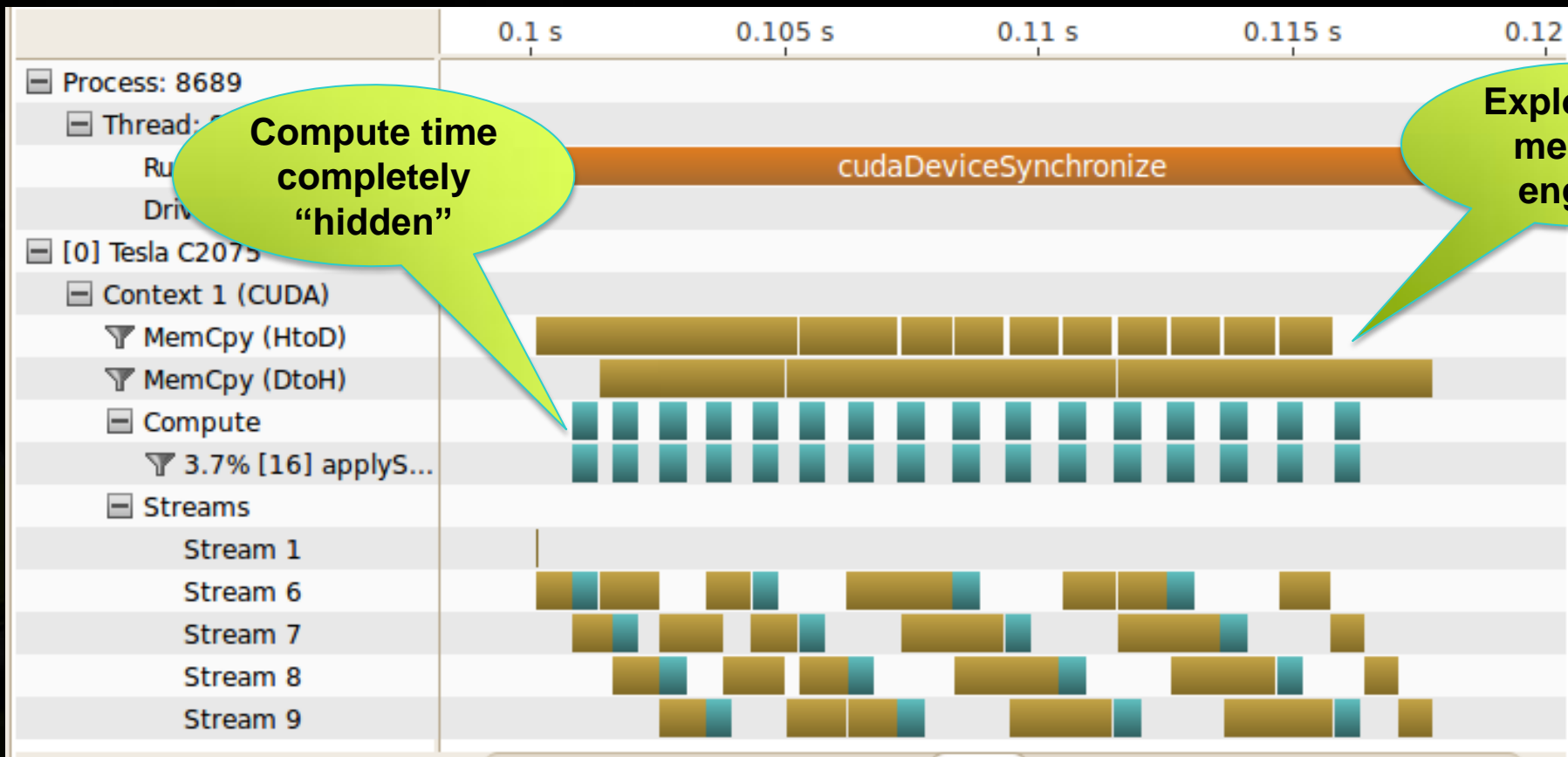
Data Partitioning Example



Overlapped Compute/Memcpy



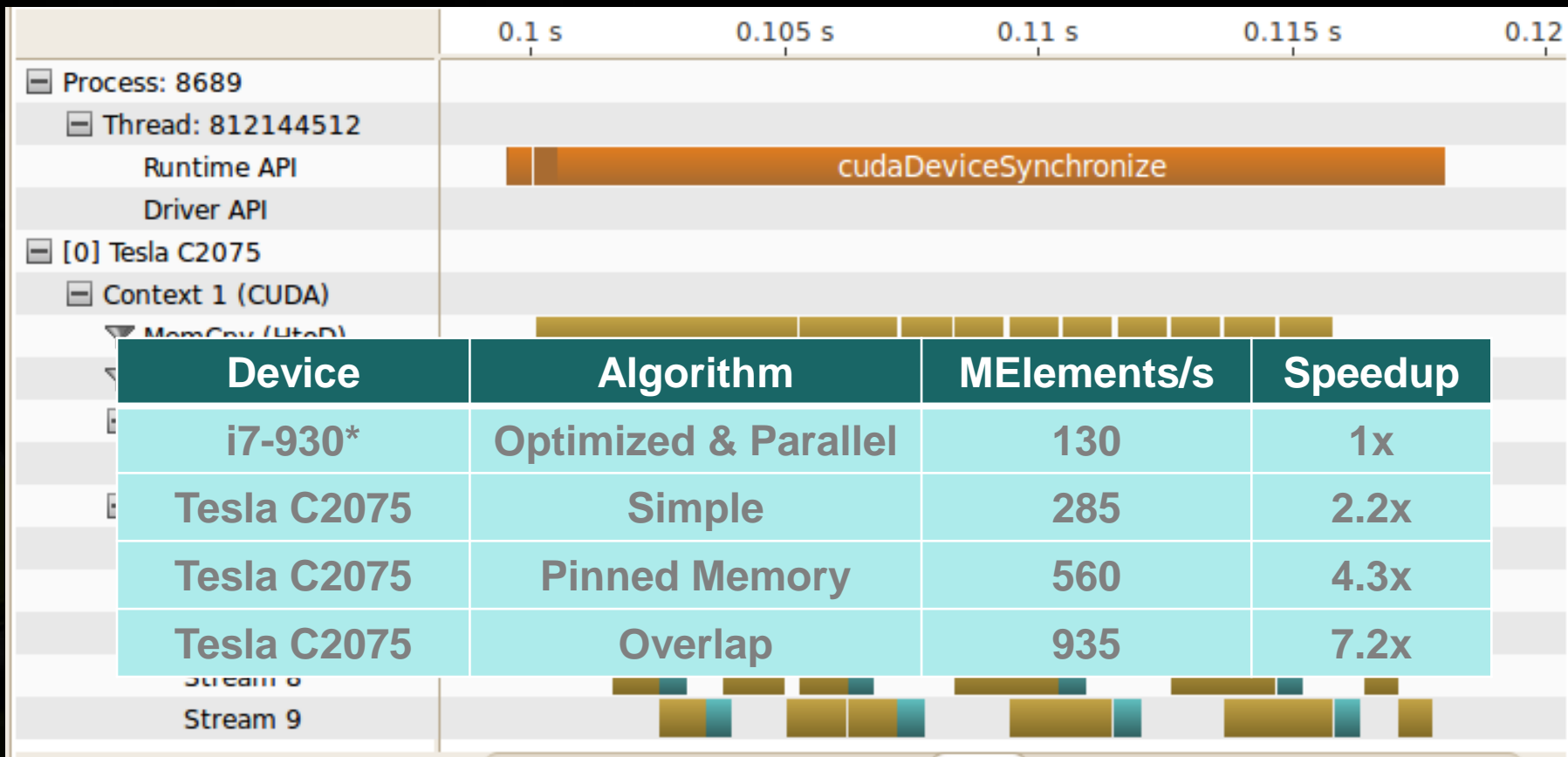
Overlapped Compute/Memcpy



Compute time completely "hidden"

Exploit dual memcpy engines

Overlapped Compute/Memcpy Result



*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

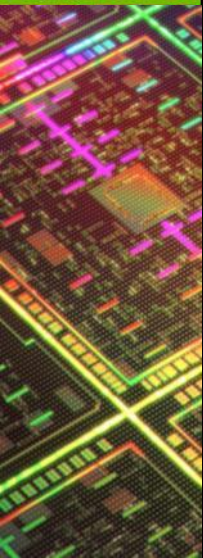


Visual Profiler Demo

Summary

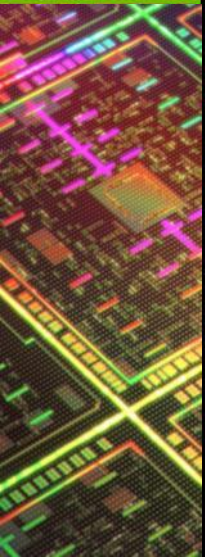
- CUDA accelerates compute-intensive parts of your application
- Visual profiler helps in performance analysis and optimization
- Get Started
 - **Download** free CUDA Toolkit: www.nvidia.com/getcuda
 - **Join** the community: developer.nvidia.com/join

Questions?



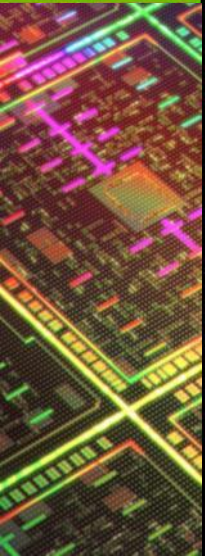
Performance optimization strategies

- 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