

**Application Optimization
Using CUDA
Development Tools**



Optimization: CPU and GPU

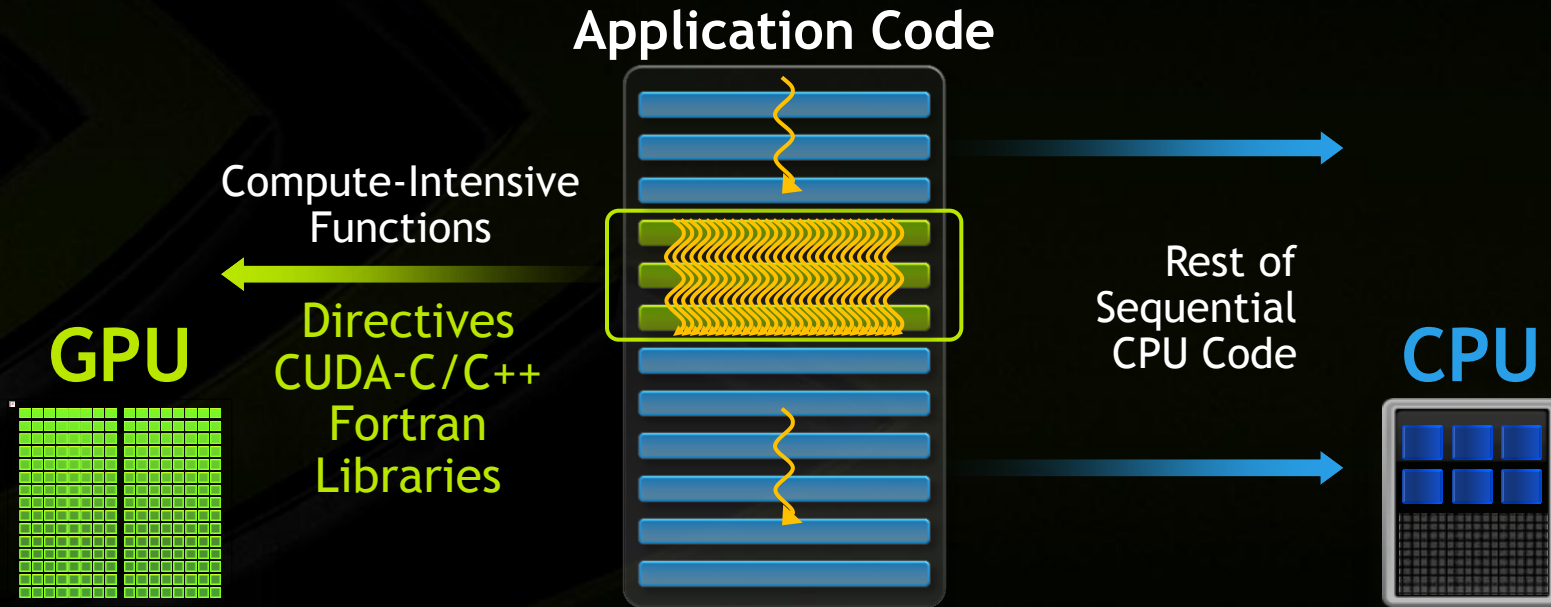


- A few cores
- Good memory bandwidth
- Best at serial execution

- 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



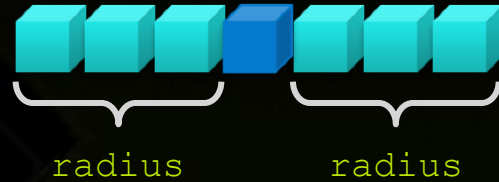
Application Optimization Process and Tools



- **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

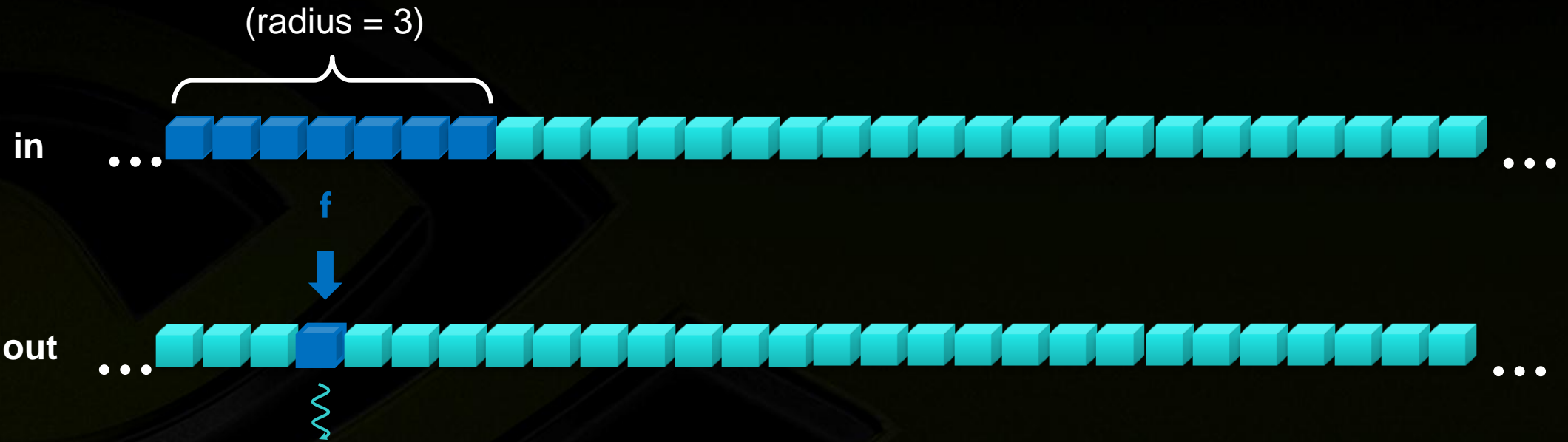


- Fundamental to many algorithms
 - Standard discretization methods, interpolation, convolution, filtering
- Our example will use weighted arithmetic mean

Serial Algorithm



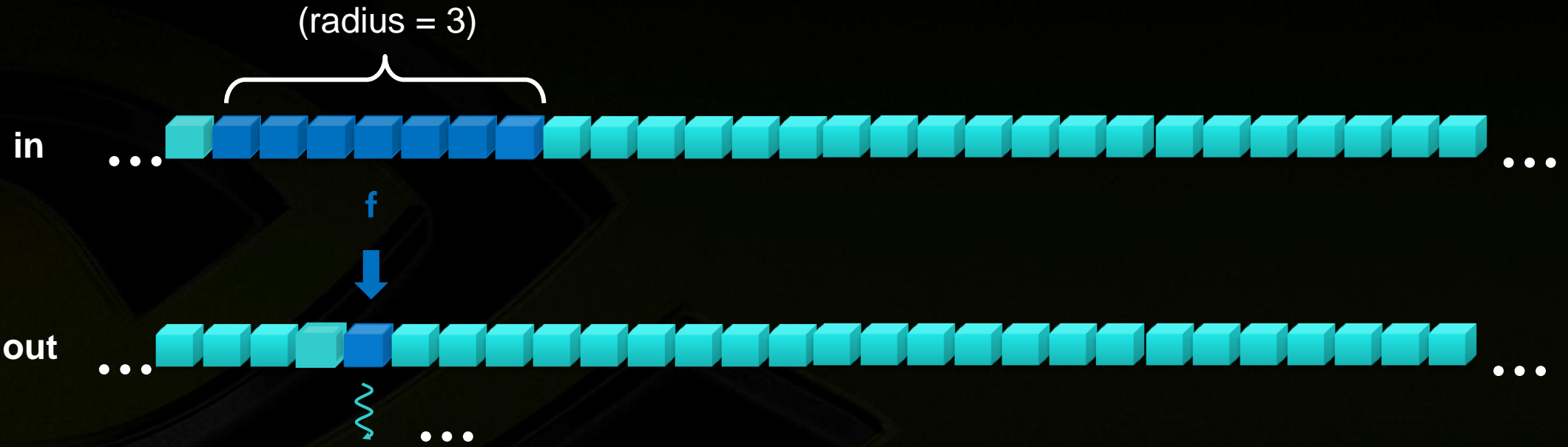
⋈ = Thread



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, 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);
    }
}
```

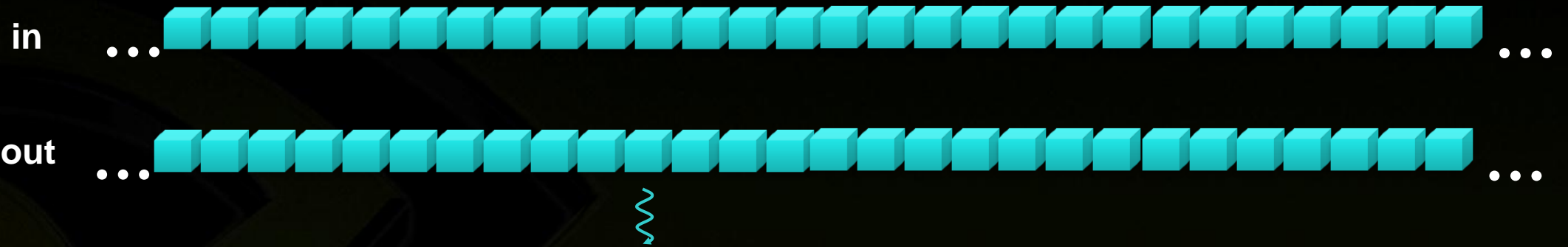
CPU	MElements/s
i7-930	30

Parallel Algorithm

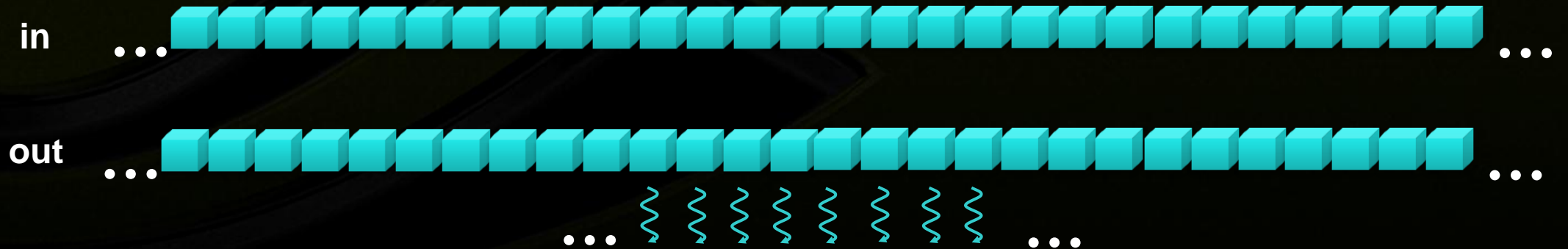


⚡ = Thread

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, 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);
    }
}
```


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);
}
```

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;
    int x = 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);
}
}
```

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;
        //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

Copy results from 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);
}
```

Indicates GPU kernel

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(size);
    float *in = (float *)malloc(wsize);
    float *out= (float *)malloc(wsize);
    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);
}
```

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

Functional Correctness



- **But our first run returns an error!**

```
$ 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 0x00000240 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
```

Bad Instruction

Error Location

Bad Address

Bad Thread

Error Type

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


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
```



```
__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);
    }
}
```



Found the
bad array
access

Debugger: cuda-gdb



Switch to the CPU thread

Switch to the frame where the allocation occurred

```
(cuda-gdb) thread 1  
  
(cuda-gdb) info stack  
[...]  
#10 0x0000000000400e86 in main  
  
(cuda-gdb) frame 10  
#10 0x0000000000400e86 in main  
  
(cuda-gdb) print wsize / 4  
31  
  
(cuda-gdb) print size / 4  
16777216
```

```
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, ...);
```

Found bad allocation size

Corrected Parallel 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);
    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

```
117 __global__ void applyStencil1D(int startIdx, int endIdx, const float *weights,
118                               float *in, float *out, size_t N, int weights_size) {
119     int i = startIdx + blockIdx.x * blockDim.x + threadIdx.x;
120
121     if (i < endIdx) {
122         float result = 0;
123         for (int j = -RADIUS; j <= RADIUS; j++) {
124             result += weights[j + RADIUS] * in[i + j];
125         }
126         out[i] = result / (2 * RADIUS + 1);
127     }
128 }
```

Detailed information

CUDA Memory Checker detected 512 threads caused an access violation:

Launch Parameters

- CUcontext = 0ec26910
- CUstream = 0f3ba6a8
- CUmodule = 10bed300
- CUfunction = 10bf8bf8
- FunctionName = _Z14applyStencil1D1iPKFPFS1_ji
- gridDim = {32768,1,1}
- blockDim = {512,1,1}
- sharedSize = 0

Parameters:

- N = Unsupported read from GridLaunch scope.
- weights_size = Unsupported read from GridLaunch scope.
- startIdx = 15
- endIdx = 16777201
- weights = 0x055a0000 0.0012512589
- in = 0x055a0200 0.66304511
- out = 0x055a0400 -7.9388899e-13

Parameters (raw):

```
0x0000000f 0x00fffff1 0x055a0000 0x055a0200
0x055a0400 0x01000000 0x0000007c
```

GPU State:

Address	Size	Type	Mem	Block	Thread	blockIdx	threadIdx	PC	Source
055a0680	63873	adr 1d		0	288	{0,0,0}	{288,0,0}	0001d0	c:\temp\stencil\stencil1d.cu:124
055a0684	63873	adr 1d		0	289	{0,0,0}	{289,0,0}	0001d0	c:\temp\stencil\stencil1d.cu:124
055a0688	63873	adr 1d		0	290	{0,0,0}	{290,0,0}	0001d0	c:\temp\stencil\stencil1d.cu:124
055a068c	63873	adr 1d		0	291	{0,0,0}	{291,0,0}	0001d0	c:\temp\stencil\stencil1d.cu:124

High level message of the access violation

Parallel Nsight Debug

Memory Checker detected 512 access violations.
error = access violation on load
blockIdx = {0,0,0}
threadIdx = {288,0,0}
address = 0x055a0680
accessSize = 63873
Please see the output window for details.

Parallel Nsight for Visual Studio



stencil_vc100 (Debugging) - Microsoft Visual Studio (Administrator)

```
117 __global__ void applyStencilID(int startIdx, int endIdx, const float *weights,
118                               float *in, float *out, size_t N, int weights_size) {
119     int i = startIdx + blockIdx.x*blockDim.x+threadIdx.x;
120
121     if (i < endIdx) {
122         float result = 0;
123         for (int j = -RADIUS; j <= RADIUS; j++) {
124             result += weights[j+RADIUS] * in[i+j];
125         }
126         out[i] = result / (2 * RADIUS + 1);
127     }
```

cuda-nsight

Warp 7

Current	Cucontext	Grid ID	blockIdx	Warp Index	threadIdx	PC	Active Mask	Status	Exception	Exception Details	Lanes
	0x00376910	10	(0, 0, 0)	0	(0, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	1	(32, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	2	(64, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	3	(96, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	4	(128, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	5	(160, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	6	(192, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
→	0x00376910	10	(0, 0, 0)	7	(224, 0, 0)	0x0001b2d0	0xffffffff	! Exception	OutOfRangeException	MemorySpace=5 Size	█
	0x00376910	10	(0, 0, 0)	8	(256, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	9	(288, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	10	(320, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	11	(352, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	12	(384, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	13	(416, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	14	(448, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	15	(480, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	16	(512, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	17	(544, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	18	(576, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	19	(608, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	20	(640, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	21	(672, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	22	(704, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	23	(736, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	24	(768, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	25	(800, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	26	(832, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	27	(864, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	28	(896, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	29	(928, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	30	(960, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	31	(992, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	32	(1024, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	33	(1056, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	34	(1088, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	35	(1120, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	36	(1152, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	37	(1184, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	38	(1216, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	39	(1248, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	40	(1280, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	41	(1312, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	42	(1344, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	43	(1376, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	44	(1408, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	45	(1440, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	46	(1472, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	47	(1504, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	48	(1536, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	49	(1568, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	50	(1600, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	51	(1632, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	52	(1664, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	53	(1696, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	54	(1728, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	55	(1760, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	56	(1792, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	57	(1824, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	58	(1856, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	59	(1888, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	60	(1920, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	61	(1952, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	62	(1984, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	63	(2016, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	64	(2048, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	65	(2080, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	66	(2112, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	67	(2144, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	68	(2176, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	69	(2208, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	70	(2240, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	71	(2272, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	72	(2304, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	73	(2336, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	74	(2368, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	75	(2400, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	76	(2432, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	77	(2464, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	78	(2496, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	79	(2528, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	80	(2560, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	81	(2592, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	82	(2624, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	83	(2656, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	84	(2688, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	85	(2720, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	86	(2752, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	87	(2784, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	88	(2816, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	89	(2848, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	90	(2880, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	91	(2912, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	92	(2944, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	93	(2976, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	94	(3008, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	95	(3040, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	96	(3072, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	97	(3104, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	98	(3136, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	99	(3168, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	100	(3200, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	101	(3232, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	102	(3264, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	103	(3296, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	104	(3328, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	105	(3360, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x00376910	10	(0, 0, 0)	106	(3392, 0, 0)	0x0001b2d0	0xffffffff	None	None	None	█
	0x003769										

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
    (
    cudaMemcpy(out,
    //free resource
    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

- Commonly used for debugging, available on GPU

```
__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



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_example/stencil/run_gpu  
GPU: 0.058926 seconds, 2.27773 GBytes/s, 0.284716 GElements/s
```

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 ...)

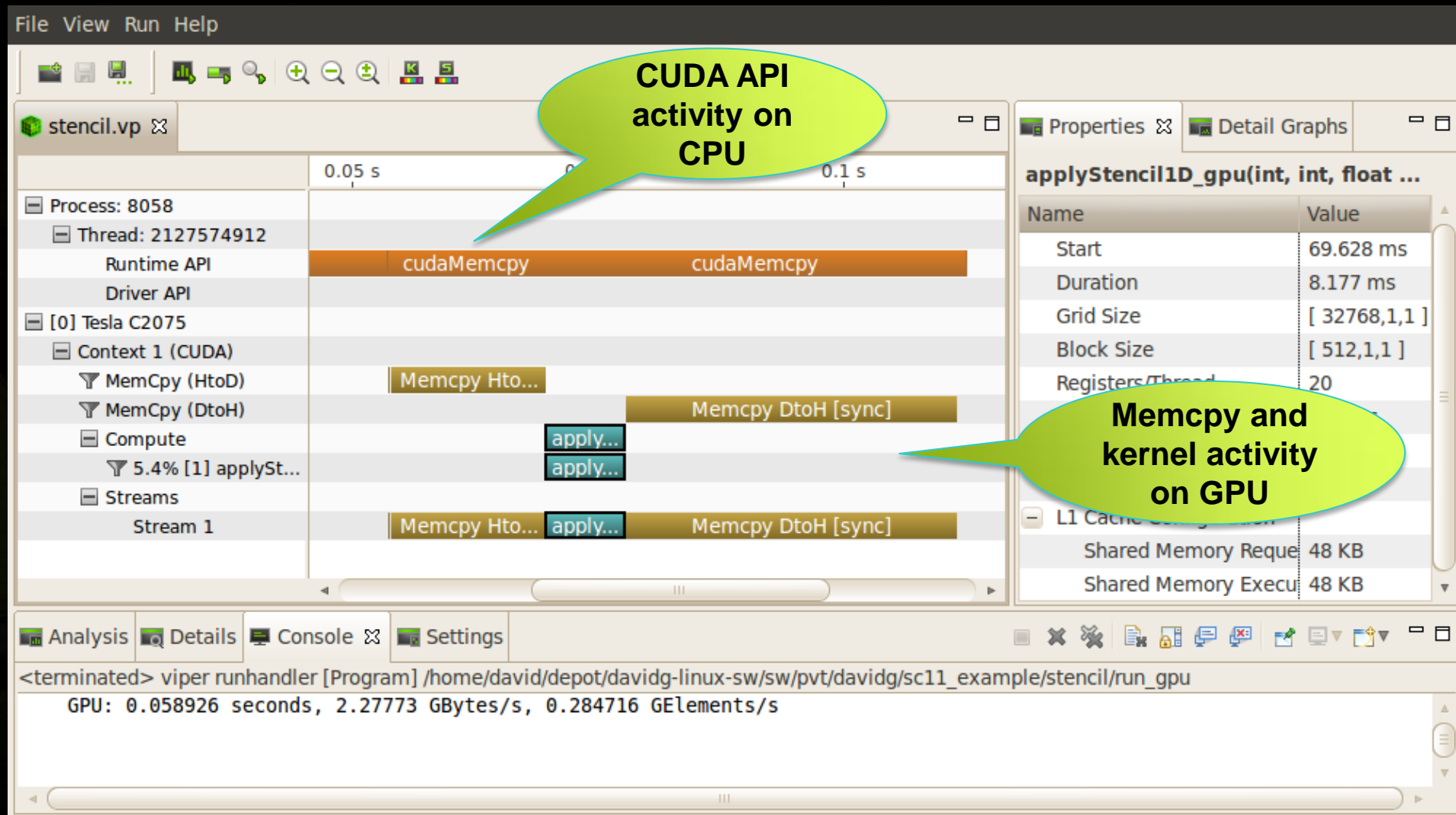
Name	Value
Start	69.628 ms
Duration	8.177 ms
Grid Size	[32768,1,1]
Block Size	[512,1,1]
Registers/Thread	20
Shared Memory/Block	0 bytes
Occupancy	
Theoretical	100%
L1 Cache Configuration	
Shared Memory Reque	48 KB
Shared Memory Execu	48 KB

Analysis Details Console Settings

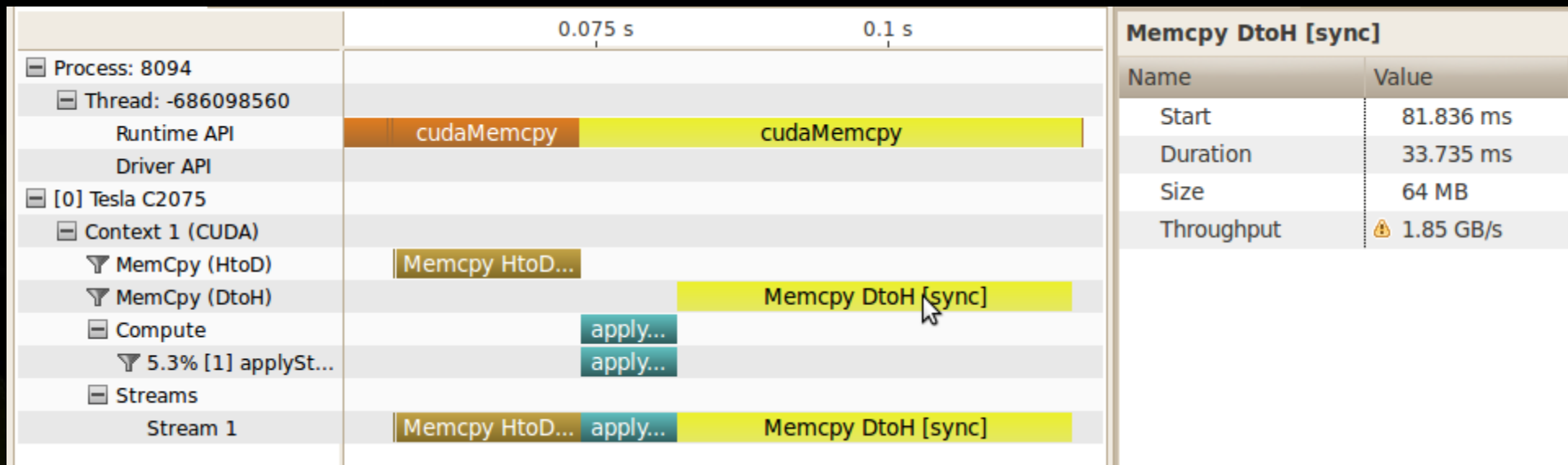
<terminated> viper runhandler [Program] /home/david/depot/davidg-linux-sw/sw/pvt/davidg/sc11_exam...
GPU: 0.058926 seconds, 2.27773 GBytes/s, 0.284716 GElements/s

Kernel and memcpy details

NVIDIA Visual Profiler



Detecting Low Memory Throughput



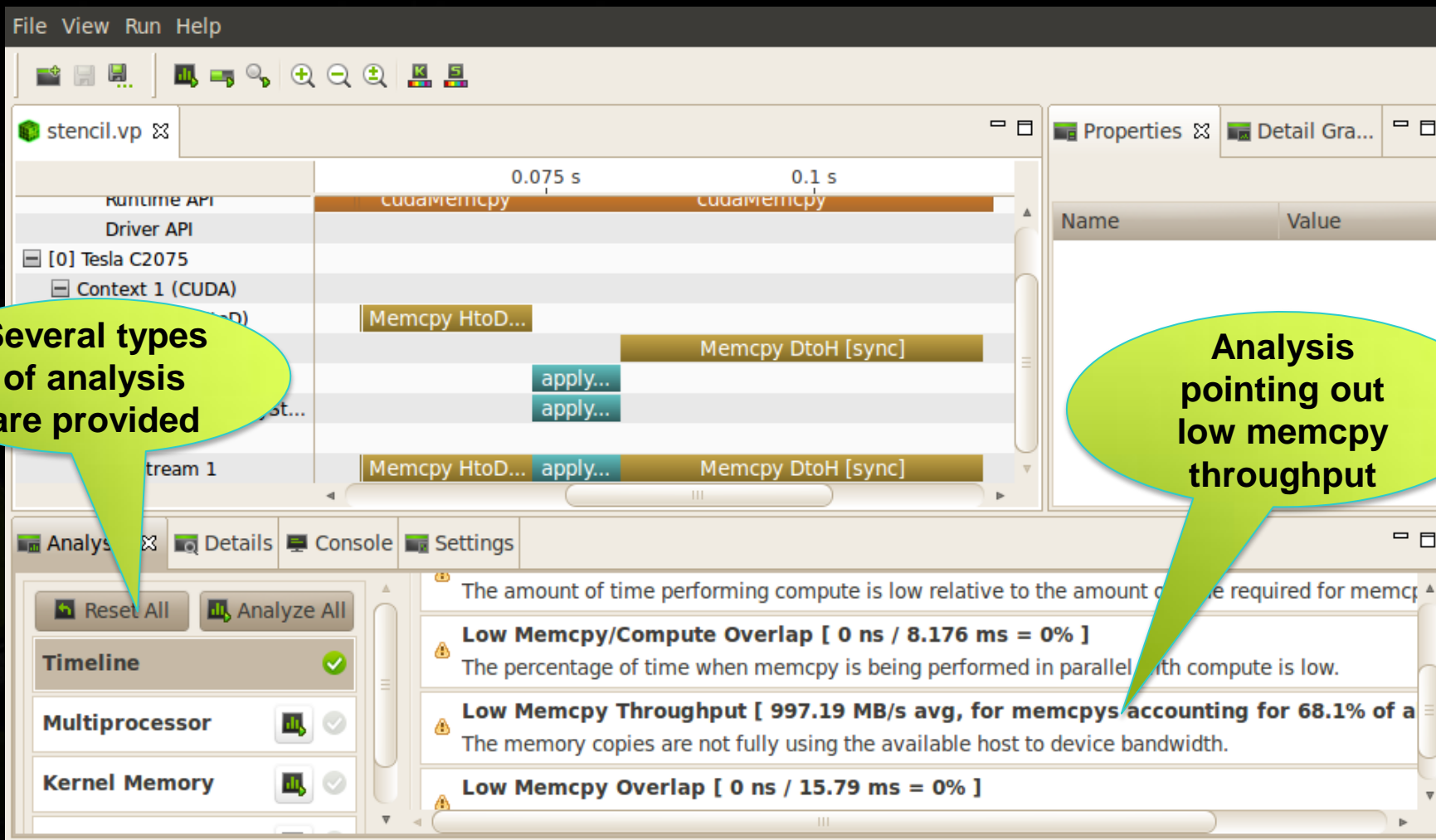
- Spend majority of time in data transfer
 - Often can be overlapped with preceding or following computation
- From timeline can see that throughput is low
 - PCIe x16 can sustain > 5GB/s

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 menu 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

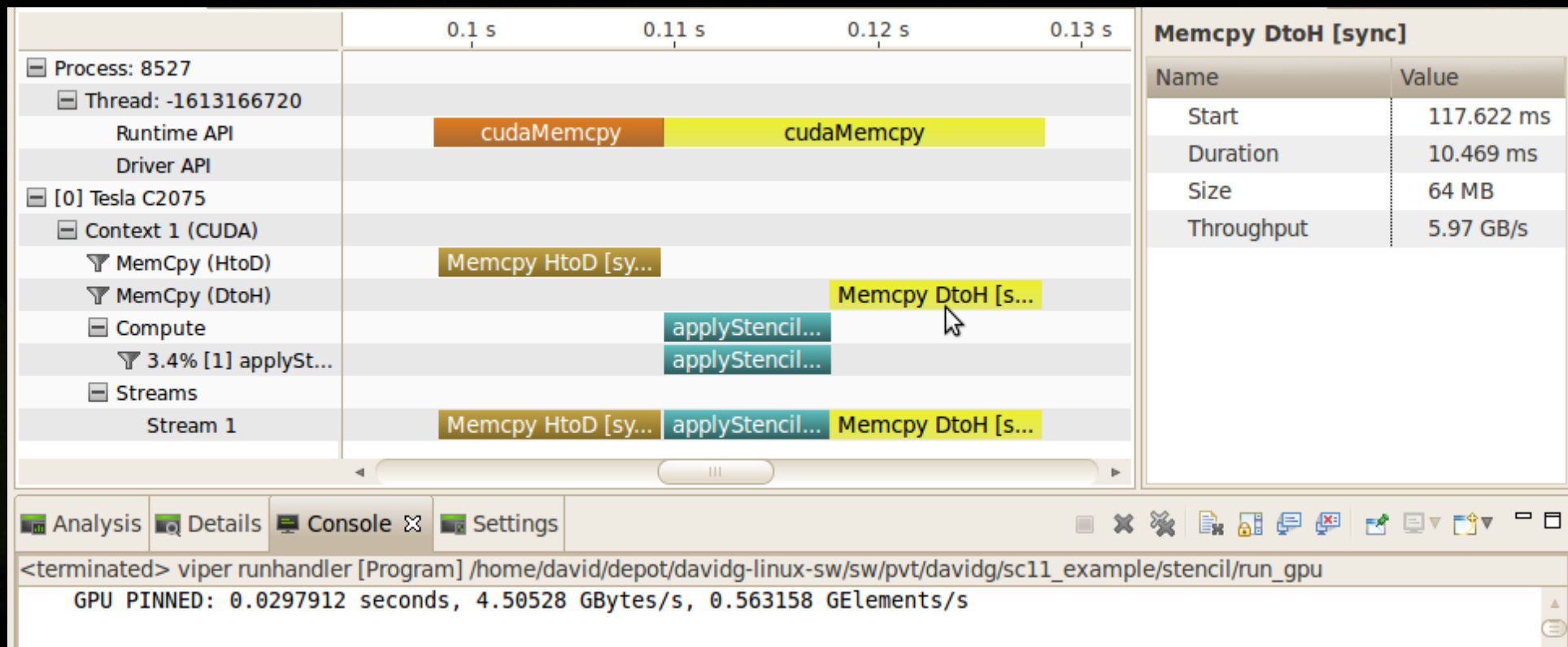
Pinned CPU Memory Implementation

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

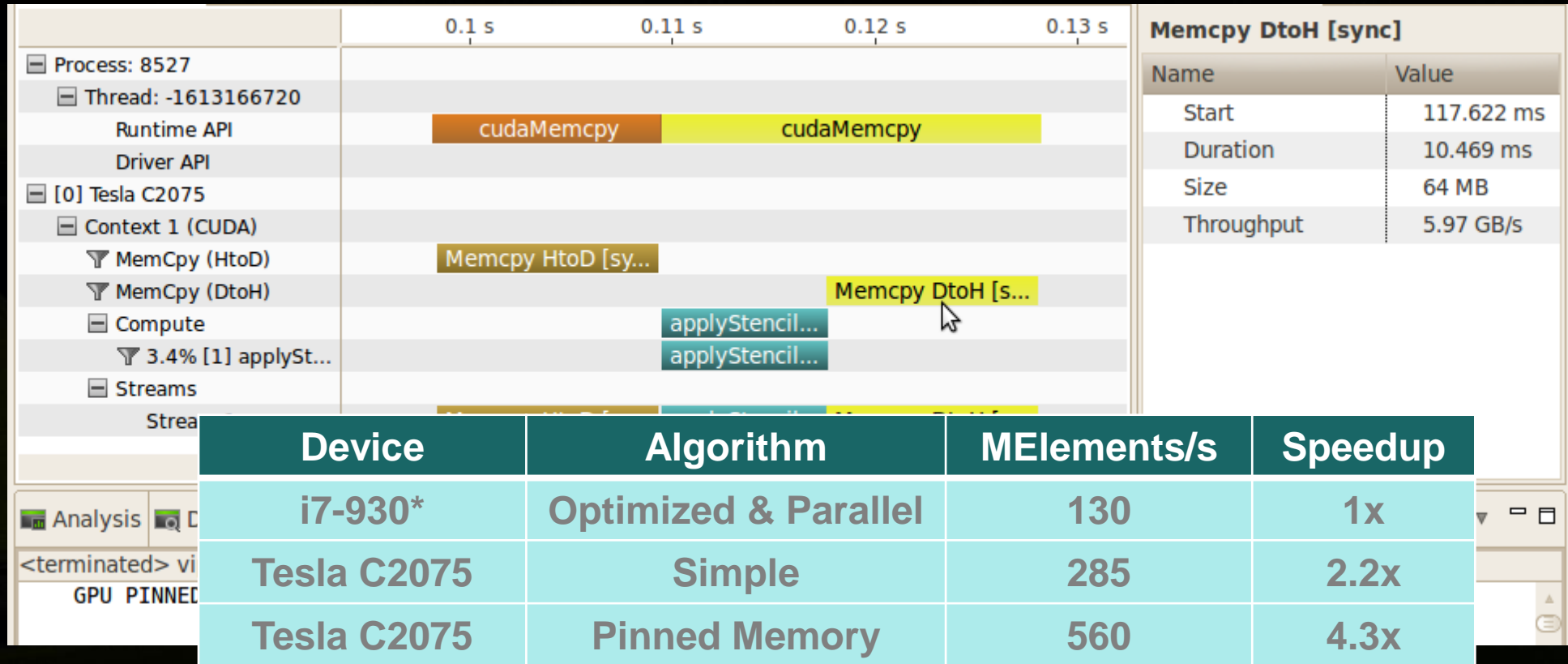
CPU allocations
use pinned
memory to enable
fast memcpy

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)



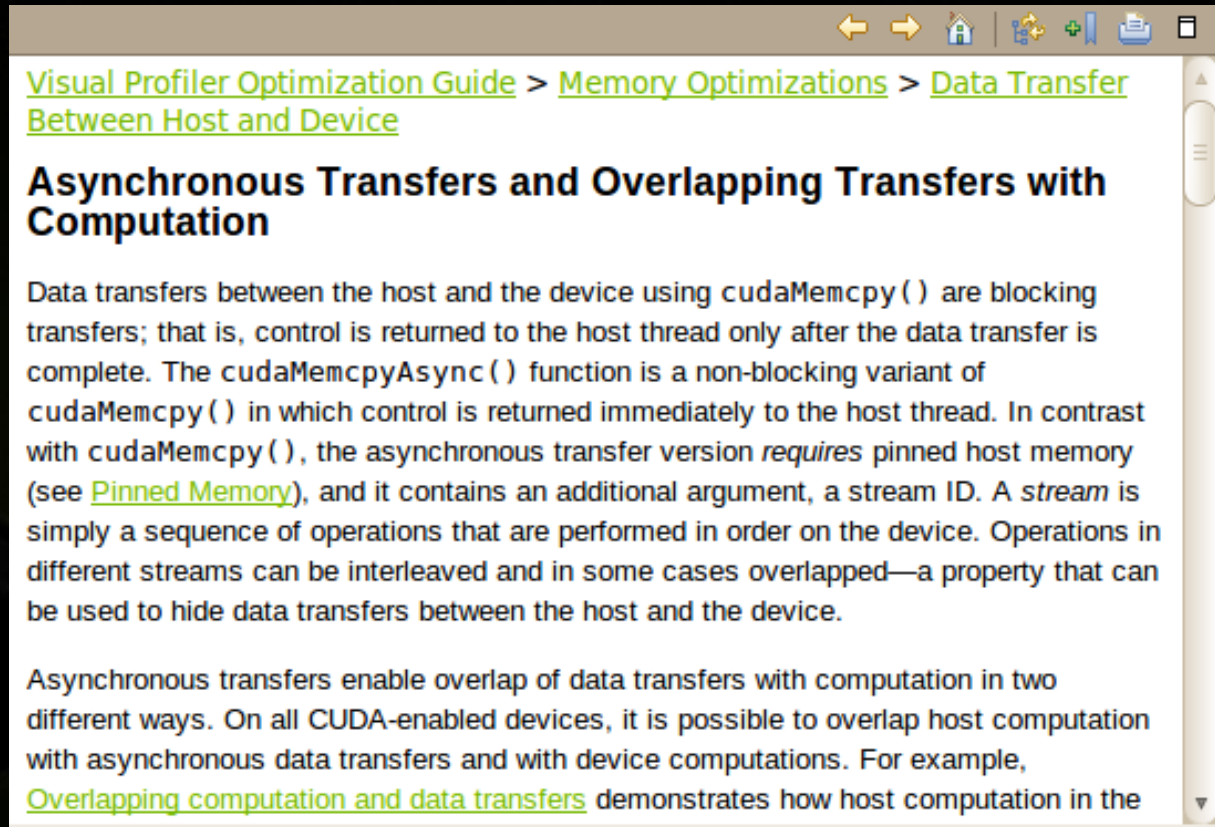
Low Memcpy/Compute Overlap [0 ns / 8.176 ms = 0%]



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

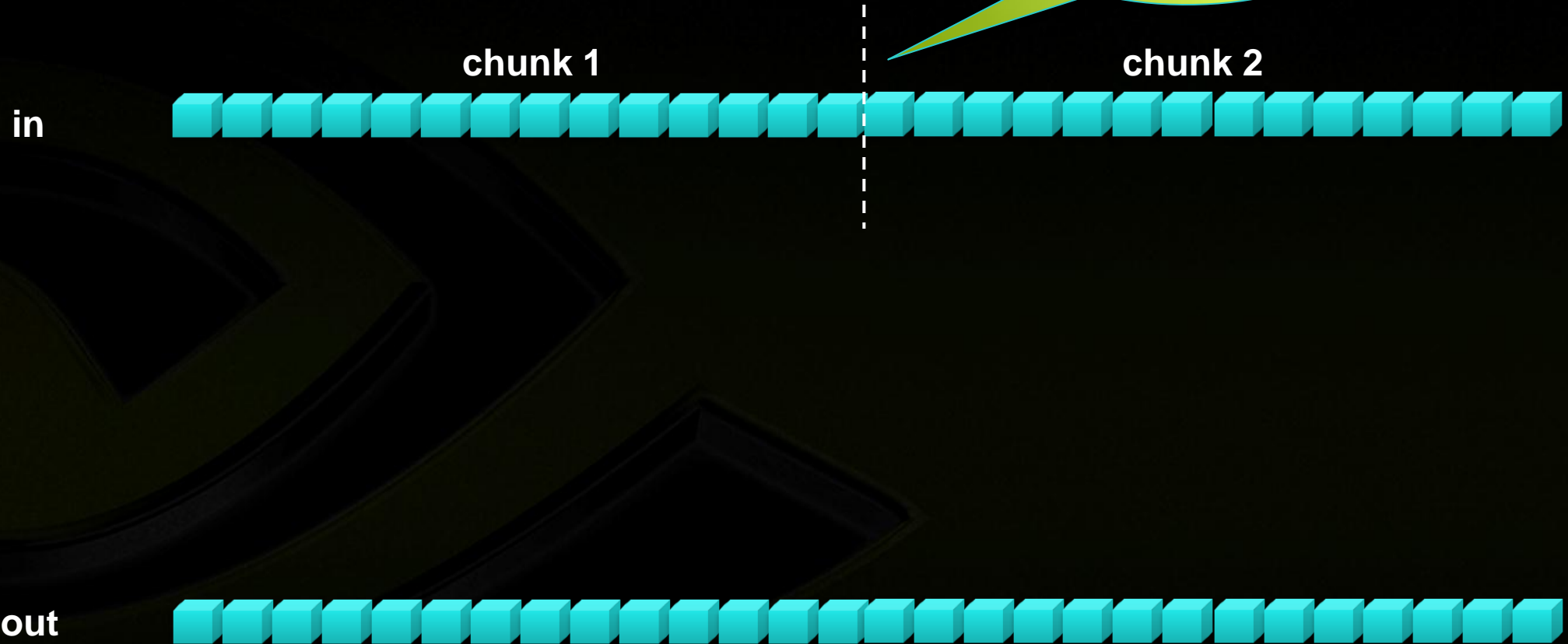
[More..](#)

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

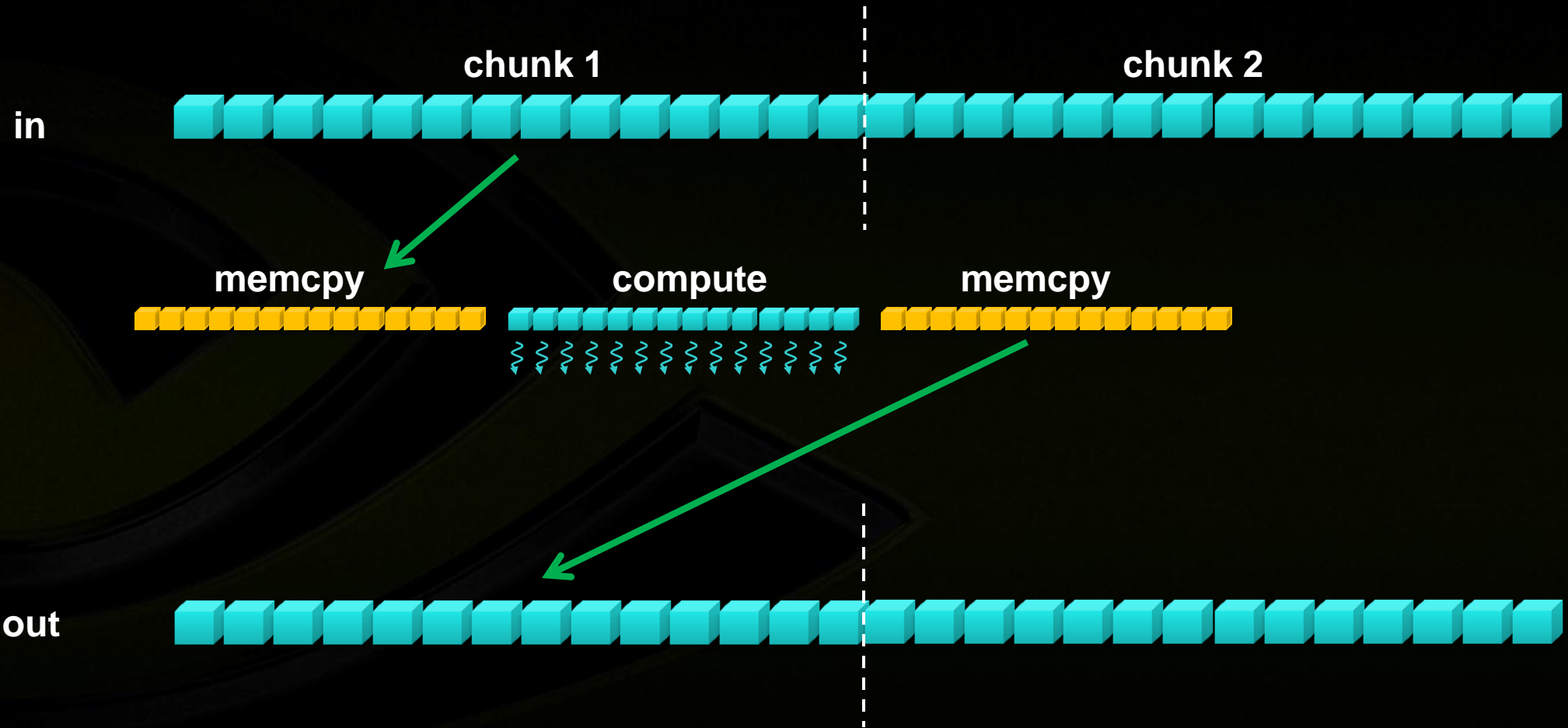


The screenshot shows a web page from the Visual Profiler Optimization Guide. The breadcrumb navigation is: [Visual Profiler Optimization Guide](#) > [Memory Optimizations](#) > [Data Transfer Between Host and Device](#). The main heading is **Asynchronous Transfers and Overlapping Transfers with Computation**. The text explains that `cudaMemcpy()` is a blocking function, while `cudaMemcpyAsync()` is non-blocking. It notes that asynchronous transfers require pinned host memory and a stream ID. The text concludes by stating that asynchronous transfers enable overlap of data transfers with computation in two ways: overlapping host computation with asynchronous data transfers, and overlapping device computation with asynchronous data transfers. A link to [Overlapping computation and data transfers](#) is provided.

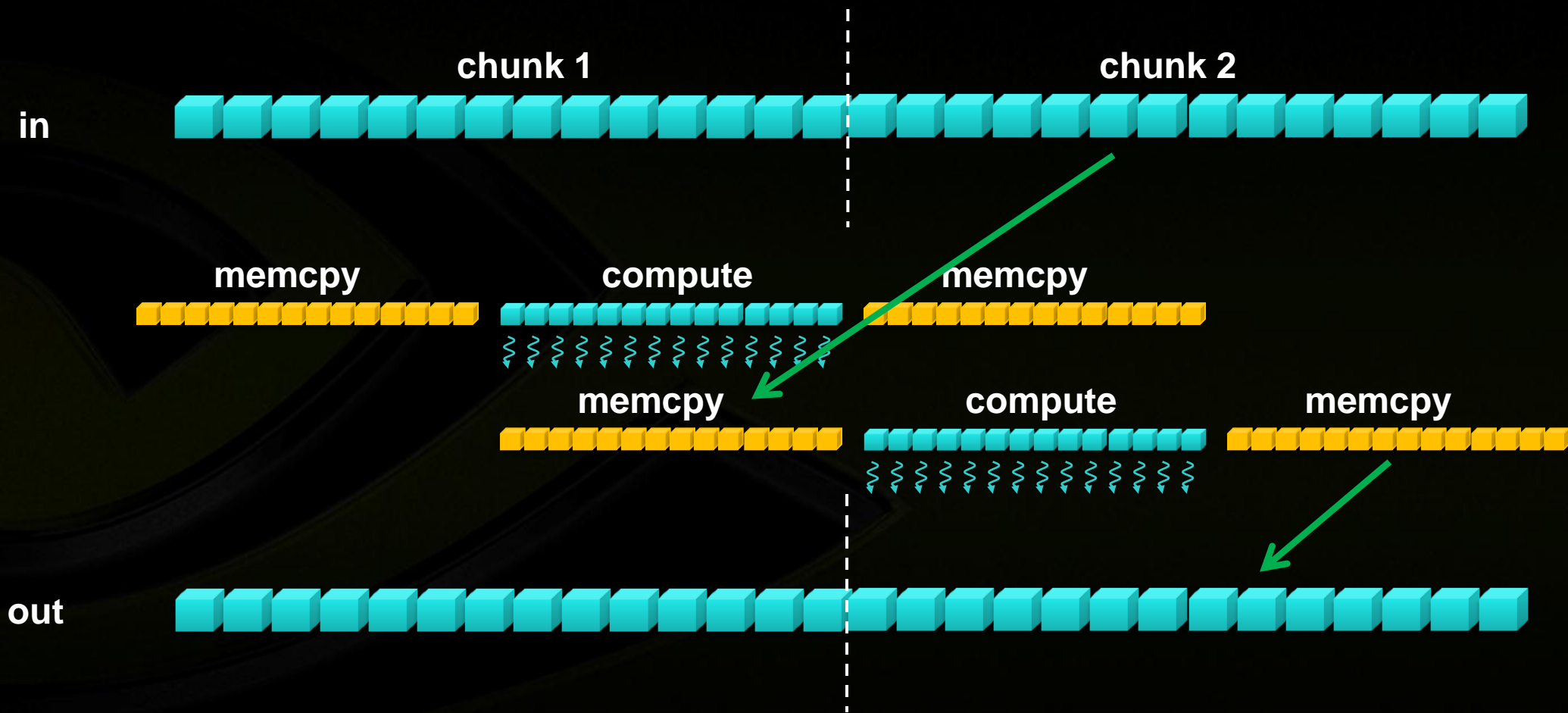
Data Partitioning Example



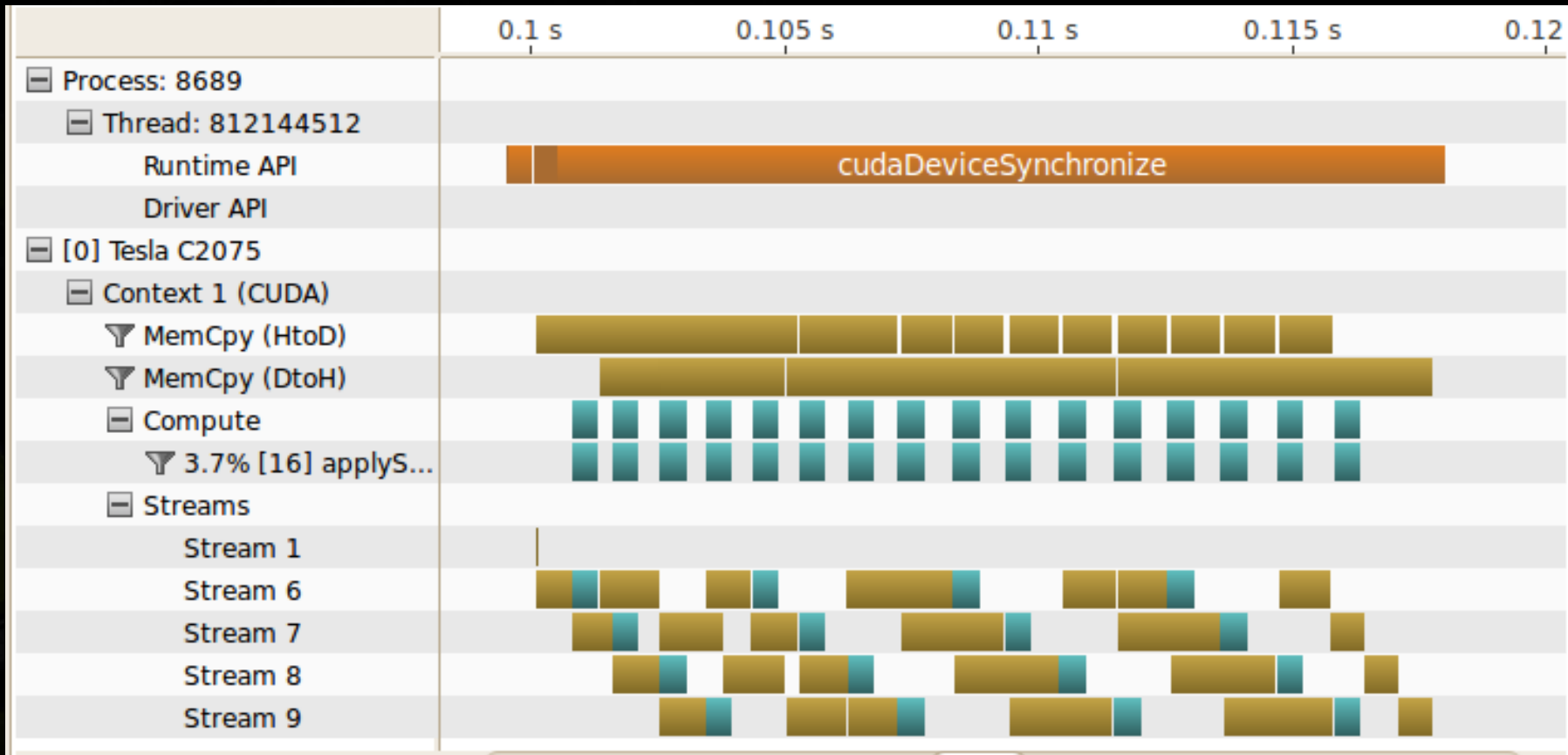
Data Partitioning Example



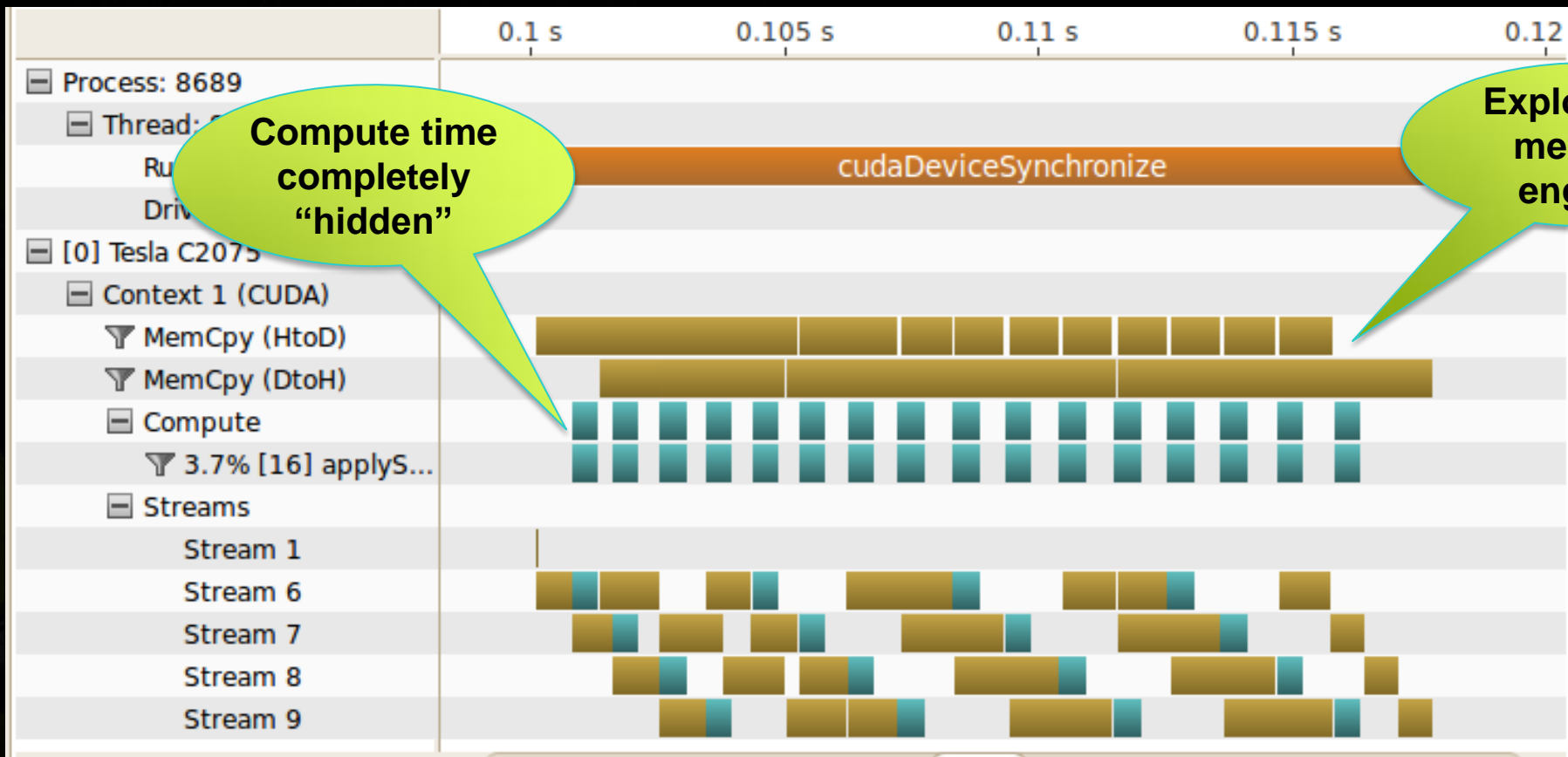
Data Partitioning Example



Overlapped Compute/Memcpy



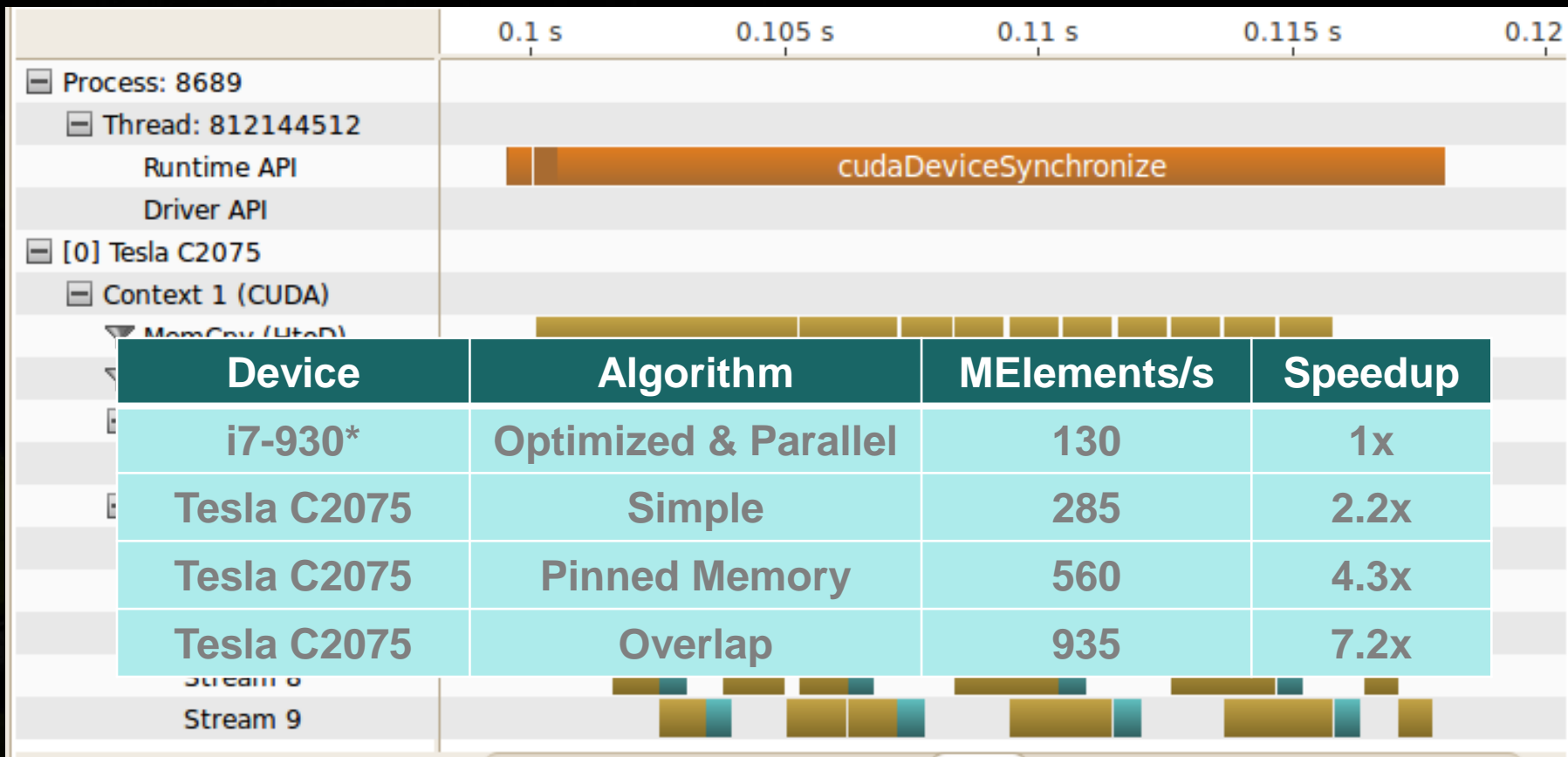
Overlapped Compute/Memcpy



Compute time completely "hidden"

Exploit dual memcpy engines

Overlapped Compute/Memcpy Result



*4 cores + hyperthreading

Parallel Nsight For Visual Studio



The screenshot displays the Parallel Nsight interface for Visual Studio. The top window shows a timeline of CPU and GPU activity. The CPU activity includes `cudaDeviceSynchronize [4564]` and `cuCtxSynchronize [4564]`. The GPU activity shows a `1.4 % [18] applyStencil...` kernel launch. The bottom window shows the correlation details for the selected kernel launch.

Timeline CPU & GPU

Correlation Hierarchy

- 1.4 % [18] applyStencil1D [CUDA Device Function Call Row]
- Cursor Information
 - applyStencil1D [CUDA Launch]
 - cuLaunchKernel [CUDA Function Call]
 - 1 [Context]
 - 4564 [Thread]
 - Stack Trace
 - cuLaunchKernel [CUDA Kernel]

Actions	Address	Module Name	Function	File Name	Line
	0x00D8C628	nvidia.dll	cuLaunchKernel		
	0x10003E39	cuda32_41_15.dll			
	0x10021CE9	cuda32_41_15.dll			
	0x0138842C	stencil1d.exe	enum cudaError __cdecl cudaLaunch<char>(char *)	c:\program files\NVIDIA Corporation\NVIDIA Computing Tools SDK\bin\win32_x86\bin\stencil1d.exe	
	0x01381E26	stencil1d.exe	void __cdecl __device_stub_Z14applyStencil1DiiPKFPFS1_(int,int,float const *,float *,float *)	c:\users\mstrengert\nvidia\bin\win32_x86\bin\stencil1d.exe	
	0x01381E4C	stencil1d.exe	void __cdecl applyStencil1D(int,int,float const *,float *,float *)	c:\share_presentation\stencil1d.exe	
	0x01381C83	stencil1d.exe	gpu_overlap	c:\share_presentation\stencil1d.exe	
	0x01381711	stencil1d.exe	_main	c:\share_presentation\stencil1d.exe	
	0x01389CFD	stencil1d.exe	_tmainCRTStartup	c:\share_presentation\stencil1d.exe	
	0x76EF93677	kernel32.dll		f:\dd\vctools\crt_bld\self_x86\crt\src\kernel32.dll	
	0x778D9F02	ntdll.dll			

Correlation Details

Parallel Nsight For Visual Studio



stencil1d111108_015_Capture_000.nvreport - Microsoft Visual Studio (Administrator) - Experimental Instance

File Edit View Debug Team Nsight Data Tools Test Window Help

hypot

stencil1d111108_01...apture_000.nvreport

Filter Viewing: 18 / 18

Drag a column header and d

Instruction, Branch, Memory and Other Analysis

Function Name	Dynamic Shared Memory per Block (bytes)	Local Memory per Thread (bytes)	Local Memory (bytes)	Cache Configuration Executed	Grid Dimensions	Block Dimensions	API Call ID
1 applyStencil1D	0	0	36	44433408	PREFER_SHARED	{32768, 1, 1}	{512, 1, 1}
2 applyStencil1D	0	0	72	45088768	PREFER_SHARED	{32768, 1, 1}	{512, 1, 1}

applyStencil1D [CUDA Launch]

- applyStencil1D [CUDA Kernel]
 - Experiment Results
 - CUDA Occupancy
 - CUDA Instruction Statistics
 - CUDA Branch Statistics
 - CUDA Issue Efficiency
 - CUDA Achieved Flops
 - CUDA Memory Statistics

View By: Size

Name	Total	Per Warp	Per Second
Global			
Requests	33,030,140.00	63.00	1,321,830,000.00
Loads	32,505,860.00	62.00	1,300,849,000.00
Stores	524,288.00	1.00	20,981,430.00
Transactions	49,283,040.00	94.00	1,972,253,000.00
Loads	48,234,460.00	92.00	1,930,290,000.00
Stores	1,048,574.00	2.00	41,962,780.00
Size	5.87 GB	11.75 kB	235.11 GB/s
Loads	5.75 GB	11.50 kB	230.11 GB/s
Stores	128.00 MB	256.00 B	5.00 GB/s
Replay Overhead	0.16 %		
Local			
Requests	0.00	0.00	0.00

Overview Global Local Atomics Shared Texture Caches Buffers

Ready

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**
 - **Download** free CUDA Toolkit: www.nvidia.com/getcuda
 - **Join** the community: 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

Questions?

