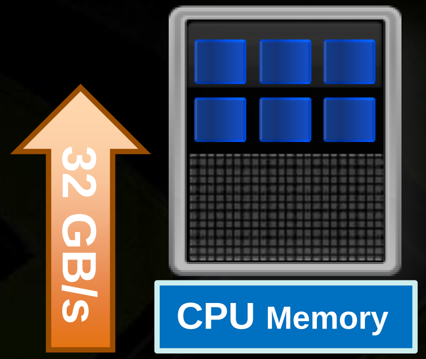
Application Optimization Using CUDA Development Tools

NVIDIA

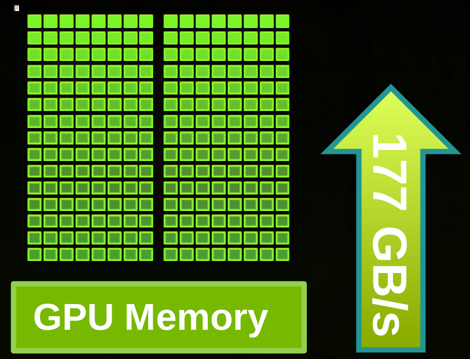
Optimization: CPU and GPU

CPU



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

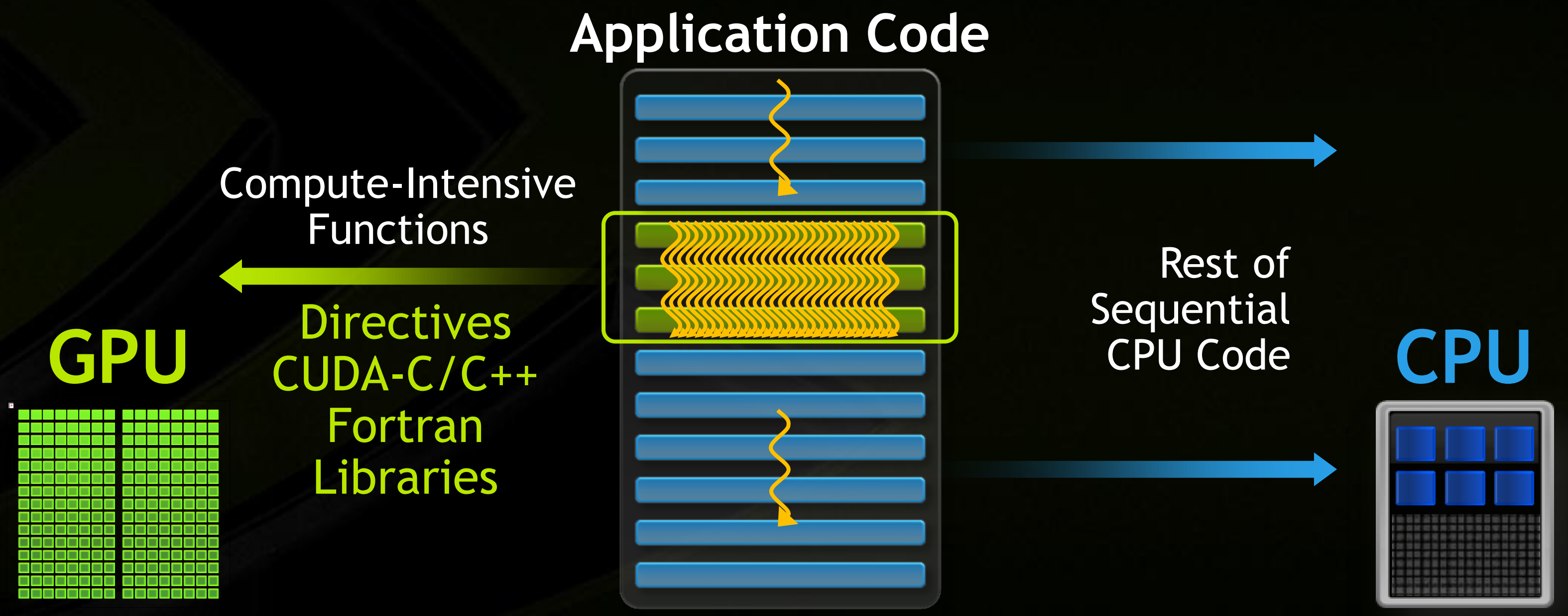
GPU



- Hundreds of cores
- Great memory bandwidth
- Best at parallel execution

Optimization: Maximize Performance

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

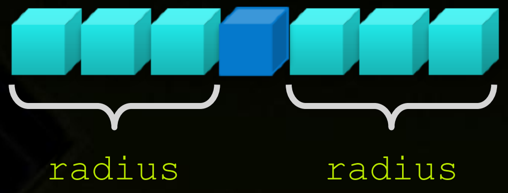


Application Optimization Process and Tools

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

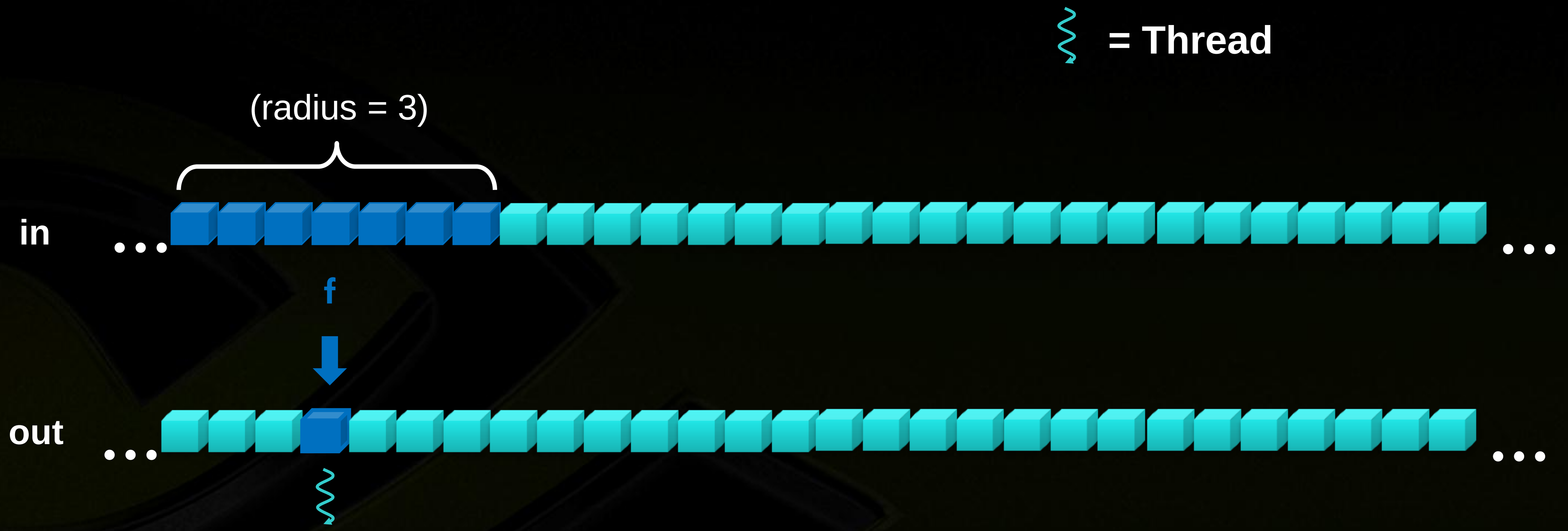
1D Stencil: A Common Algorithmic Pattern

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

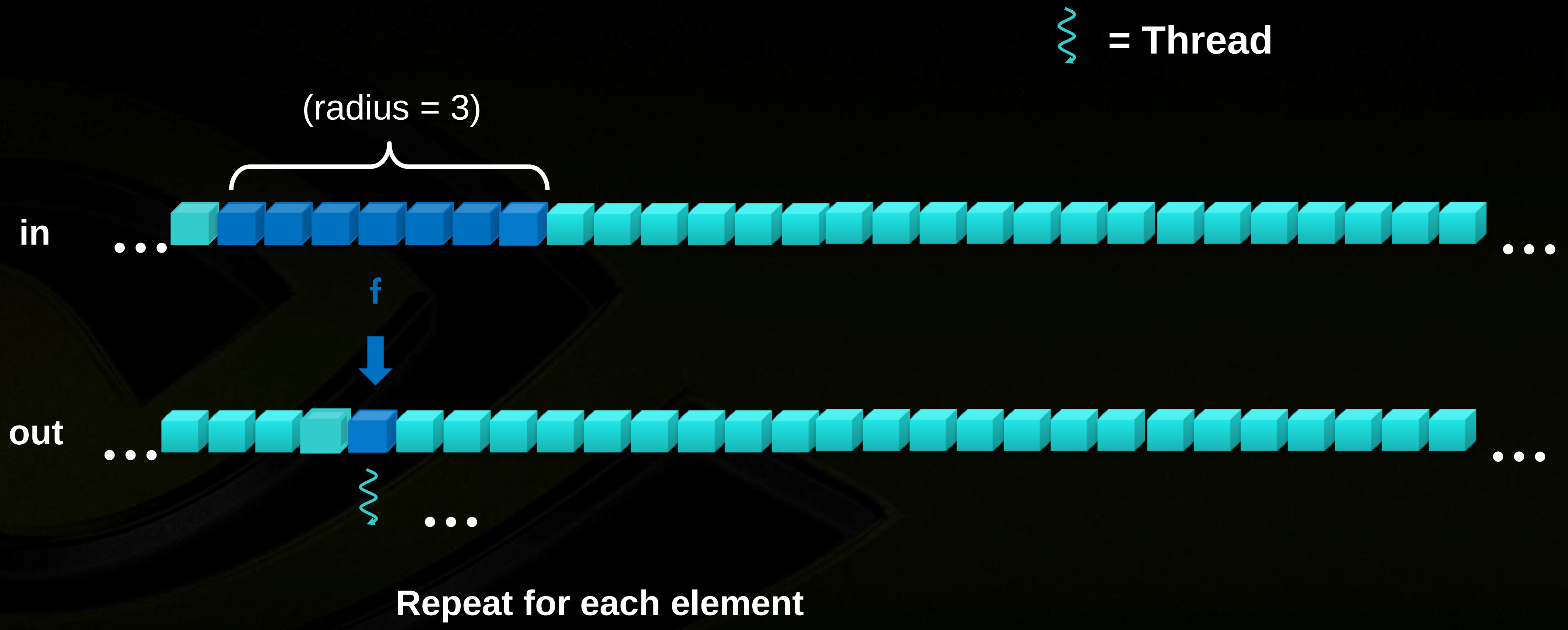


- Fundamental to many algorithms
  - Standard discretization methods, interpolation, convolution, filtering
- Our example will use weighted arithmetic mean

Serial Algorithm



Serial Algorithm



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);
}
```

```
void applyStencil1D(int sIdx, int eIdx, const
          float *weights, float *in, float *out) {

    = 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

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(
  initializeWeights(weights,
  initializeArray(in, N);

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

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

```
void applyStencil1D(int sIdx, int
         float *weights, float *i

  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);
  }
}
```

Weighted mean over radius

For each element…

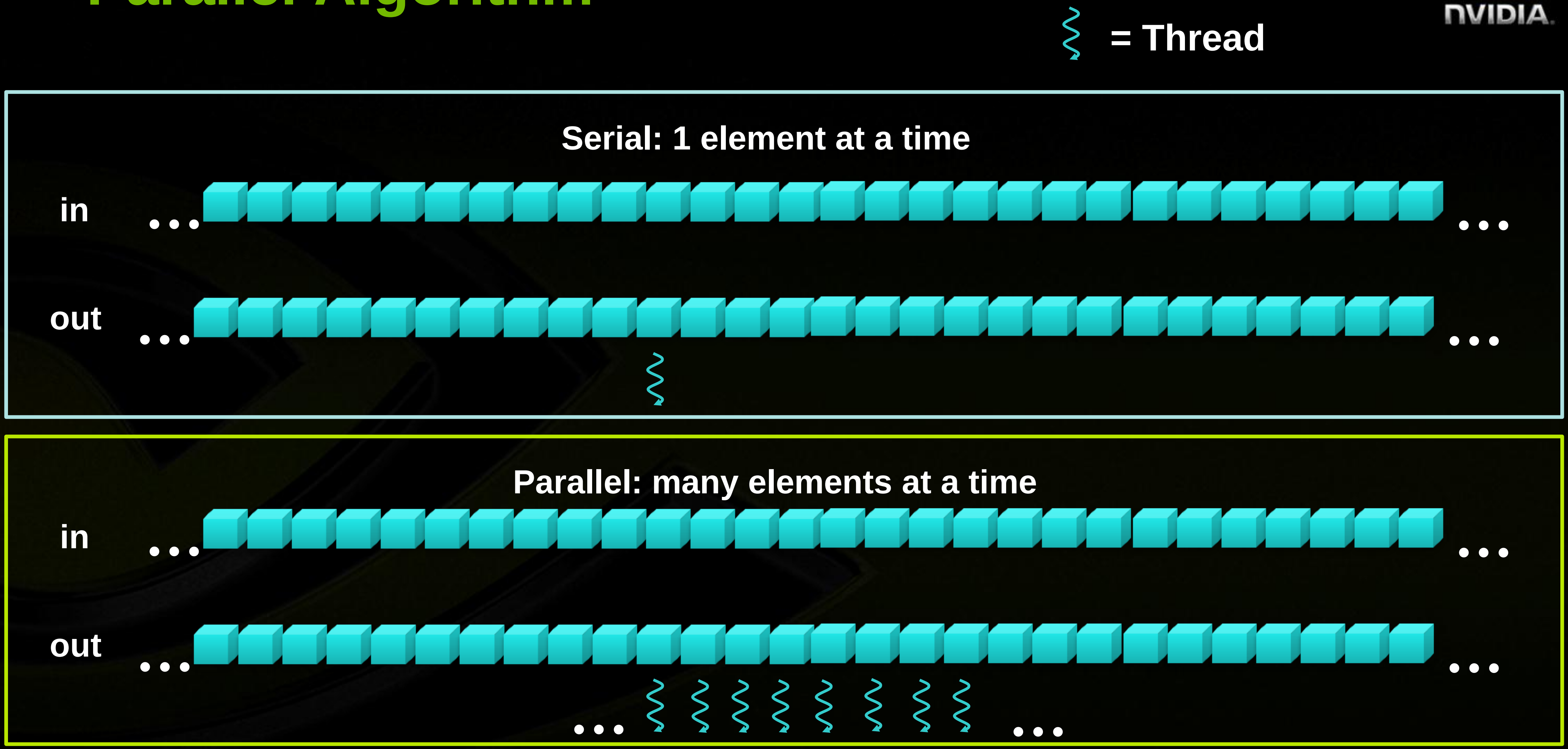
Serial Implementation Performance

```
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

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

```
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



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) {

      Idx + blockIdx.x*blockDim.x + threadIdx.x;
        x) {
        = 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;
                all elements in the stencil
                DIUS; j <= RADIUS; j++) {
                ts[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)
  //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_w
  float *d_in;        cudaMalloc(&d_in
  float *d_out;       cudaMalloc(&d_out, w

  cudaMemcpy(d_weights,weights,wsize,c  aMemcpyHostToDevice);
  cudaMemcpy(d_in, in, wsize, cudaMe  cpyHostToDevice);
  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);
}
```

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
  float *in = (float *)ma
  float *out= (float *)mal
  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

```
(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
```

Switch to the CPU thread

Switch to the frame where the allocation occurred

Found bad allocation size

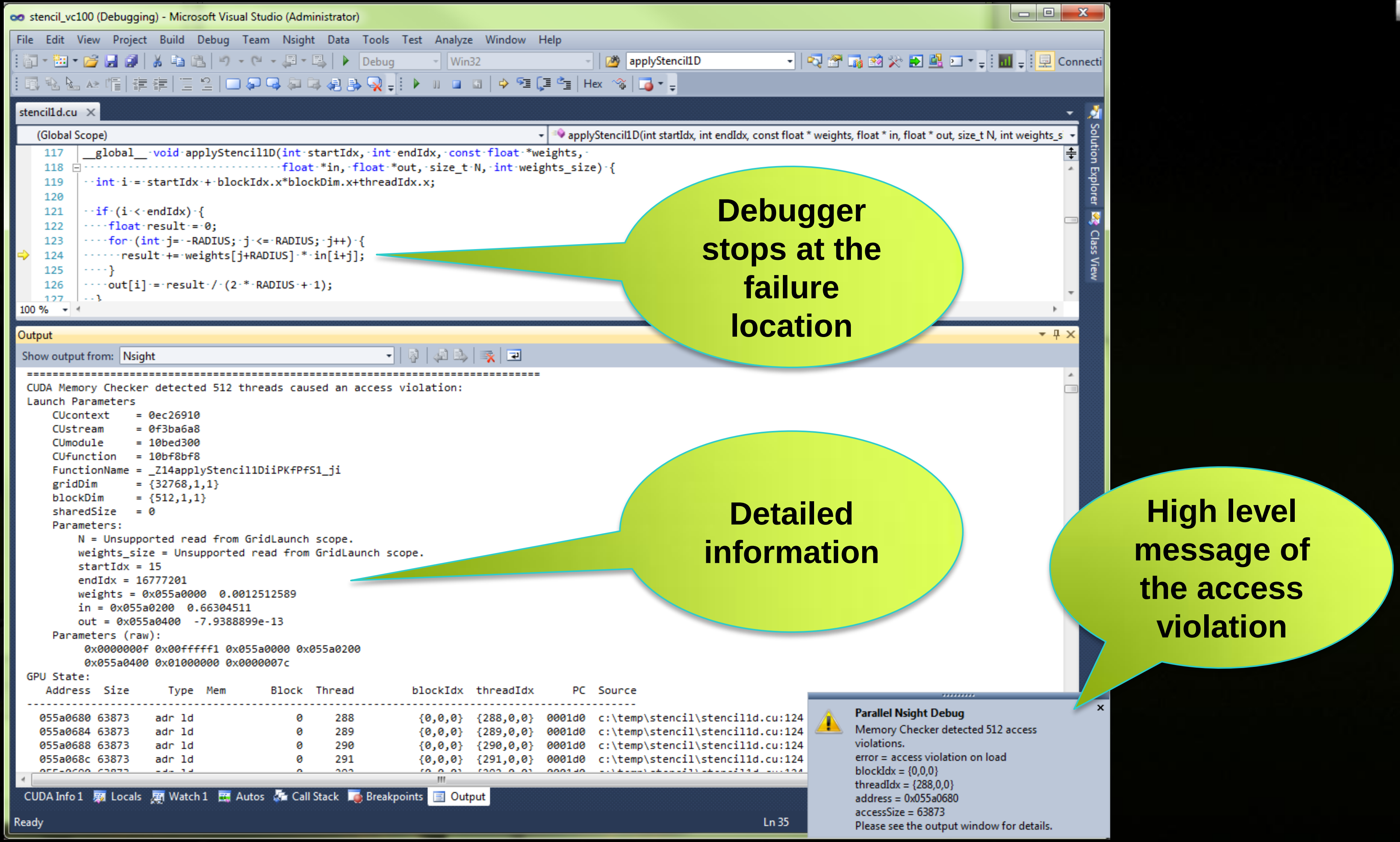
```
float *d_weights; cudaMalloc(&d_weights , wsize);
float *d_in; cudaMalloc(&d_in , wsize);
float *d_out; cudaMalloc(&d_out , wsize);

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, ...);
```

Corrected Parallel Implementation

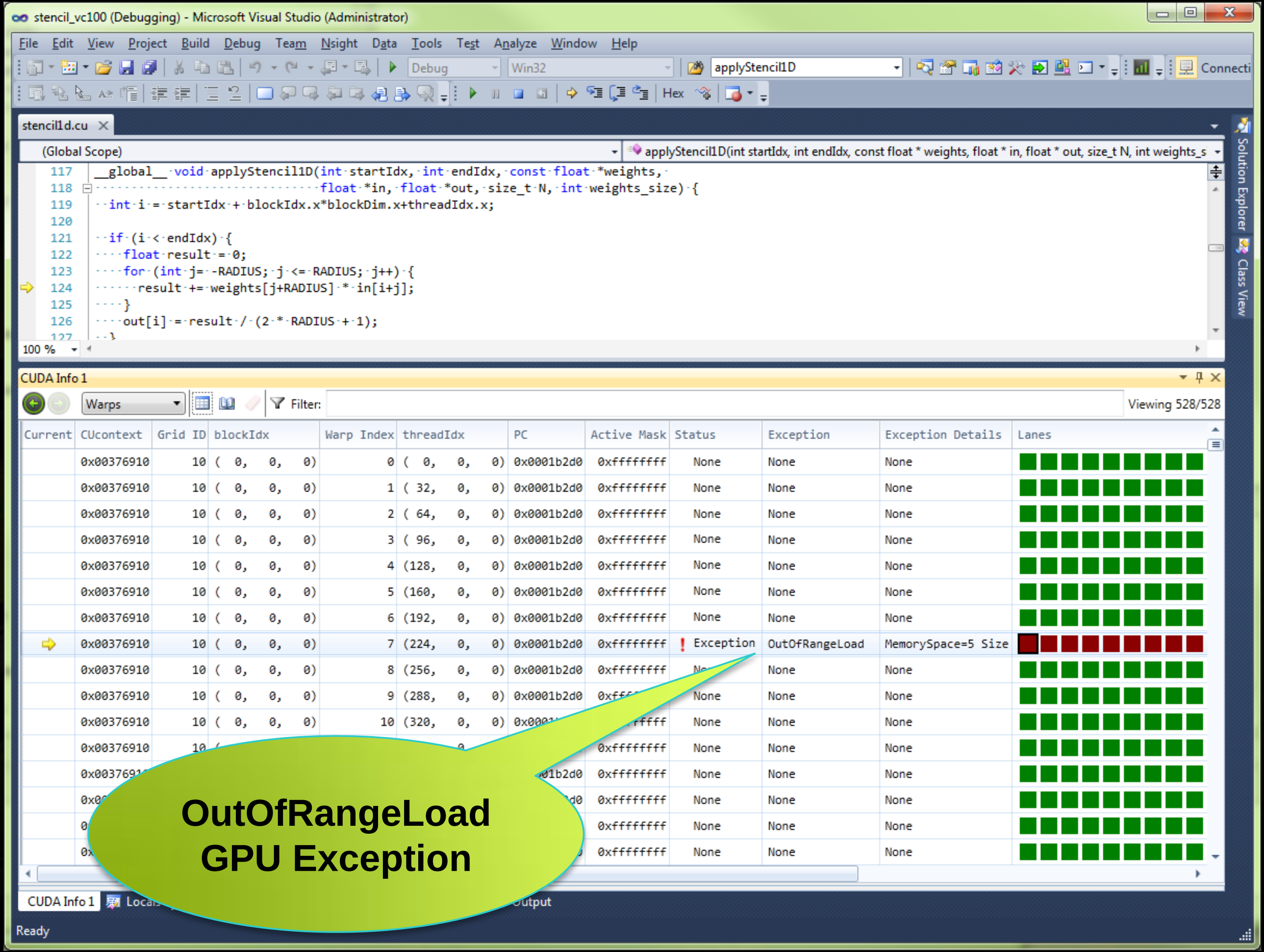
```
int main() {
  int size = N * sizeof(float);
  int wsize = (2 * RADIUS + 1) * sizeof(float);
  //allocate resources
  float *weights = (float *)malloc(wsize);
  float *in = (float *)malloc(size);
  float *out= (float *)malloc(size);
  initializeWeights(weights, RADIUS);
  initializeArray(in, N);
  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



Parallel Nsight for Visual Studio



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 resourc
  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

Printf

- **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, …)

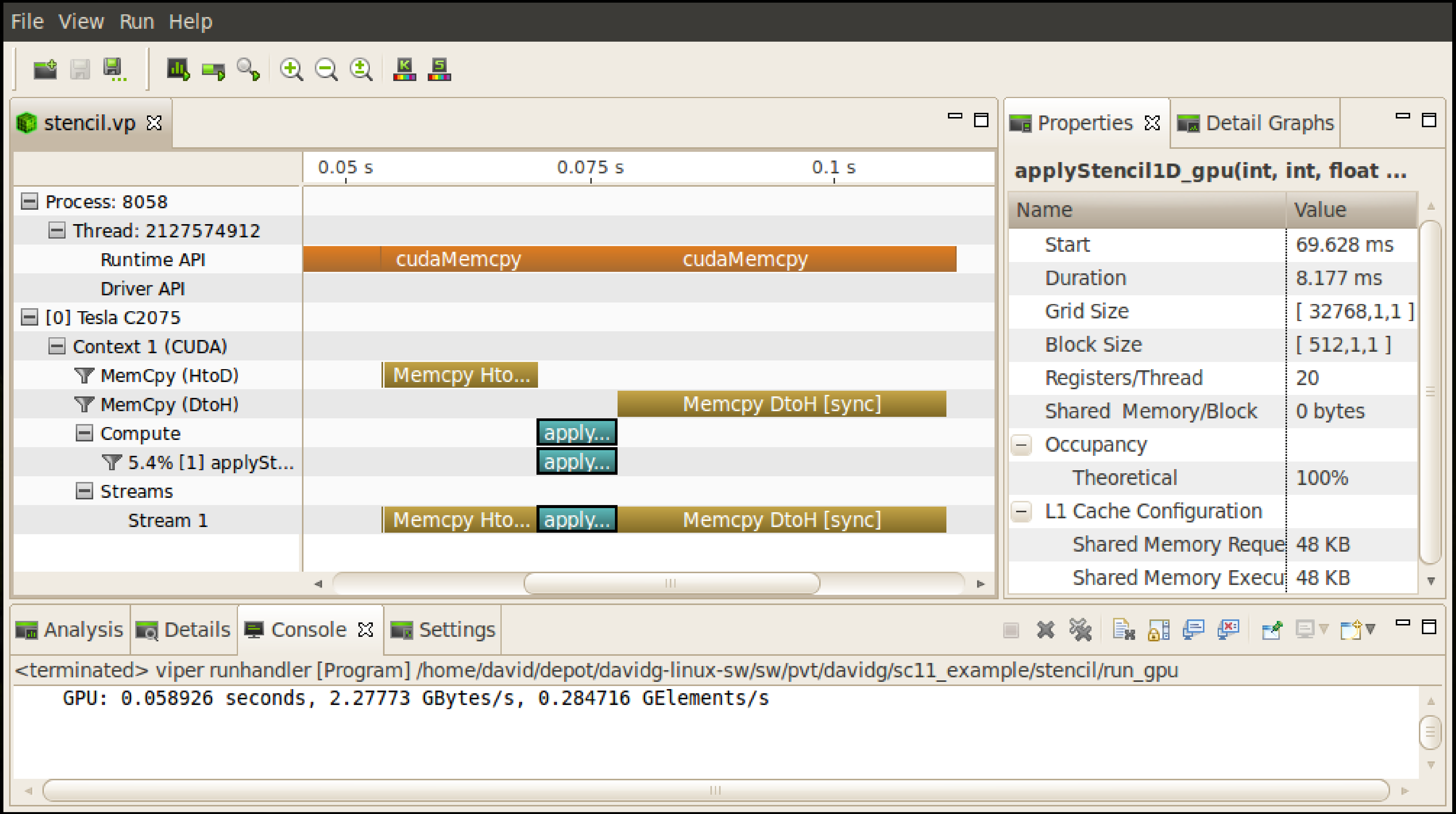
Application Optimization Process (Revisited)

- Identify Optimization Opportunities
  - 1D stencil algorithm
- Parallelize with CUDA, confirm functional correctness
  - cuda-gdb, cuda-memcheck
- Optimize
  - ?

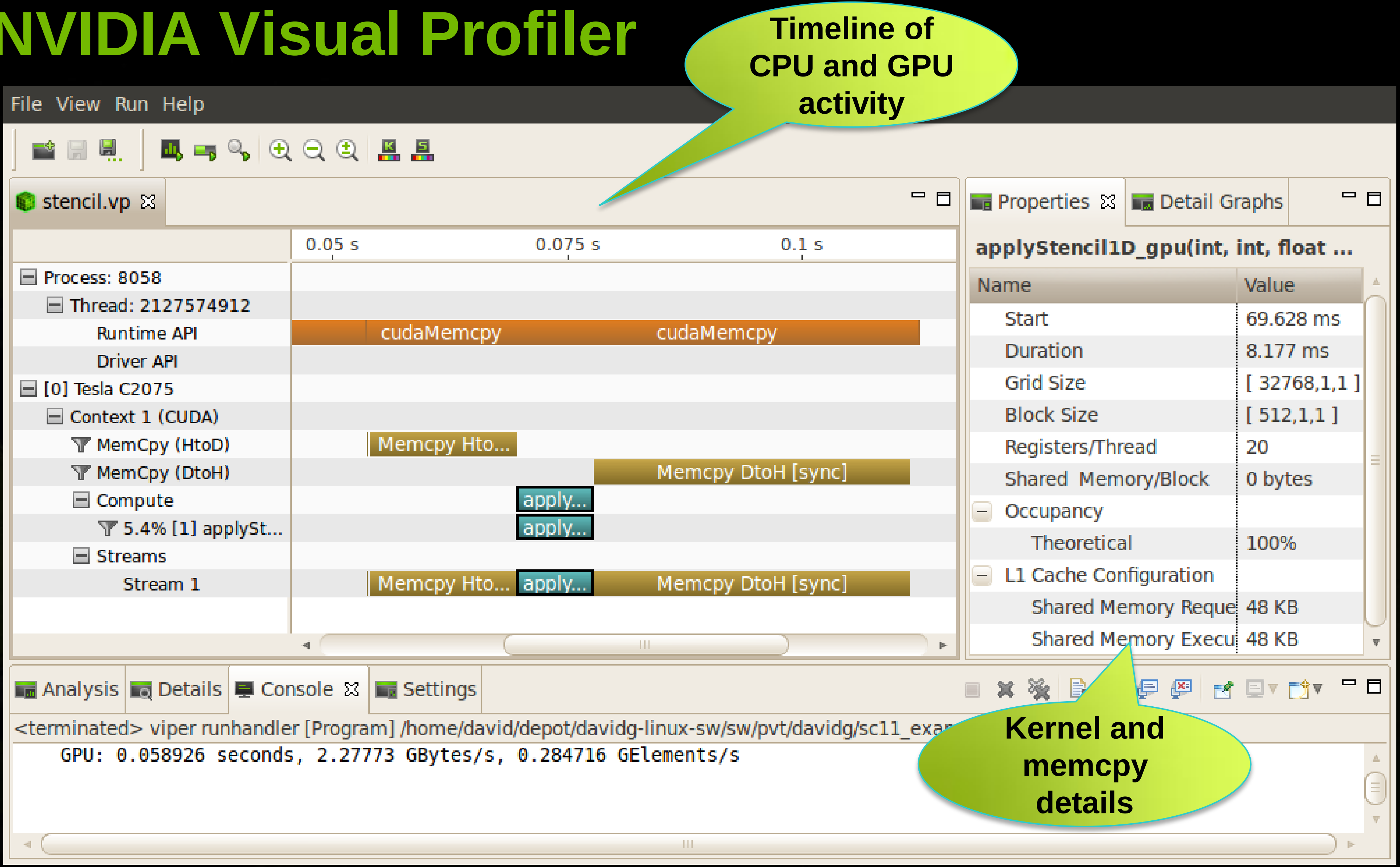
Optimize

- Can we get more performance?
- Visual Profiler
  - Visualize CPU and GPU activity
  - Identify optimization opportunities
  - Automated analysis

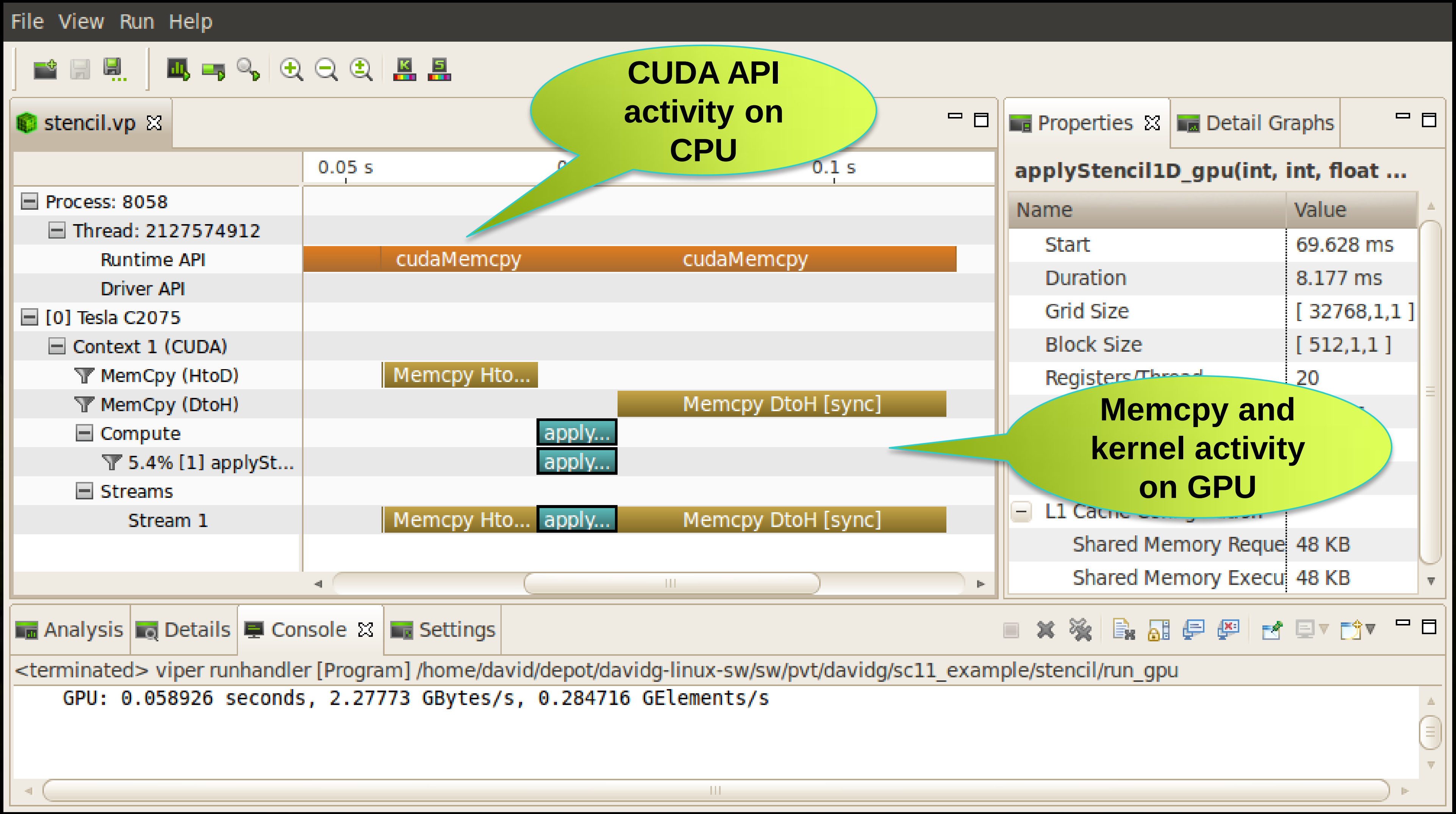
NVIDIA Visual Profiler



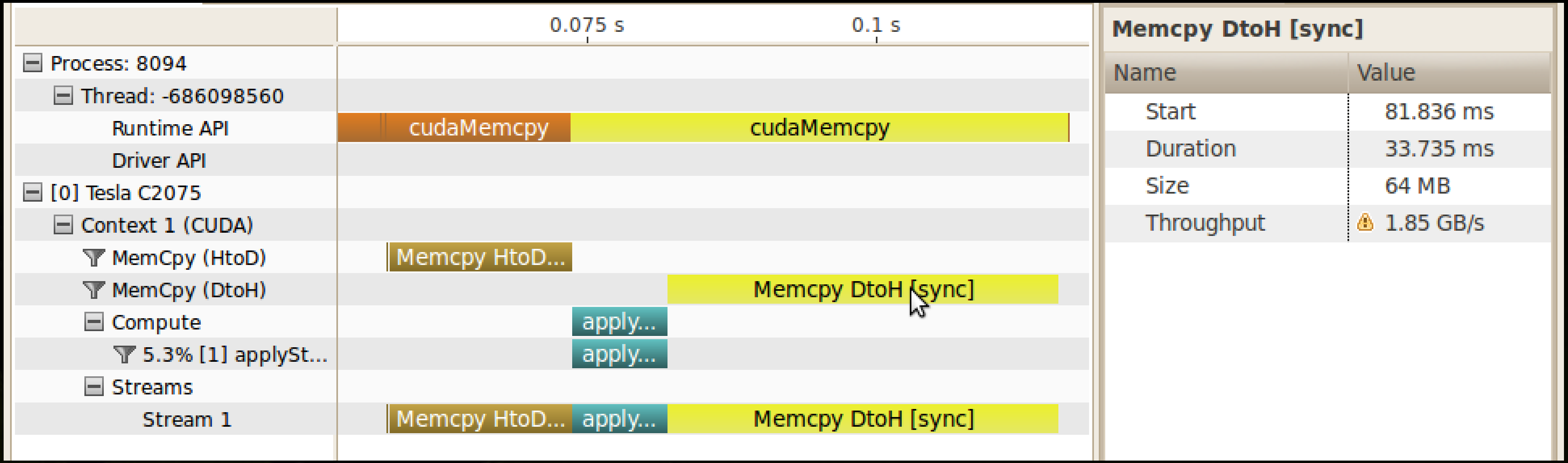
NVIDIA Visual Profiler



NVIDIA Visual Profiler



Detecting Low Memory Throughput

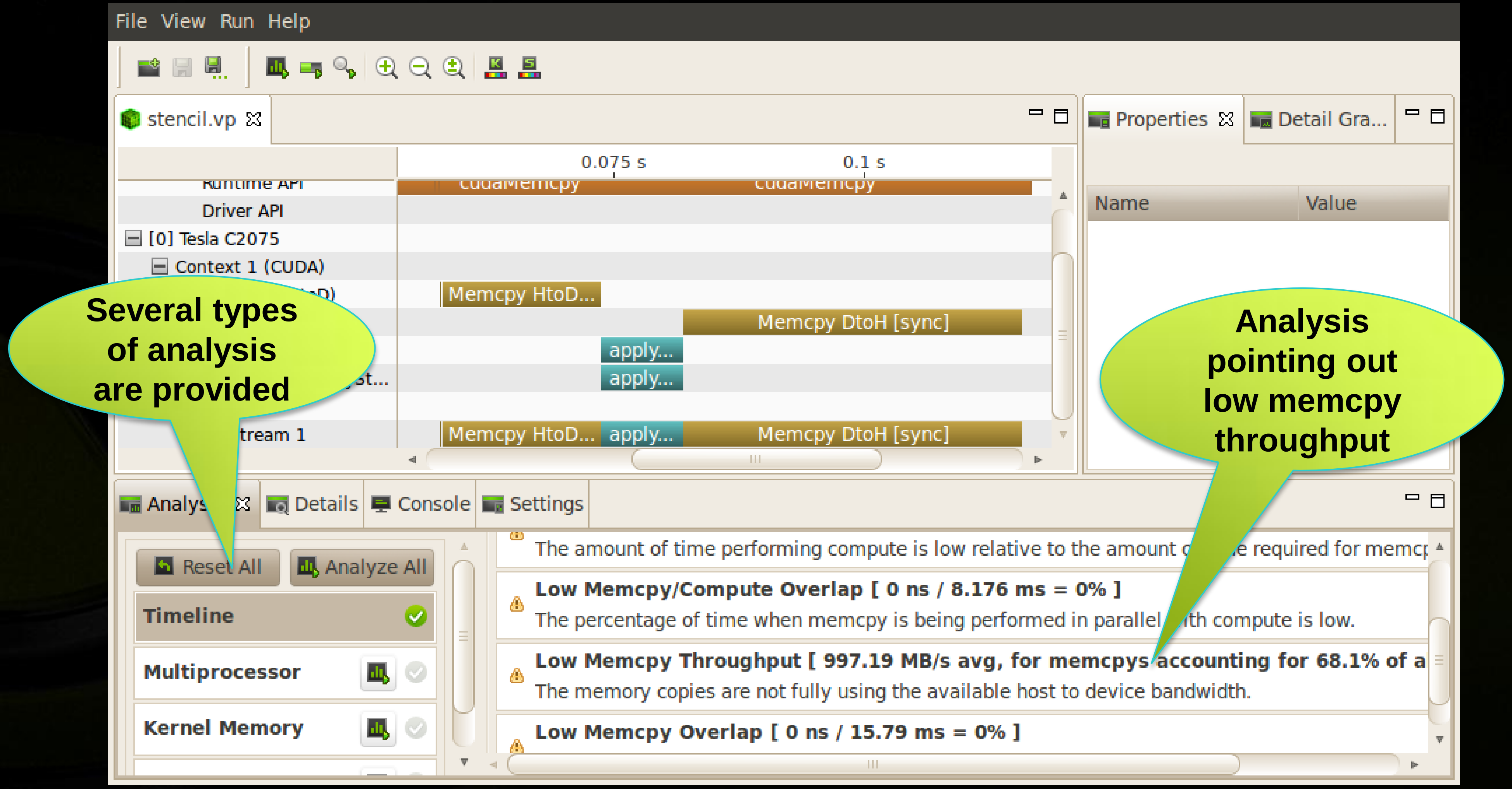


- **Spend majority of time in data transfer**
  - **Often can be overlapped with preceding or following computation**
- **From timeline can see that throughput is low**
  - **PCIe x16 can sustain > 5GB/s**

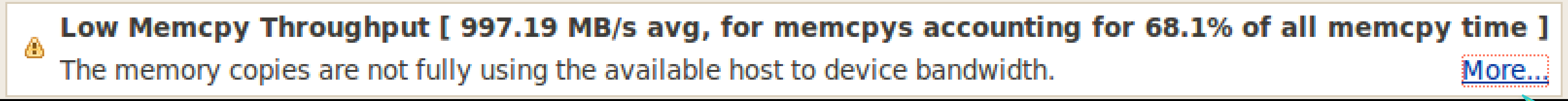
Visual Profiler Analysis

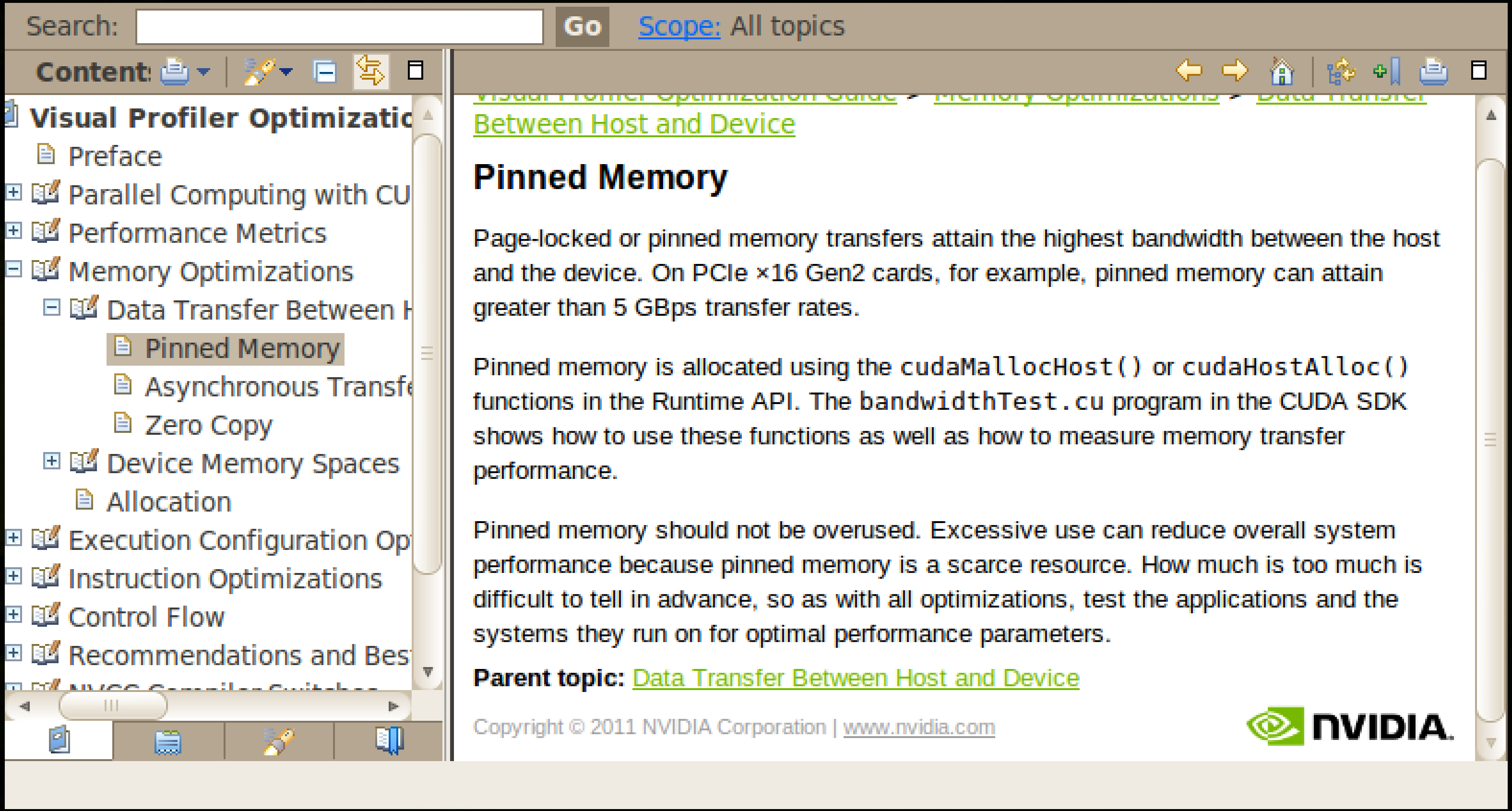
- How do we know when there is an optimization opportunity?
  - Timeline visualization seems to indicate an opportunity
  - Documentation gives guidance and strategies for tuning
    - CUDA Best Practices Guide
    - CUDA Programming Guide
- Visual Profiler analyzes your application
  - Uses timeline and other collected information
  - Highlights specific guidance from Best Practices
  - Like having a customized Best Practices Guide for your application

Visual Profiler Analysis



Online Optimization Help





Each analysis has link to Best Practices documentation

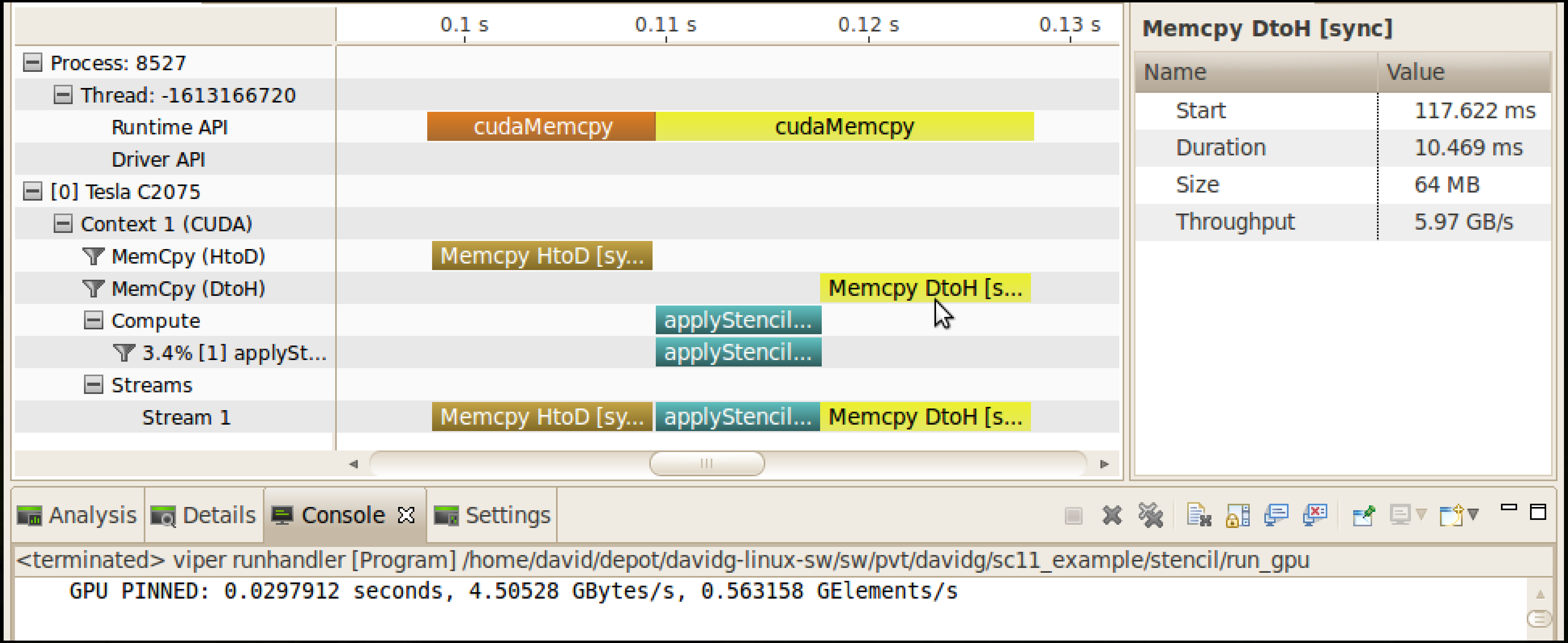
Pinned CPU Memory Implementation

```
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);
  …
```

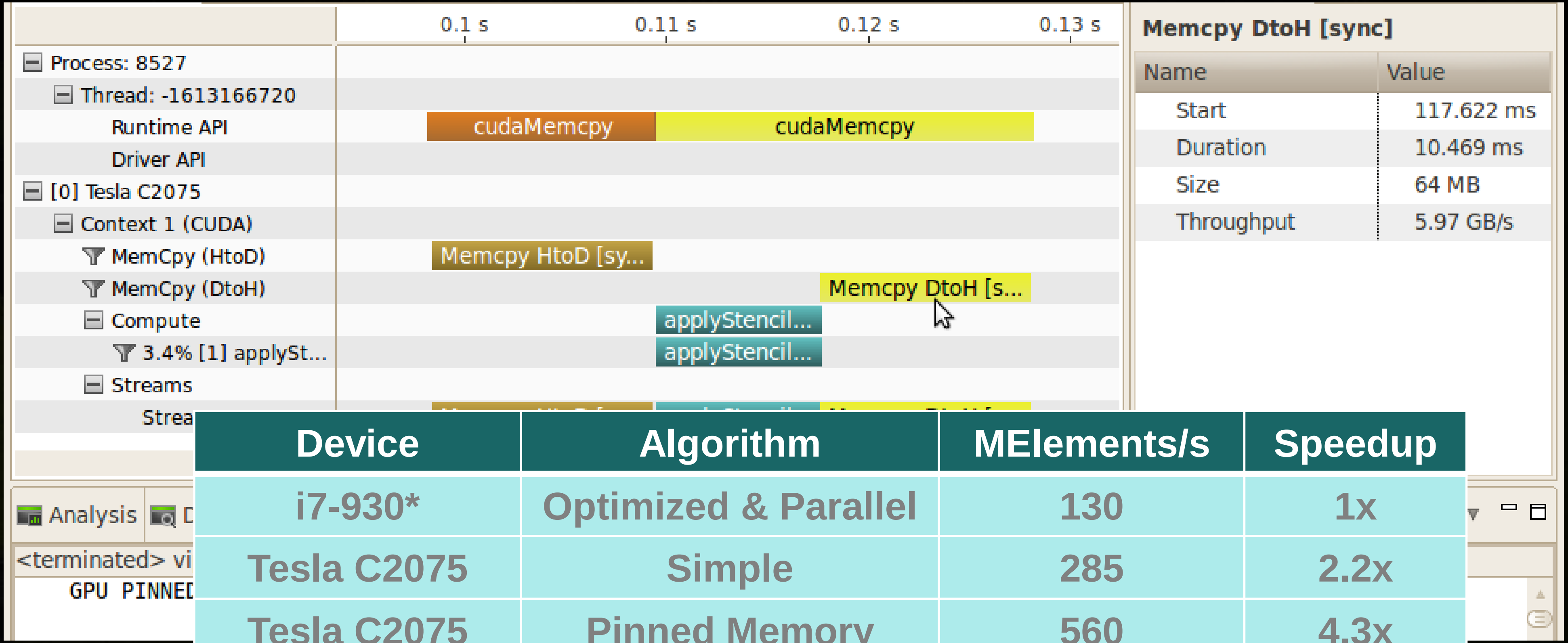
CPU allocations use pinned memory to enable fast memcpy

No other changes

Pinned CPU Memory Result



Pinned CPU Memory Result



Device	Algorithm	MElements/s	Speedup
i7-930*	Optimized & Parallel	130	1x
Tesla C2075	Simple	285	2.2x
Tesla C2075	Pinned Memory	560	4.3x

*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**

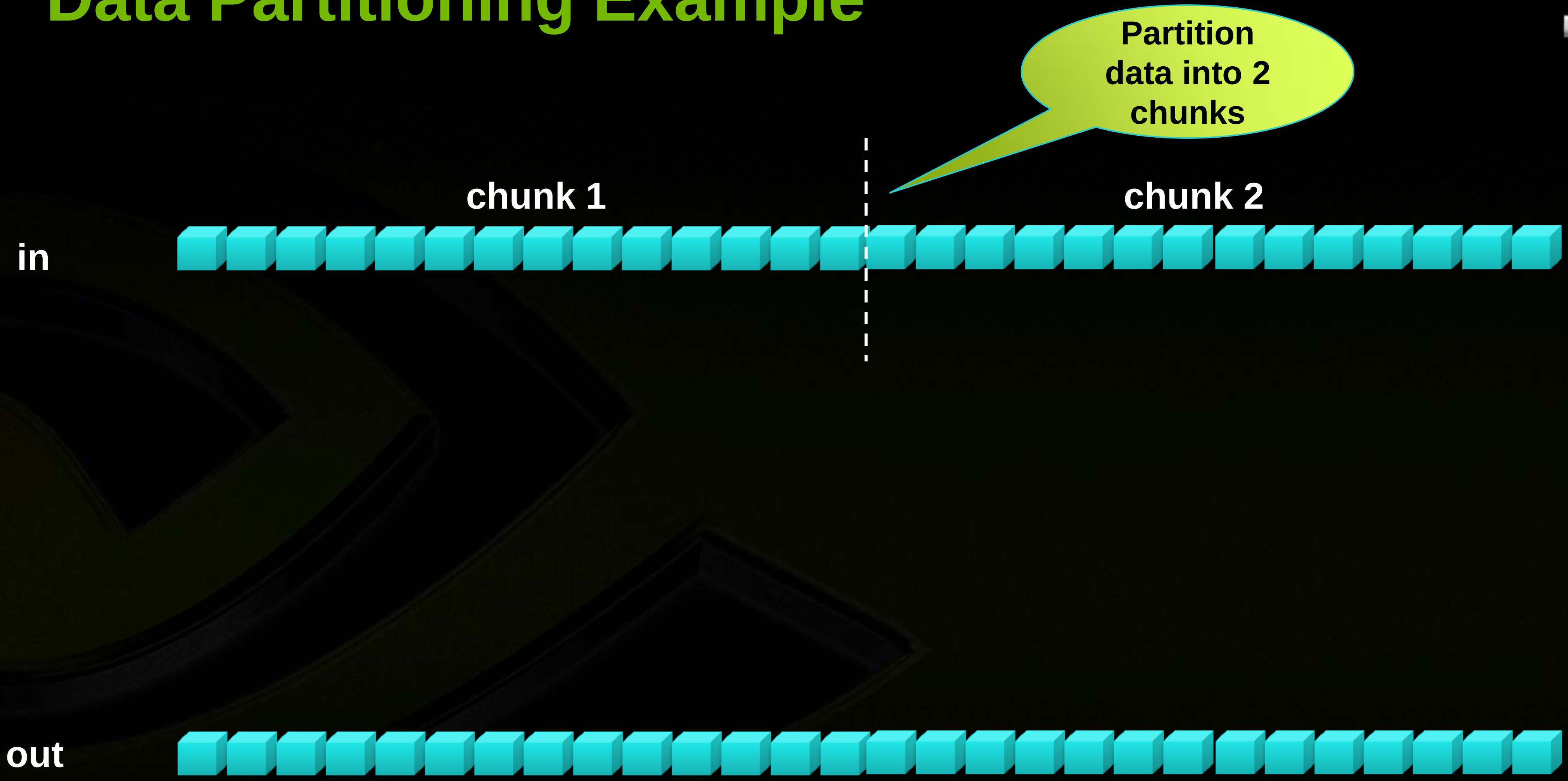
Visual Profiler Optimization Guide > Memory Optimizations > Data Transfer Between Host and Device

Asynchronous Transfers and Overlapping Transfers with Computation

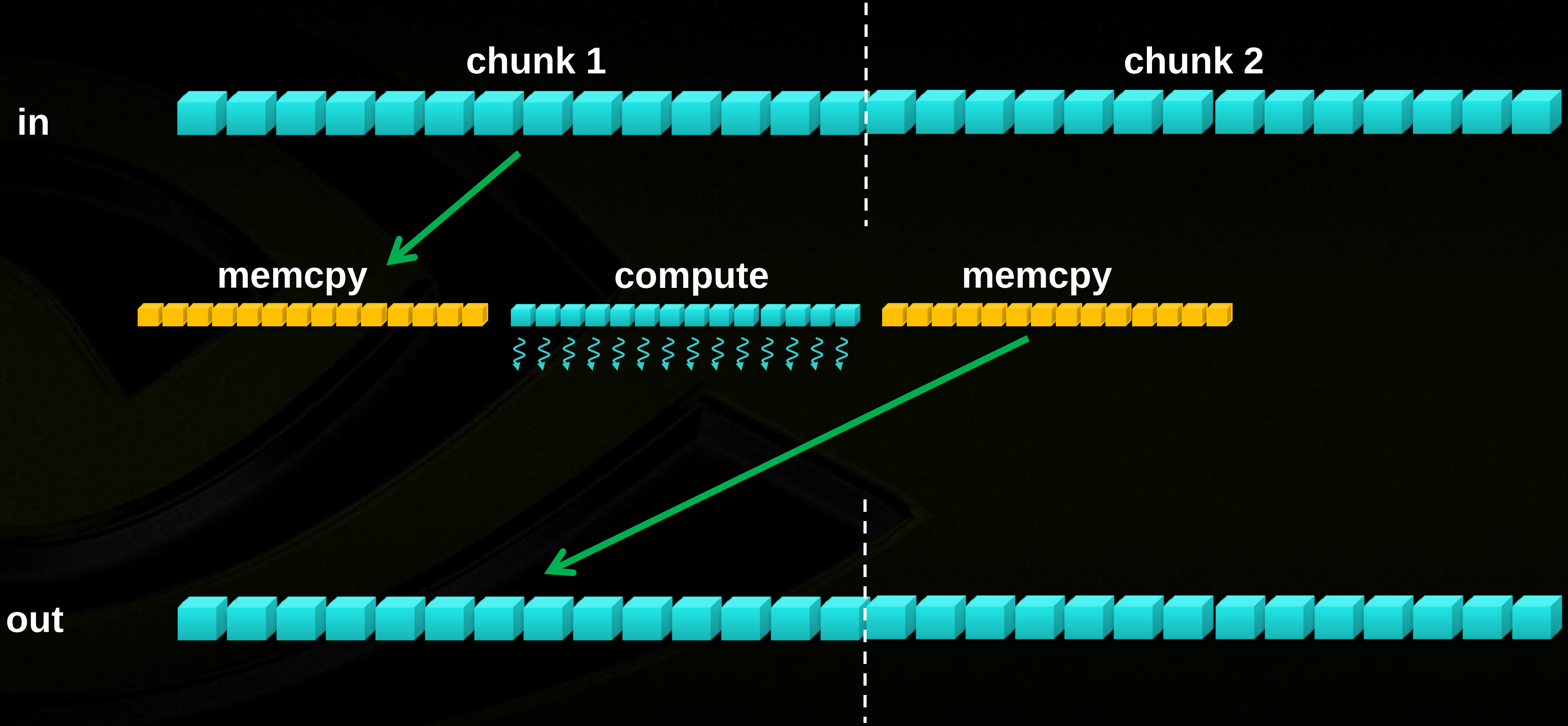
Data transfers between the host and the device using `cudaMemcpy()` are blocking transfers; that is, control is returned to the host thread only after the data transfer is complete. The `cudaMemcpyAsync()` function is a non-blocking variant of `cudaMemcpy()` in which control is returned immediately to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

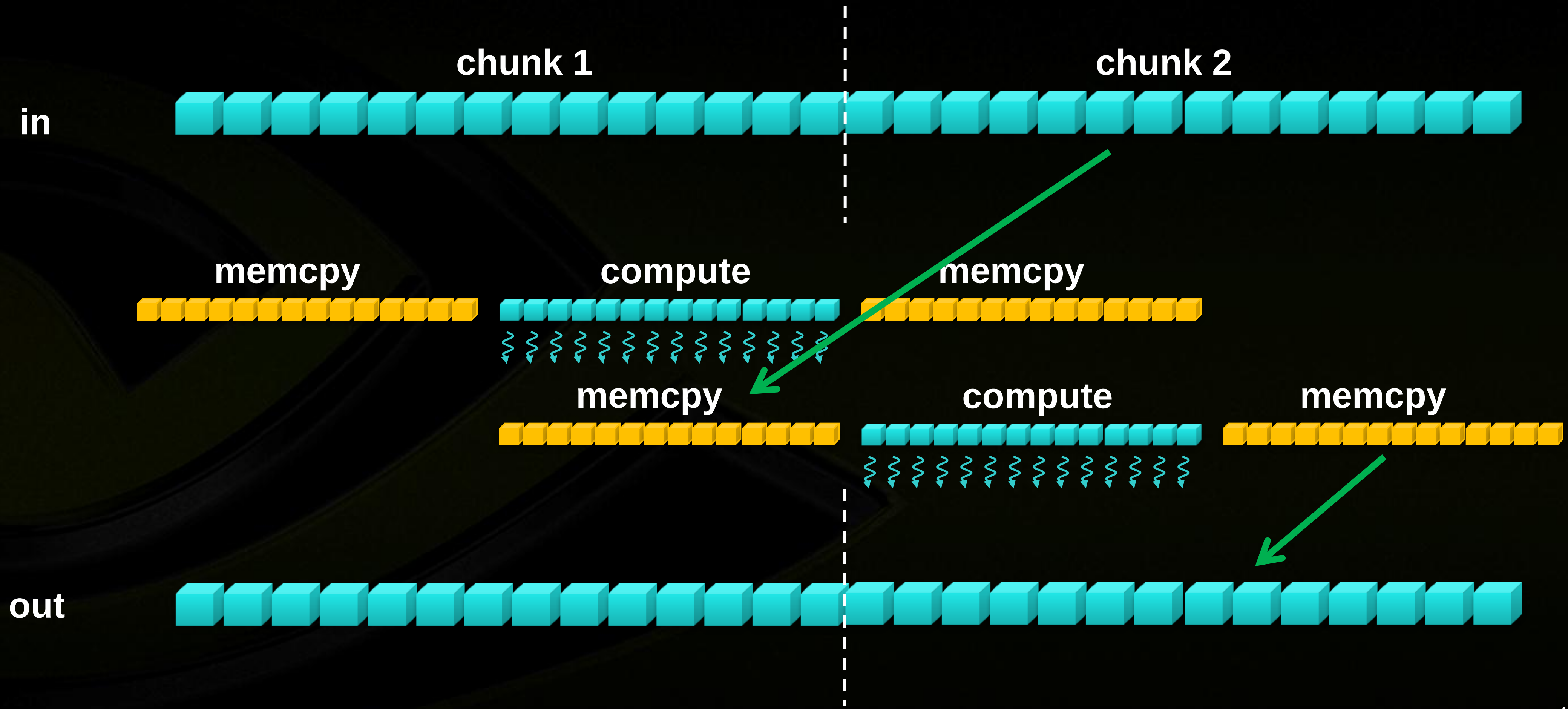
Data Partitioning Example



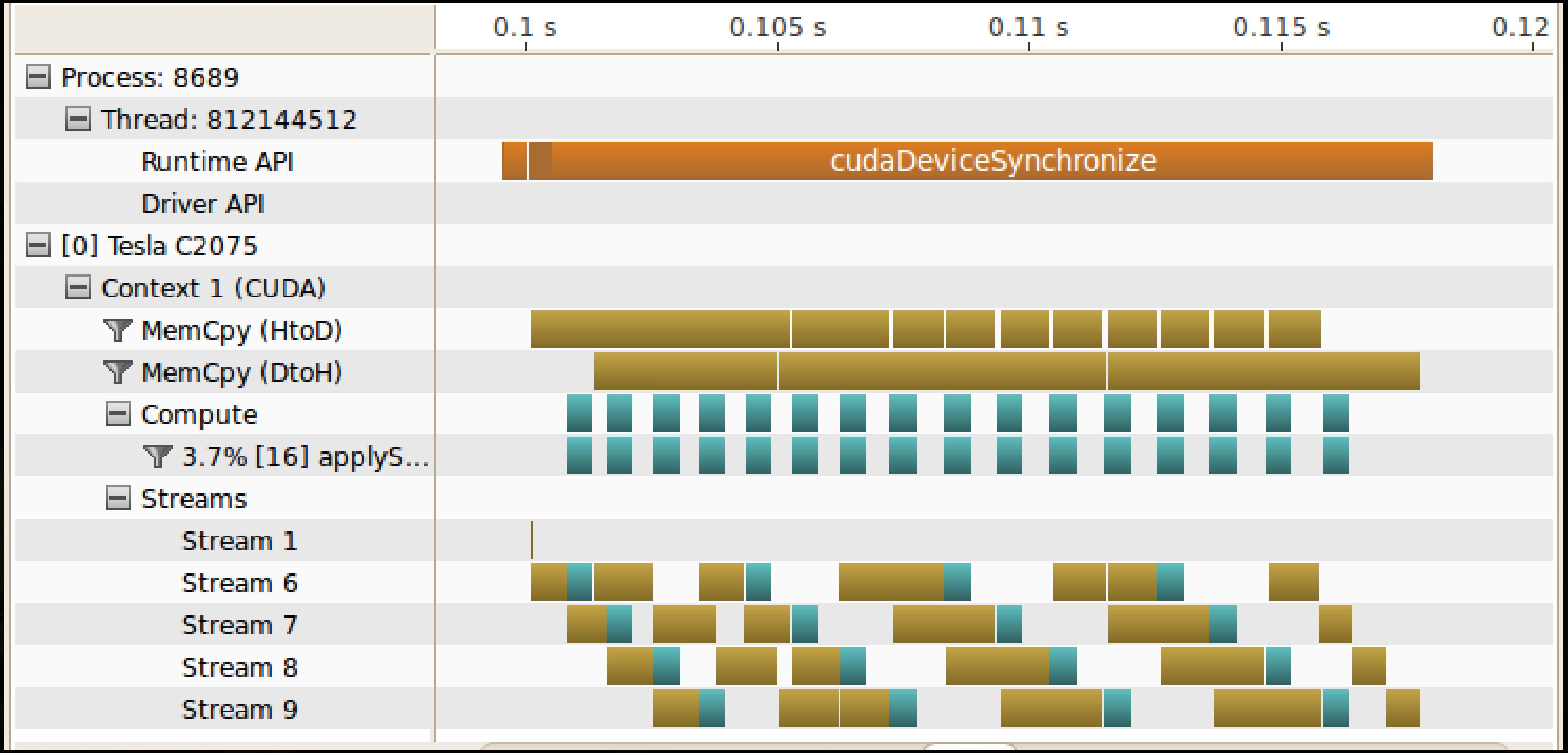
Data Partitioning Example



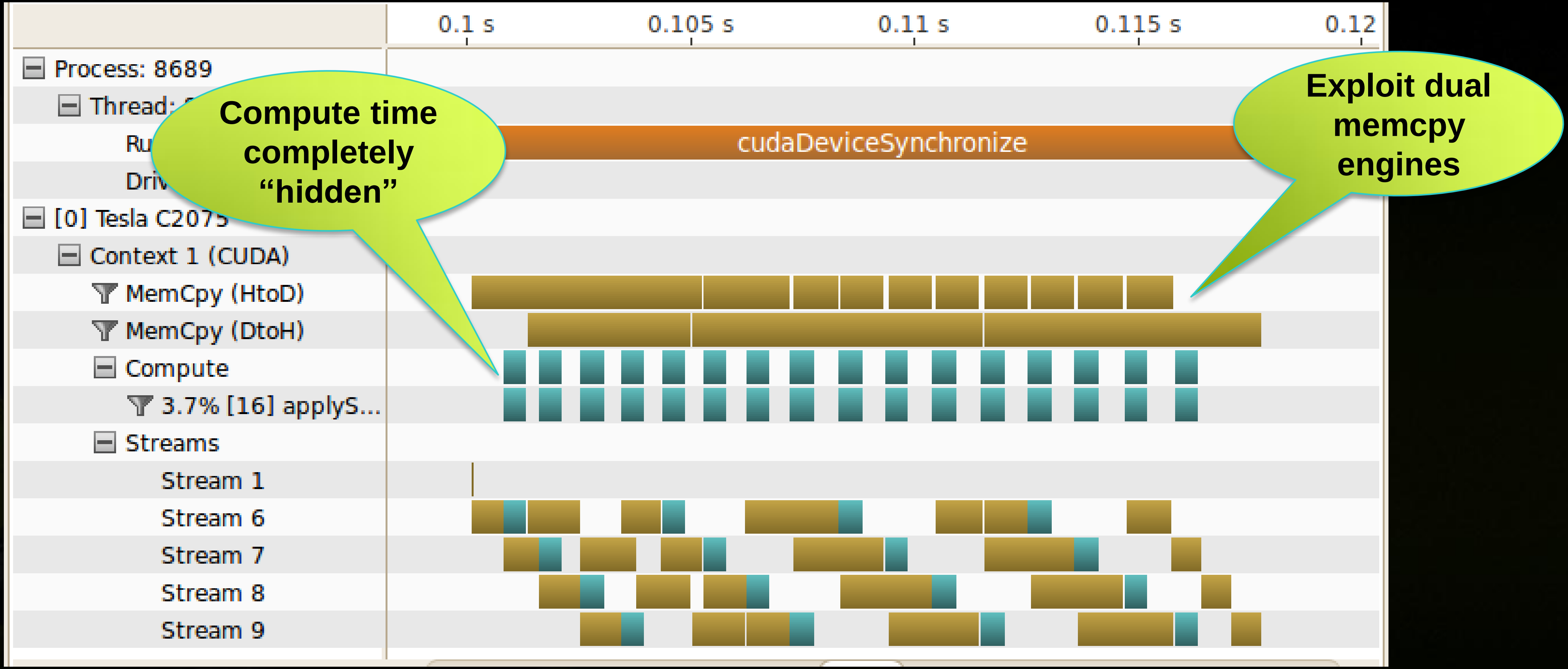
Data Partitioning Example



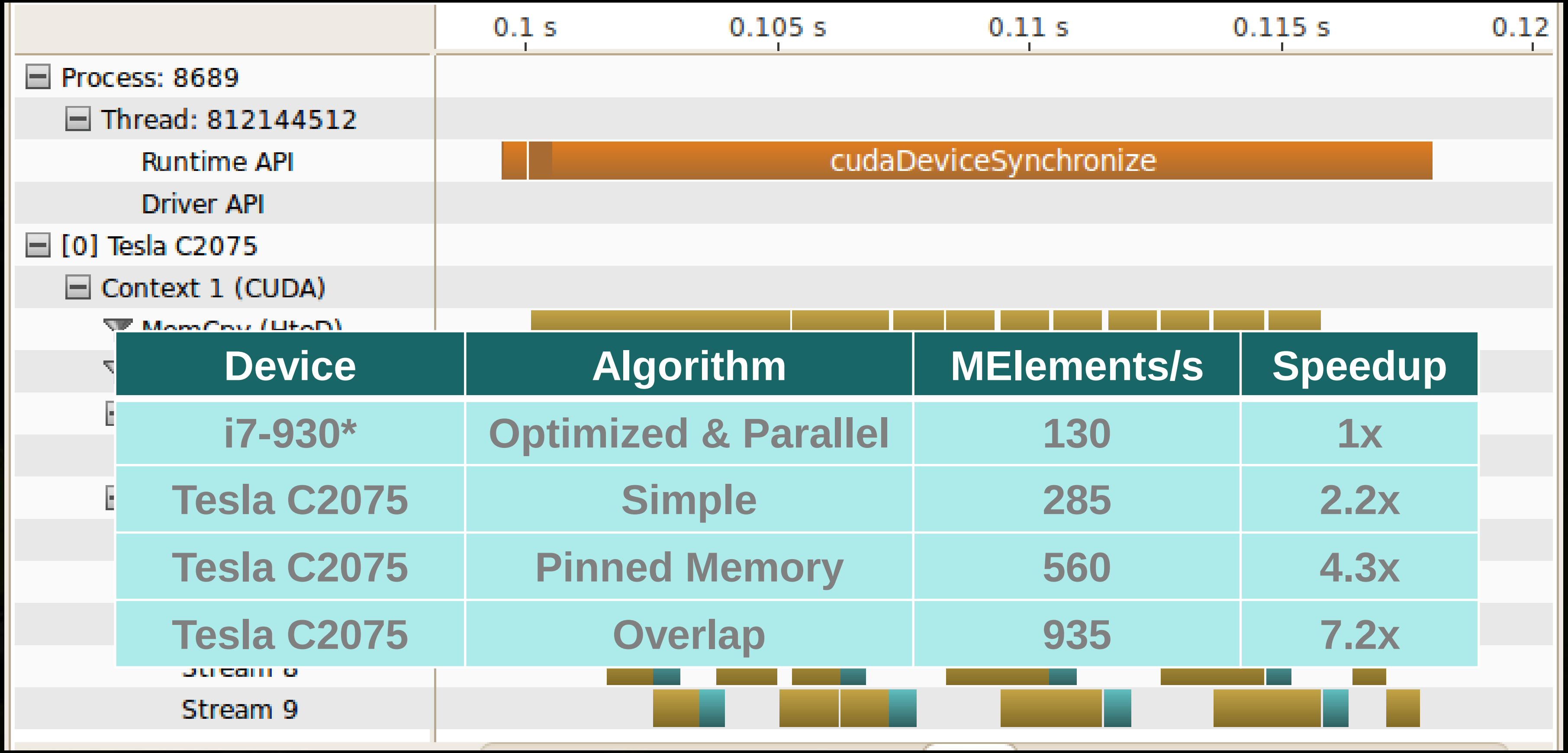
Overlapped Compute/Memcpy



Overlapped Compute/Memcpy



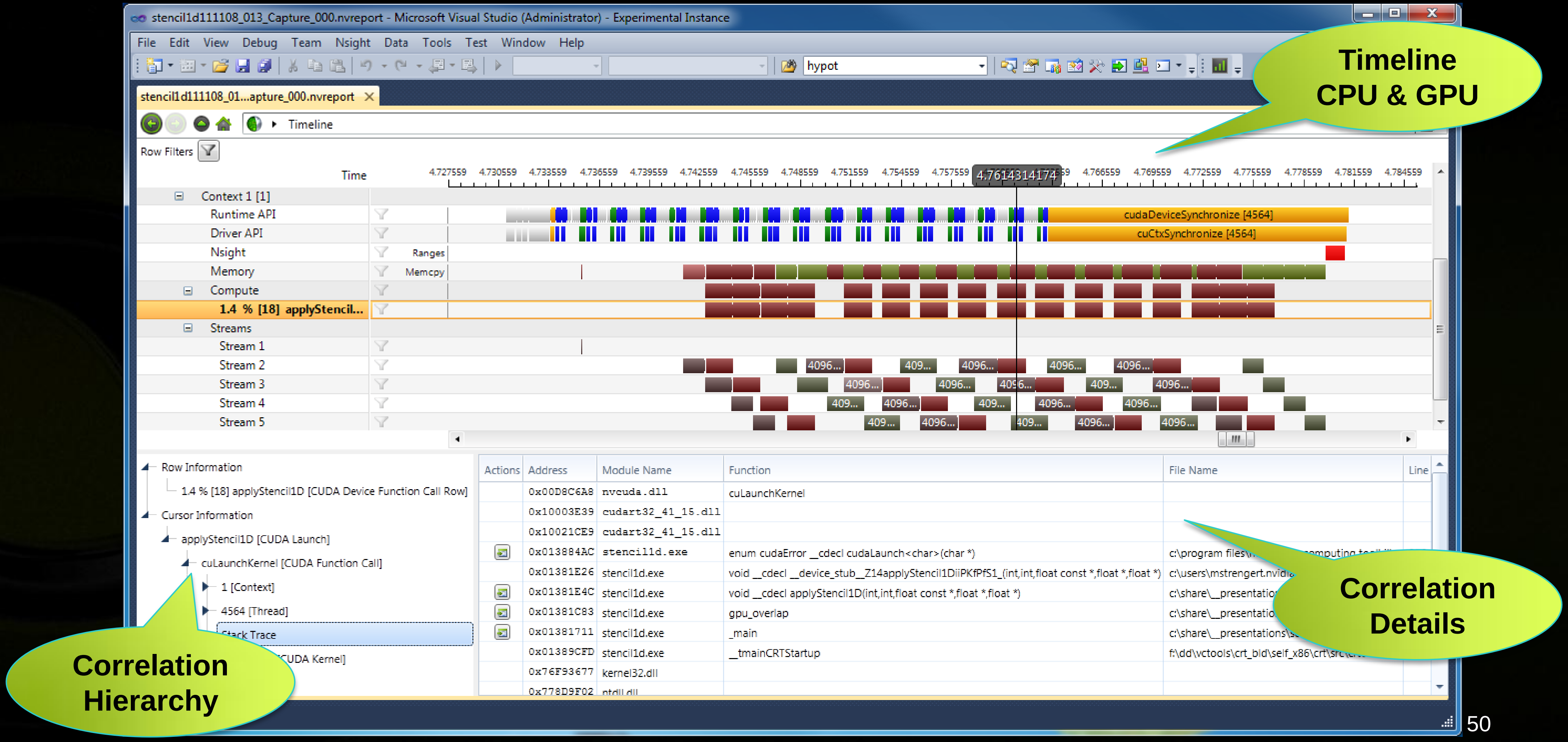
Overlapped Compute/Memcpy Result



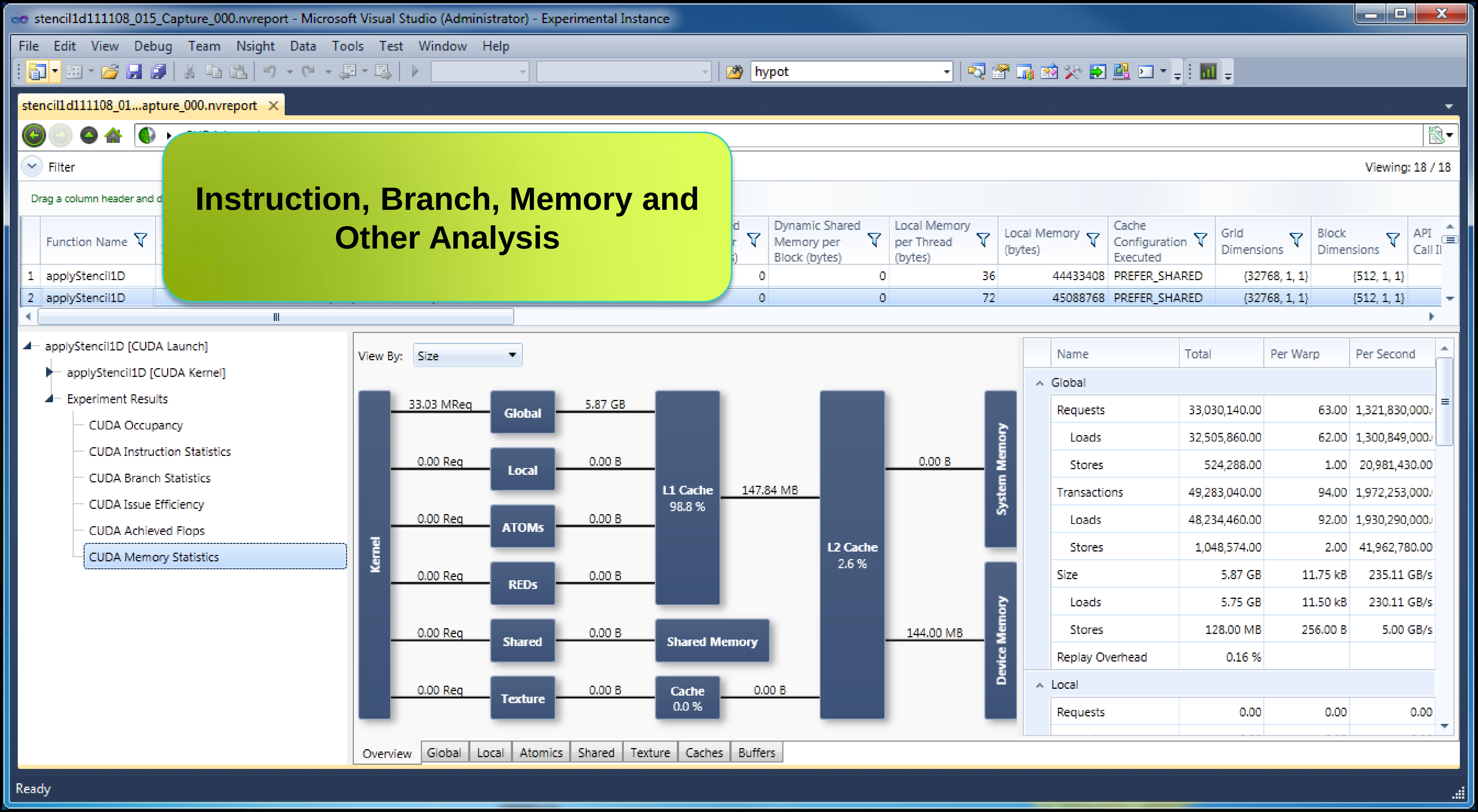
Device	Algorithm	MElements/s	Speedup
i7-930*	Optimized & Parallel	130	1x
Tesla C2075	Simple	285	2.2x
Tesla C2075	Pinned Memory	560	4.3x
Tesla C2075	Overlap	935	7.2x

*4 cores + hyperthreading

Parallel Nsight For Visual Studio



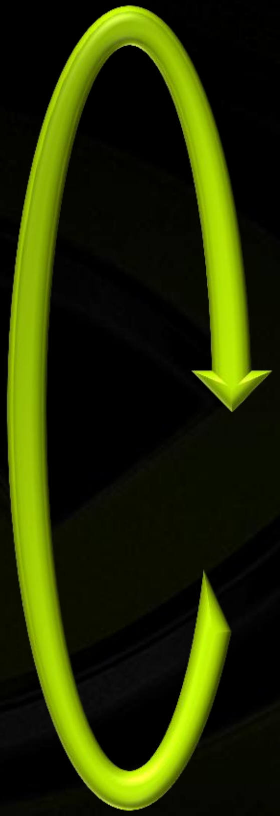
Parallel Nsight For Visual Studio



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?