Getting The Most from CUDA 5 and Kepler

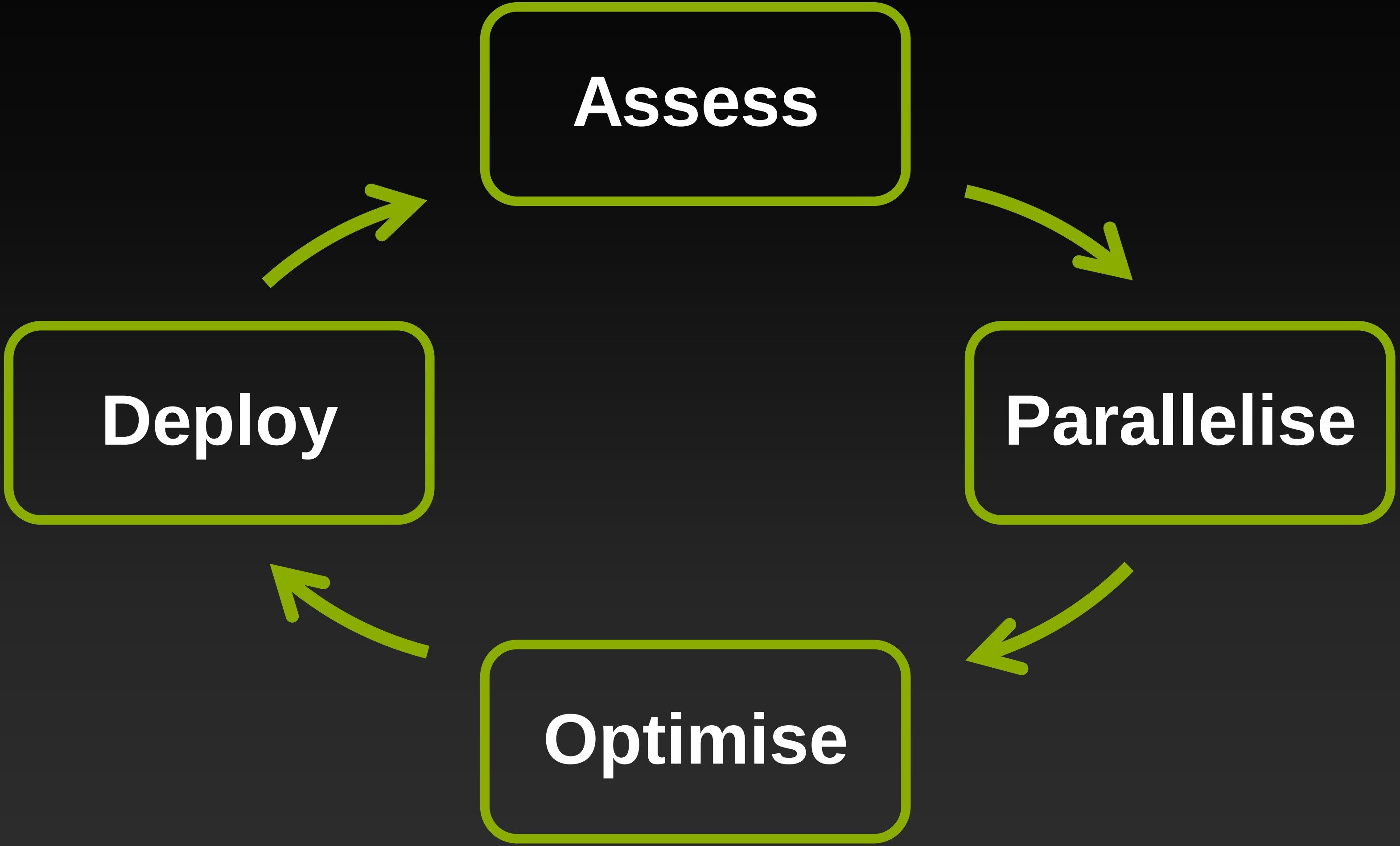
Thomas Bradley, NVIDIA
Developer Technology Group

Getting The Most from CUDA 5 and Kepler

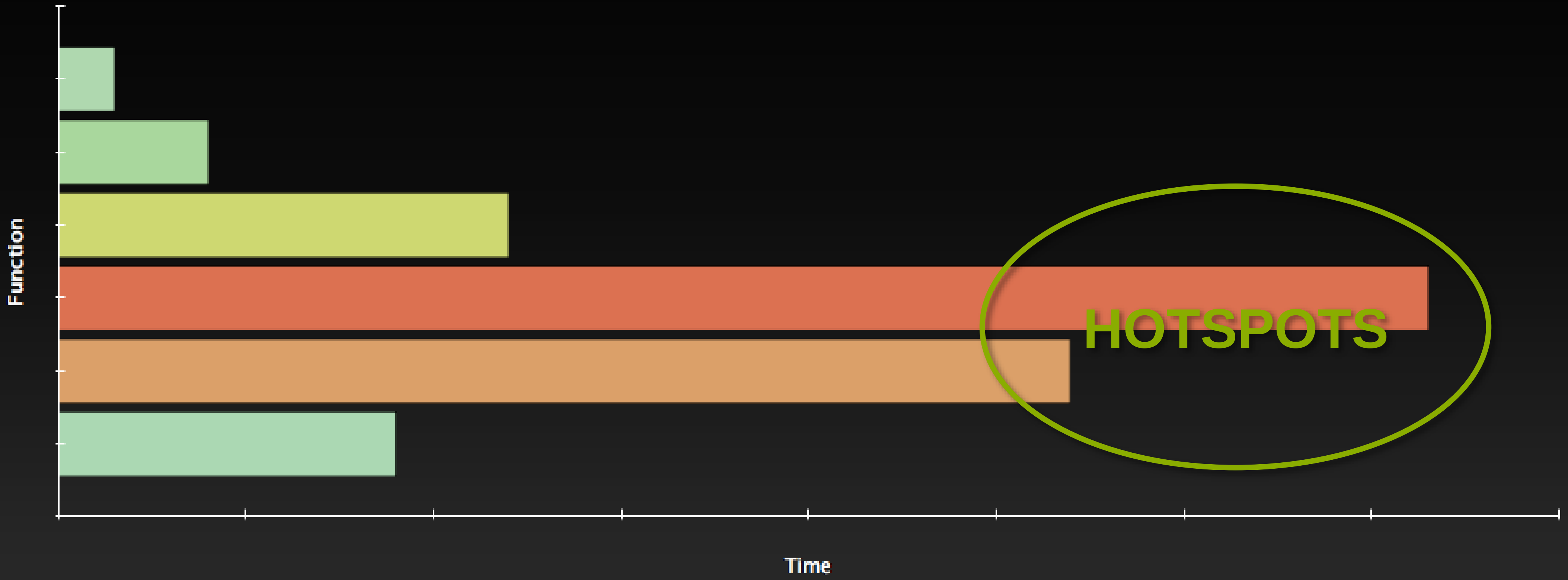
- APOD: a systematic path to performance
  - Analyse, Parallelise, Optimise, Deploy

- Getting the most from Kepler and CUDA 5.0
  - Exposing fine-grained parallelism
    - Kepler SMX architecture
    - ILP vs. TLP
    - Memory-system Parallelism
  - Leveraging coarse-grained parallelism
    - Dynamic Parallelism
    - Hyper-Q and CPU callbacks: simplified pipelining
  - Separate compilation and linking

APOD: A Systematic Path to Performance

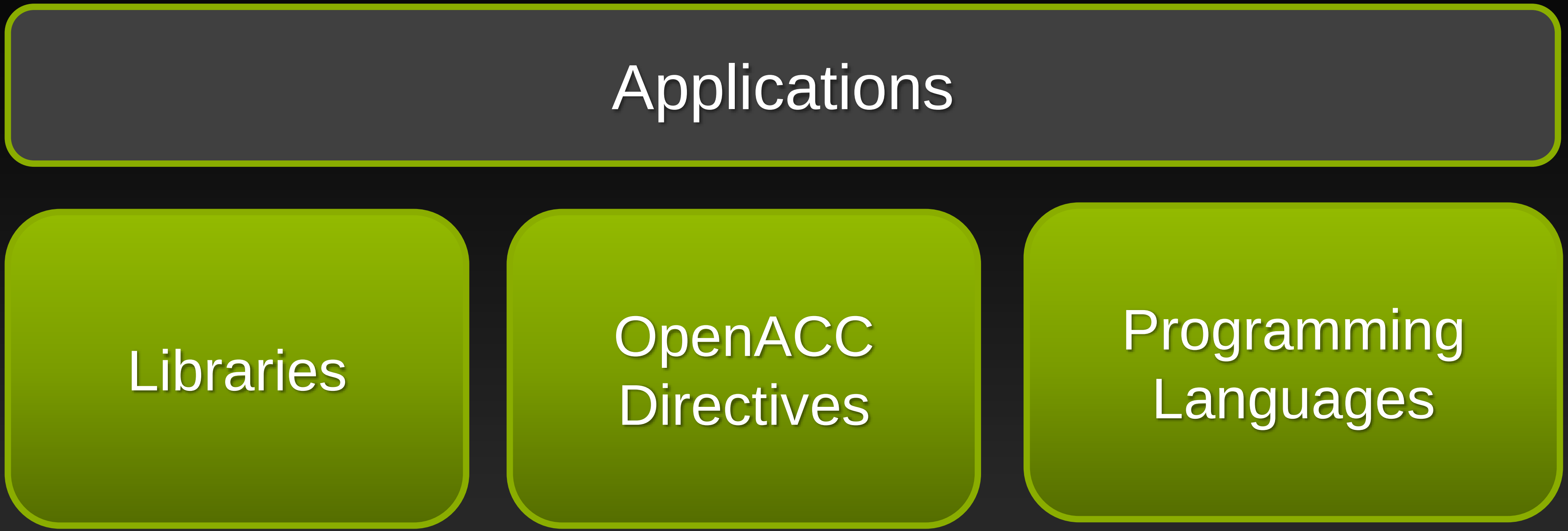


Assess



- Identify hotspots (total time, number of calls)
- Understand scaling (strong and weak)

Parallelise



Optimise

- Profile-driven optimisation

- Tools:

nsight	Visual Studio Edition or Eclipse Edition
nvvp	NVIDIA Visual Profiler
nvprof	Command-line profiling

Deploy

Productise

- Check API return values
- Run cuda-memcheck tools

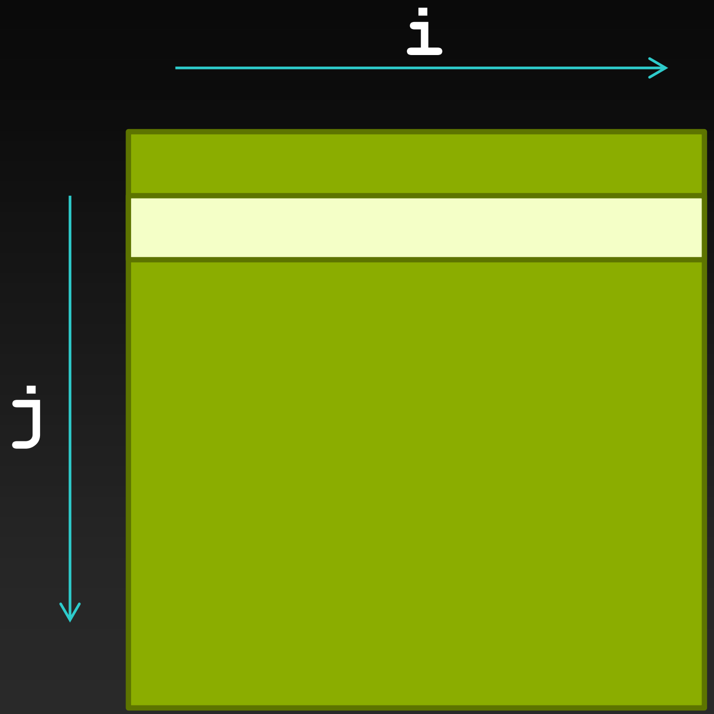
- Library distribution
- Cluster management

Early gains
Subsequent changes are evolutionary

PARALLELISE

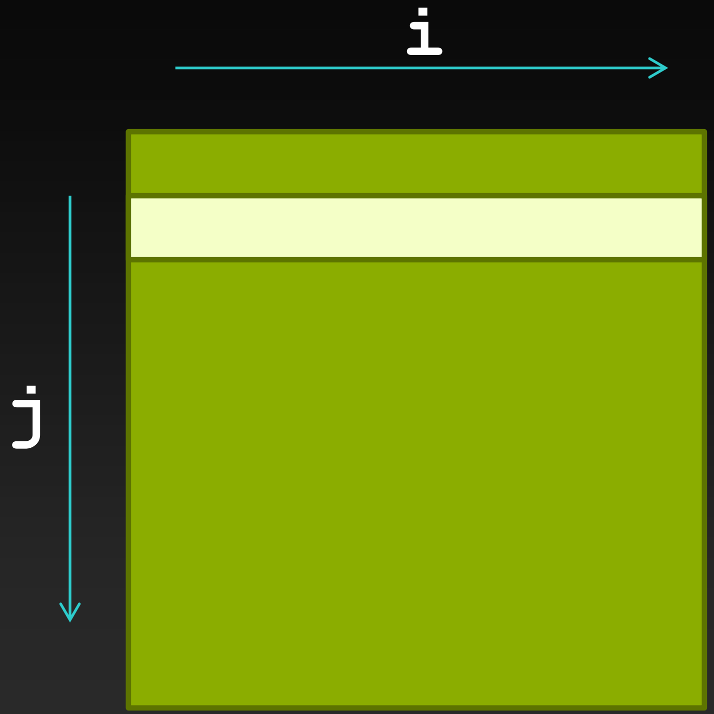
Case Study: Matrix Transpose

```
void transpose(float in[][], float out[][], int N)
{
  for(int j=0; j < N; j++)
    for(int i=0; i < N; i++)
      out[j][i] = in[i][j];
}
```



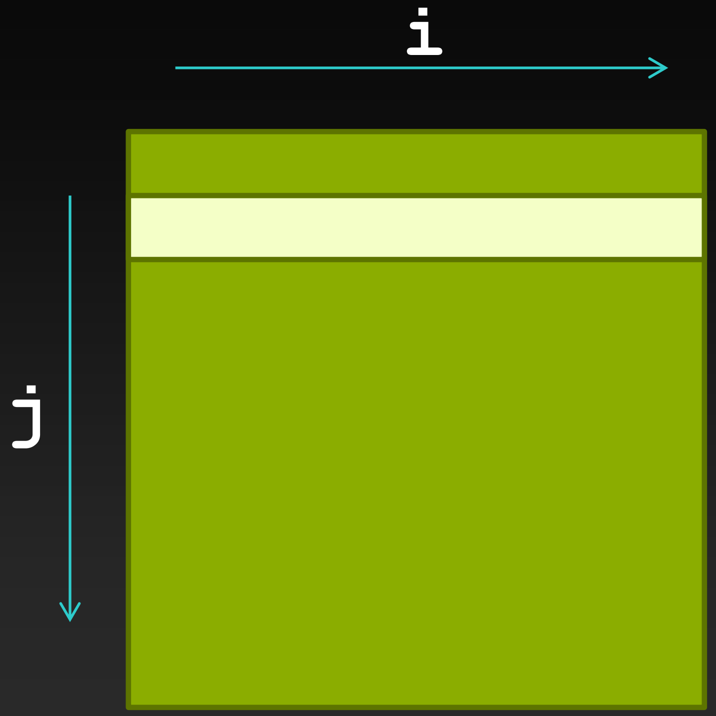
Case Study: Matrix Transpose

```
void transpose(float in[], float out[], int N)
{
  for(int j=0; j < N; j++)
    for(int i=0; i < N; i++)
      out[i*N+j] = in[j*N+i];
}

float in[N*N], out[N*N];




transpose(in, out, N);
```



An Initial CUDA Version

```
__global__ void transpose(float in[], float out[], int N)
{
  for(int j=0; j < N; j++)
    for(int i=0; i < N; i++)
      out[i*N+j] = in[j*N+i];
}

float in[N*N], out[N*N];
float *d_in, *d_out;

cudaMalloc(&d_in, sizeof(in));
cudaMalloc(&d_out, sizeof(out));
cudaMemcpy(d_in, in, sizeof(in));
transpose<<<1,1>>>(d_in, d_out, N);
cudaMemcpy(out, d_out, sizeof(out));
```



An Initial CUDA Version

```
__global__ void transpose(float in[], float out[], int N)
{
  for(int j=0; j < N; j++)
     for(int i=0; i < N; i++)
       out[i*N+j] = in[j*N+i];

}



float in[N*N], out[N*N];
…
transpose<<<1,1>>>(in, out, N);
```

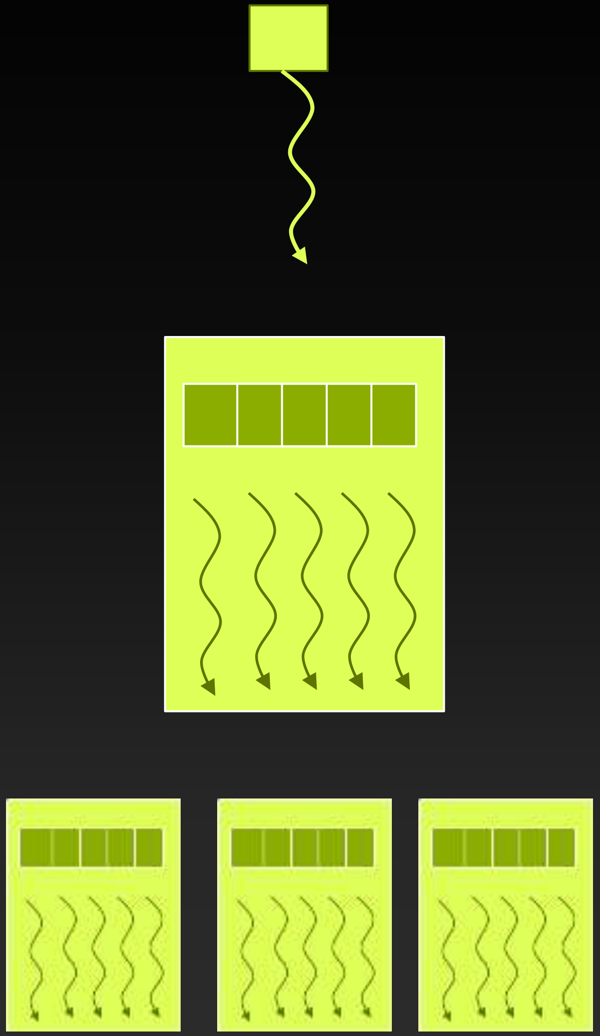
+ Quickly implemented

- Performance weak

⇒ **Need to expose parallelism!**

Review: CUDA Execution Model

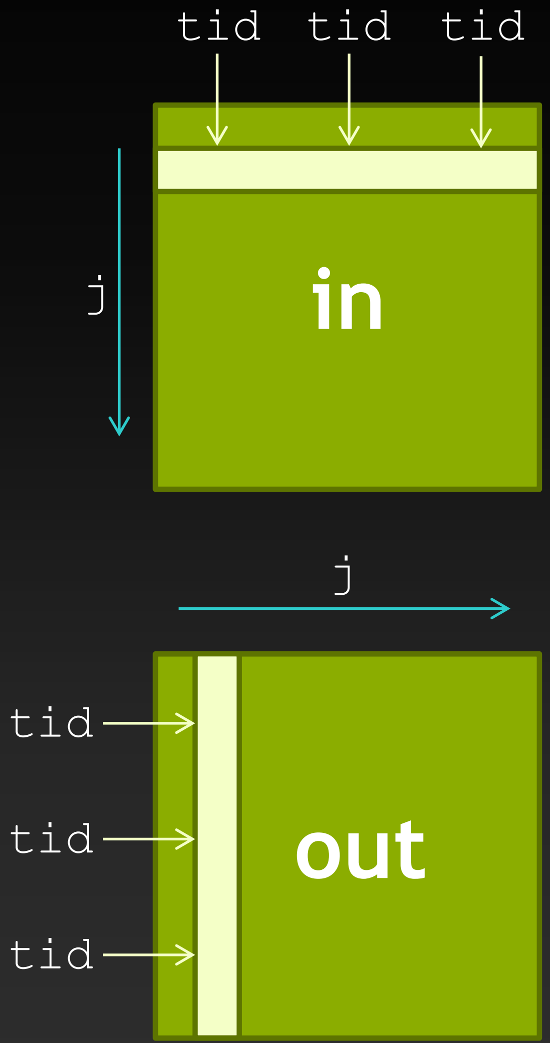
- Thread: Sequential execution unit
  - All threads execute same sequential program
  - Threads execute in parallel
- Threads Block: a group of threads
  - Executes on a single Streaming Multiprocessor (SM)
  - Threads within a block can cooperate
    - Light-weight synchronization
    - Data exchange
- Grid: a collection of thread blocks
  - Thread blocks of a grid execute across multiple SMs
  - Thread blocks do not synchronize with each other
  - Communication between blocks is expensive



First Parallelization: Inner Loop

Process input rows independently

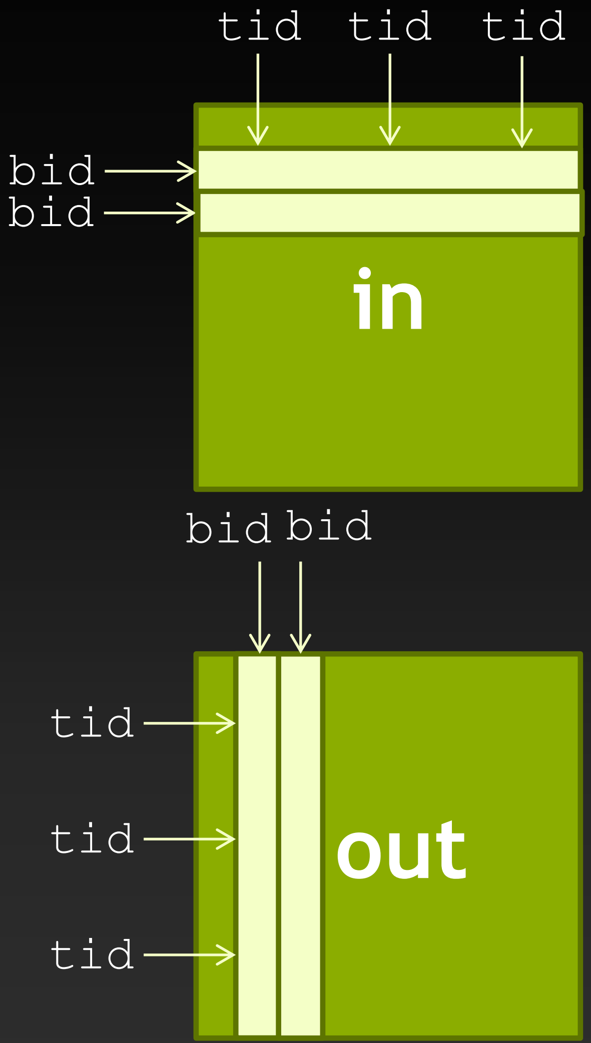
```
__global__ transpose(float in[], float out[])
{
  int tid = threadIdx.x;

  for(int j=0; j < N; j++)
    out[tid*N+j] = in[j*N+tid];
}


float in[], out[];
…
transpose<<<1,N>>>(in, out);
```



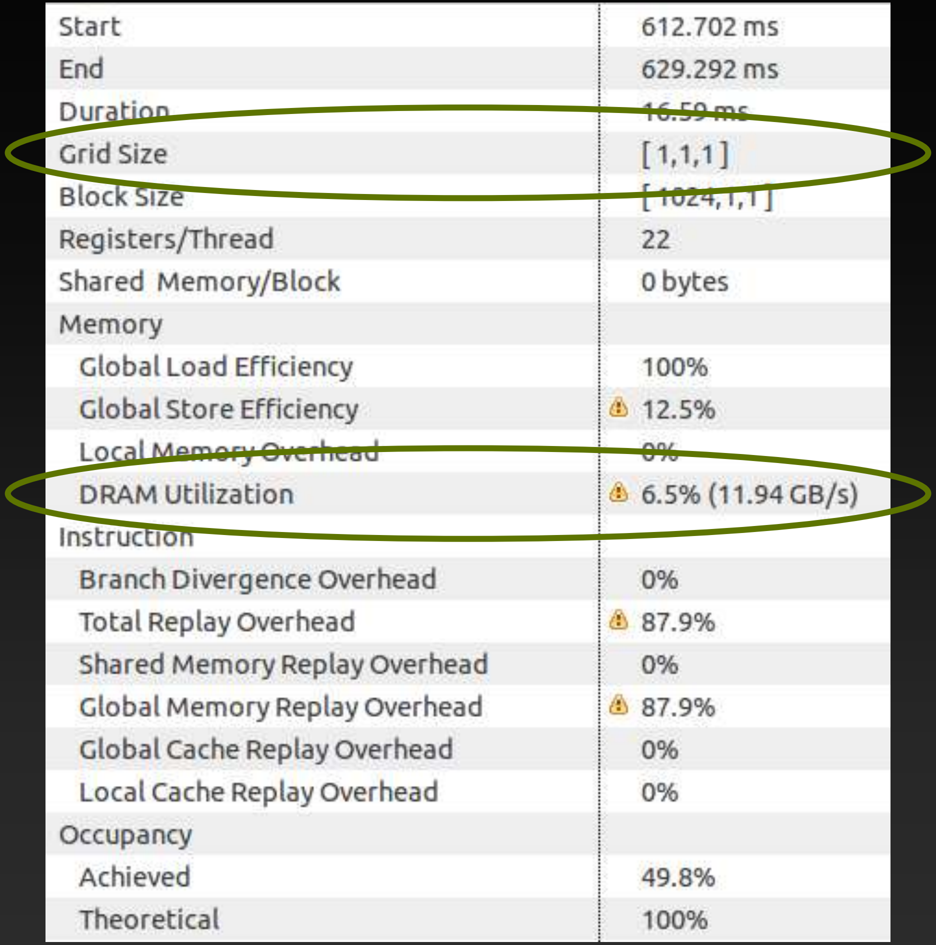
Second Parallelization: Outer Loop

Process elements independently

```
__global__ transpose(float in[], float out[])
{
  int tid = threadIdx.x;
  int bid = blockIdx.x;

  out[tid*N+bid] = in[bid*N+tid];
}


float in[], out[];
…
transpose<<<N,N>>>(in, out);
```



If we had ignored block-level parallelism…

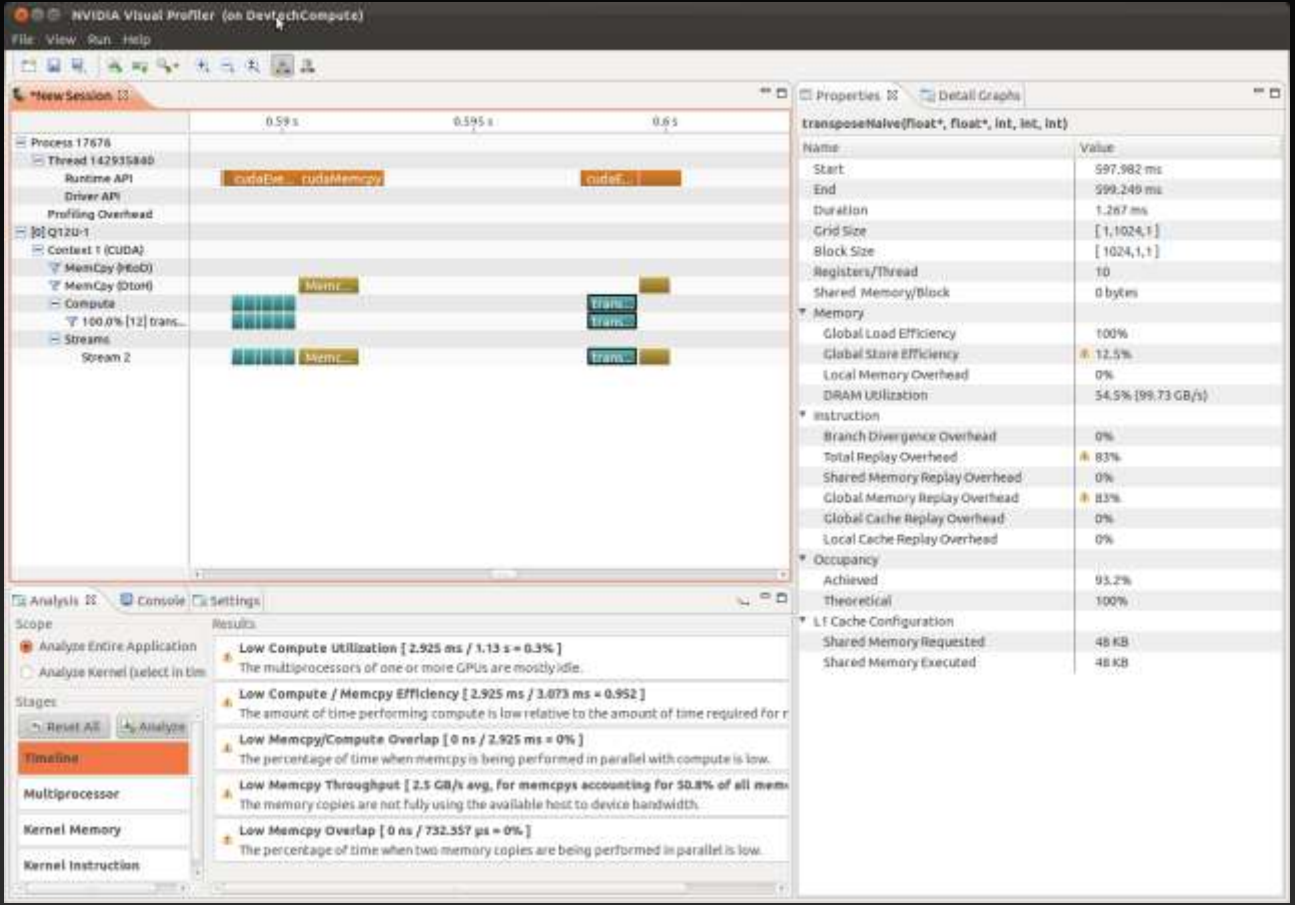


Start	612.702 ms
End	629.292 ms
Duration	16.59 ms
Grid Size	[1,1,1]
Block Size	[1024,1,1]
Registers/Thread	22
Shared Memory/Block	0 bytes
Memory	
Global Load Efficiency	100%
Global Store Efficiency	⚠ 12.5%
Local Memory Overhead	0%
DRAM Utilization	⚠ 6.5% (11.94 GB/s)
Instruction	
Branch Divergence Overhead	0%
Total Replay Overhead	⚠ 87.9%
Shared Memory Replay Overhead	0%
Global Memory Replay Overhead	⚠ 87.9%
Global Cache Replay Overhead	0%
Local Cache Replay Overhead	0%
Occupancy	
Achieved	49.8%
Theoretical	100%

OPTIMISE

NVIDIA Nsight

- CUDA Debugging
- CUDA Profiling



transposeNaive(float*, float*, int, int, int)

Name	Value
Start	597.982 ms
End	599.249 ms
Duration	1.267 ms
Grid Size	[1,1024,1]
Block Size	[1024,1,1]
Registers/Thread	10
Shared Memory/Block	0 bytes
Memory	
Global Load Efficiency	100%
Global Store Efficiency	12.5%
Local Memory Overhead	0%
DRAM Utilization	54.5% (99.73 GB/s)
Instruction	
Branch Divergence Overhead	0%
Total Replay Overhead	83%
Shared Memory Replay Overhead	0%
Global Memory Replay Overhead	83%
Global Cache Replay Overhead	0%
Local Cache Replay Overhead	0%
Occupancy	
Achieved	93.2%
Theoretical	100%
L1 Cache Configuration	
Shared Memory Requested	48 KB
Shared Memory Executed	48 KB

Low Compute Utilization [2.925 ms / 1.13 s = 0.3%]
The multiprocessors of one or more GPUs are mostly idle.

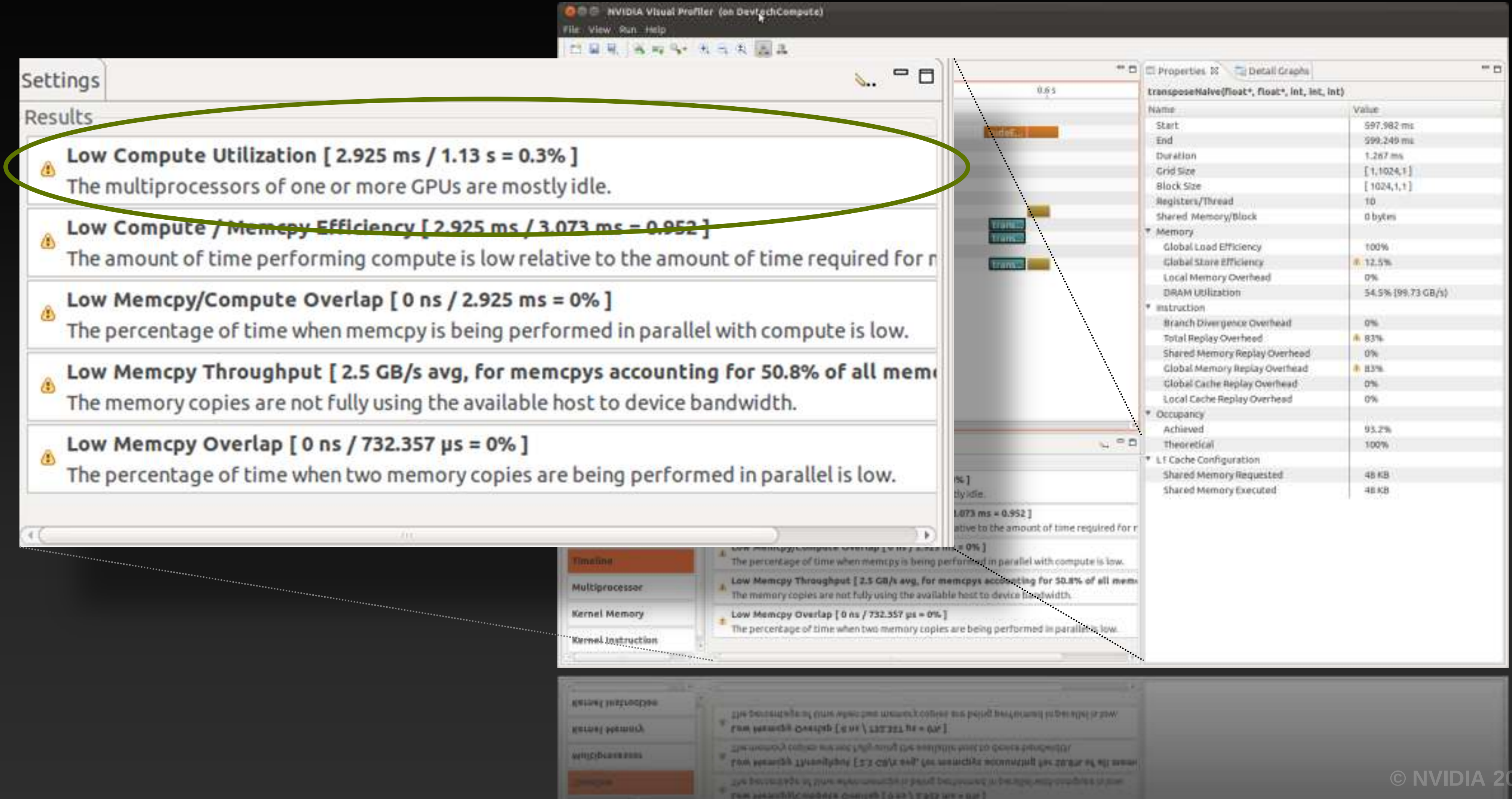
Low Compute / Memcpy Efficiency [2.925 ms / 3.073 ms = 0.952]
The amount of time performing compute is low relative to the amount of time required for r

Low Memcpy/Compute Overlap [0 ns / 2.925 ms = 0%]
The percentage of time when memcpy is being performed in parallel with compute is low.

Low Memcpy Throughput [2.5 GB/s avg, for memcpys accounting for 50.8% of all mem
The memory copies are not fully using the available host to device bandwidth.

Low Memcpy Overlap [0 ns / 732.357 µs = 0%]
The percentage of time when two memory copies are being performed in parallel is low.

Analysis-driven Optimization



Low Compute Utilization [2.925 ms / 1.13 s = 0.3%]
The multiprocessors of one or more GPUs are mostly idle.

Low Compute / Memcpy Efficiency [2.925 ms / 3.073 ms = 0.952]
The amount of time performing compute is low relative to the amount of time required for r

Low Memcpy/Compute Overlap [0 ns / 2.925 ms = 0%]
The percentage of time when memcpy is being performed in parallel with compute is low.

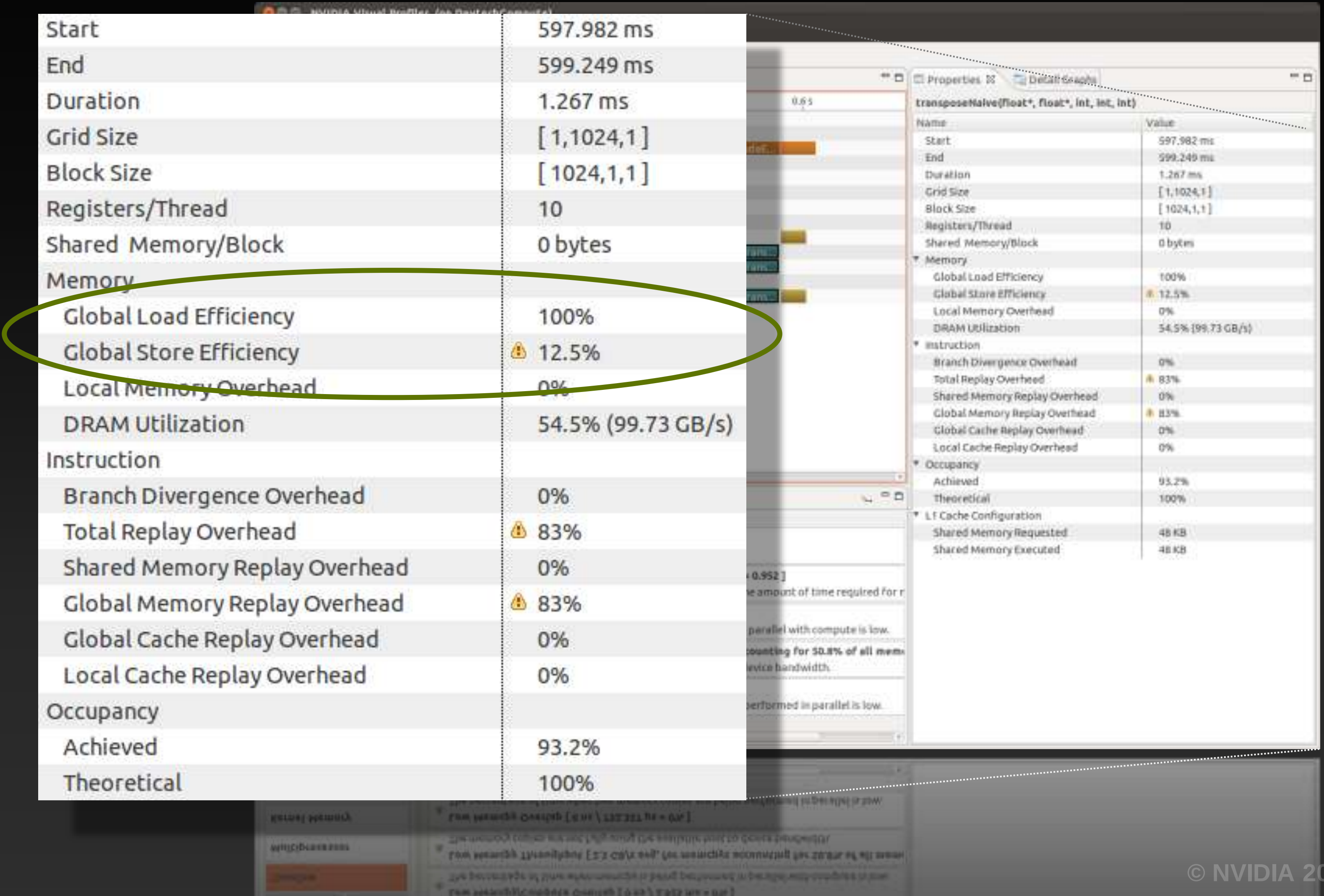
Low Memcpy Throughput [2.5 GB/s avg, for memcpys accounting for 50.8% of all mem
The memory copies are not fully using the available host to device bandwidth.

Low Memcpy Overlap [0 ns / 732.357 μs = 0%]
The percentage of time when two memory copies are being performed in parallel is low.

Properties — transposeNaive(float*, float*, int, int, int)

Name	Value
Start	597.982 ms
End	599.249 ms
Duration	1.267 ms
Grid Size	[1,1024,1]
Block Size	[1024,1,1]
Registers/Thread	10
Shared Memory/Block	0 bytes
Memory	
Global Load Efficiency	100%
Global Store Efficiency	⚠ 12.5%
Local Memory Overhead	0%
DRAM Utilization	54.5% (99.73 GB/s)
Instruction	
Branch Divergence Overhead	0%
Total Replay Overhead	⚠ 83%
Shared Memory Replay Overhead	0%
Global Memory Replay Overhead	⚠ 83%
Global Cache Replay Overhead	0%
Local Cache Replay Overhead	0%
Occupancy	
Achieved	93.2%
Theoretical	100%
L1 Cache Configuration	
Shared Memory Requested	48 KB
Shared Memory Executed	48 KB

Analysis-driven Optimization

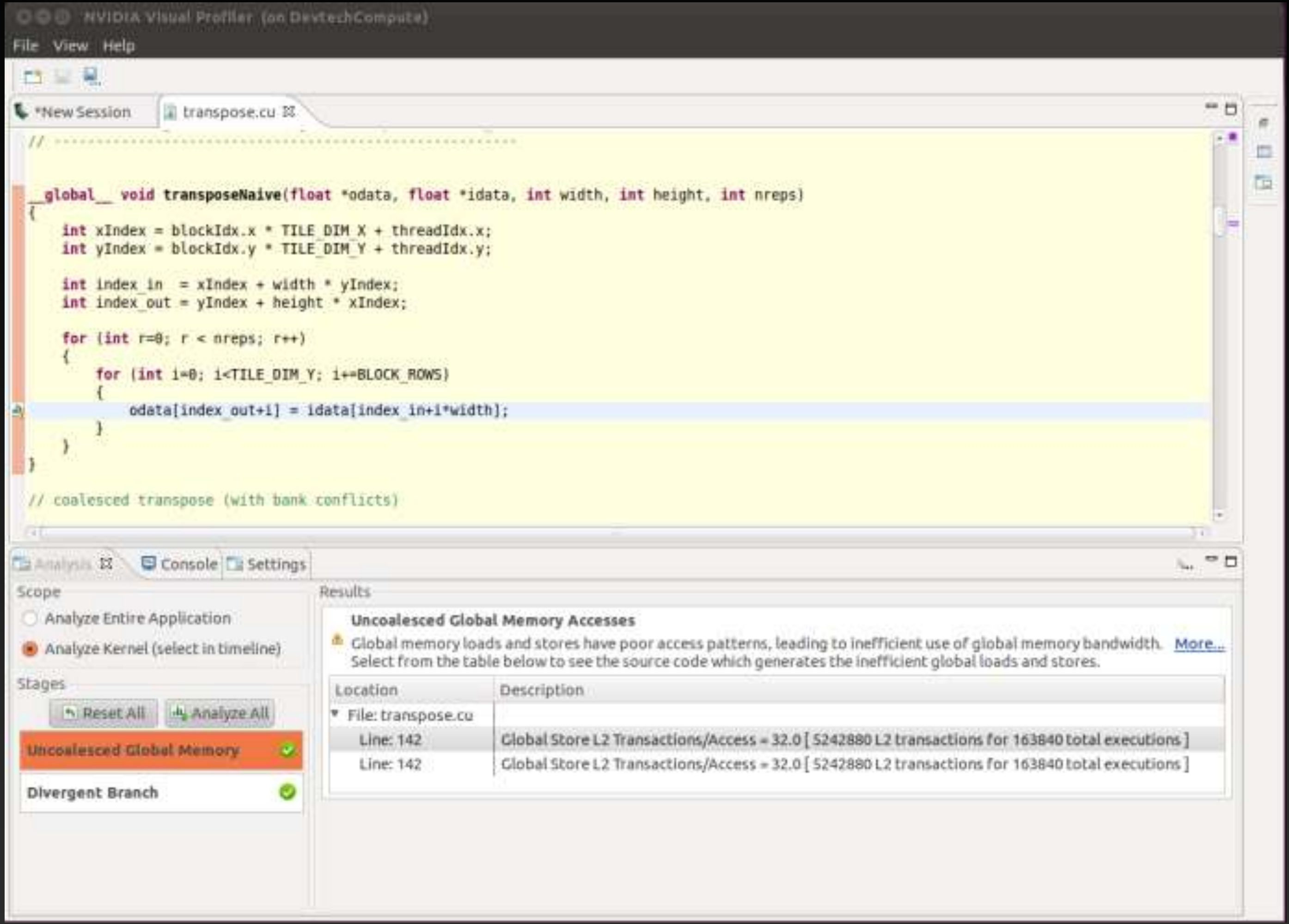


Start		597.982 ms
End		599.249 ms
Duration		1.267 ms
Grid Size		[1,1024,1]
Block Size		[1024,1,1]
Registers/Thread		10
Shared Memory/Block		0 bytes
Memory		
Global Load Efficiency		100%
Global Store Efficiency	⚠	12.5%
Local Memory Overhead		0%
DRAM Utilization		54.5% (99.73 GB/s)
Instruction		
Branch Divergence Overhead		0%
Total Replay Overhead	⚠	83%
Shared Memory Replay Overhead		0%
Global Memory Replay Overhead	⚠	83%
Global Cache Replay Overhead		0%
Local Cache Replay Overhead		0%
Occupancy		
Achieved		93.2%
Theoretical		100%

transposeNaive(float*, float*, int, int, int)

Name	Value
L1 Cache Configuration	
Shared Memory Requested	48 KB
Shared Memory Executed	48 KB

Source-Level Hotspot Analysis in Nsight



```
// ---------------------------------------------------

__global__ void transposeNaive(float *odata, float *idata, int width, int height, int nreps)
{
  int xIndex = blockIdx.x * TILE_DIM_X + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM_Y + threadIdx.y;

  int index_in  = xIndex + width * yIndex;
  int index_out = yIndex + height * xIndex;

  for (int r=0; r < nreps; r++)
  {
    for (int i=0; i<TILE_DIM_Y; i+=BLOCK_ROWS)
    {
      odata[index_out+i] = idata[index_in+i*width];
    }
  }
}

// coalesced transpose (with bank conflicts)
```

Scope

○ Analyze Entire Application

● Analyze Kernel (select in timeline)

Stages

Reset All | Analyze All

Uncoalesced Global Memory ✓

Divergent Branch ✓

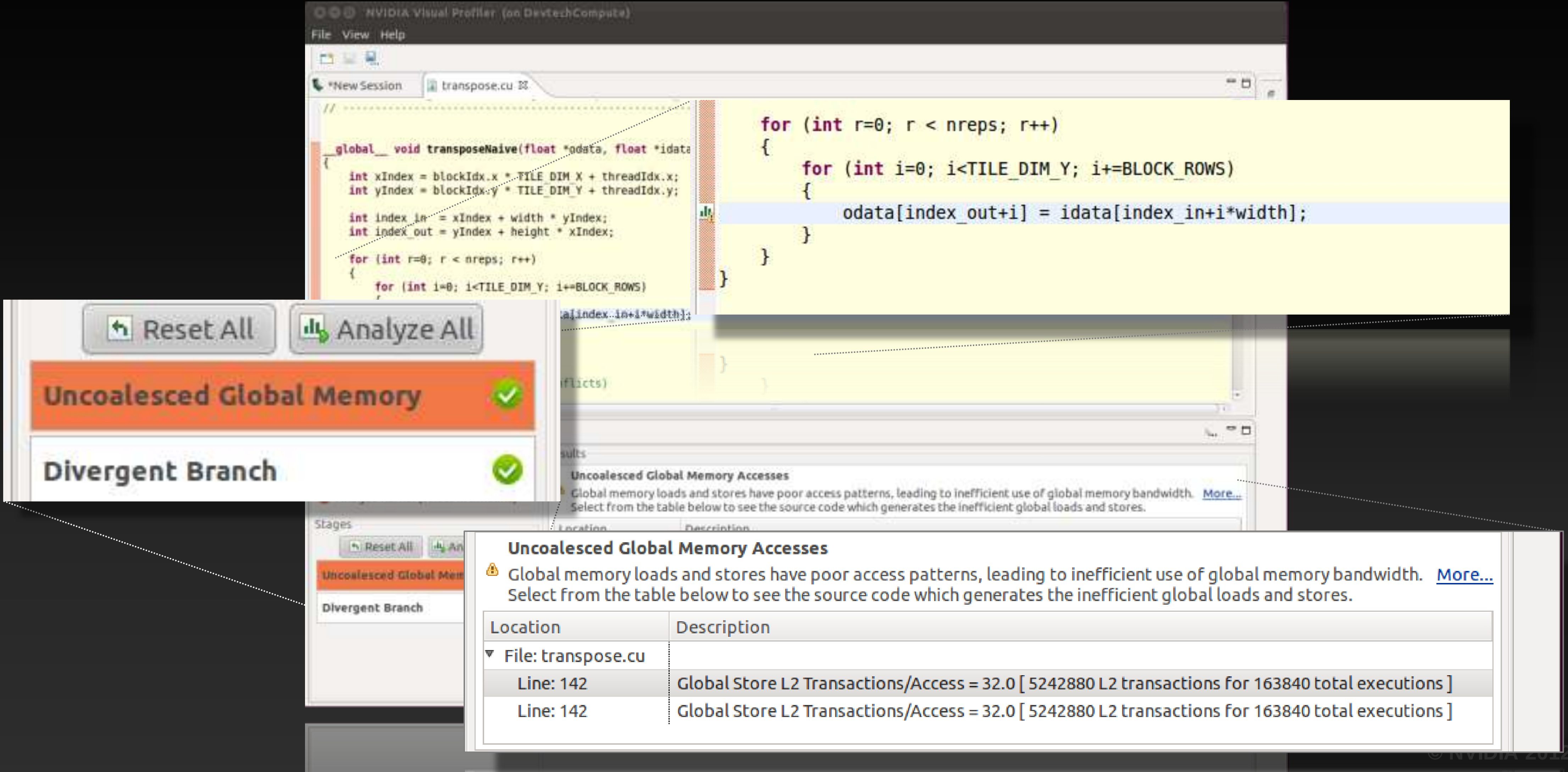
Results

Uncoalesced Global Memory Accesses

Global memory loads and stores have poor access patterns, leading to inefficient use of global memory bandwidth. More...
Select from the table below to see the source code which generates the inefficient global loads and stores.

Location	Description
File: transpose.cu	
Line: 142	Global Store L2 Transactions/Access = 32.0 [5242880 L2 transactions for 163840 total executions]
Line: 142	Global Store L2 Transactions/Access = 32.0 [5242880 L2 transactions for 163840 total executions]

Source-Level Hotspot Analysis in Nsight



```
__global__ void transposeNaive(float *odata, float *idata
{
    int xIndex = blockIdx.x * TILE_DIM_X + threadIdx.x;
    int yIndex = blockIdx.y * TILE_DIM_Y + threadIdx.y;

    int index_in  = xIndex + width * yIndex;
    int index_out = yIndex + height * xIndex;

    for (int r=0; r < nreps; r++)
    {
        for (int i=0; i<TILE_DIM_Y; i+=BLOCK_ROWS)
```

```
    for (int r=0; r < nreps; r++)
    {
        for (int i=0; i<TILE_DIM_Y; i+=BLOCK_ROWS)
        {
            odata[index_out+i] = idata[index_in+i*width];
        }
    }
}
```

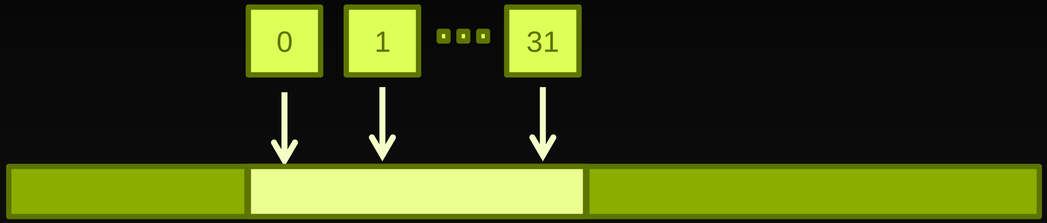
Uncoalesced Global Memory Accesses

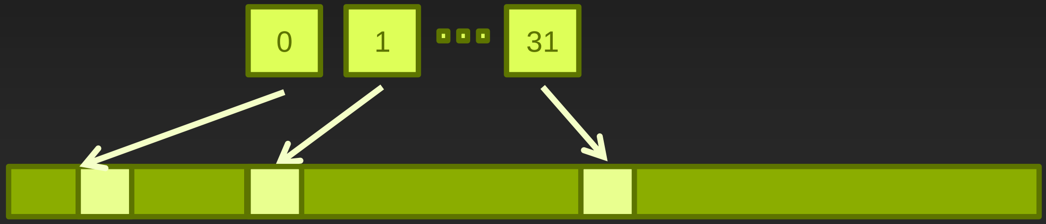
Global memory loads and stores have poor access patterns, leading to inefficient use of global memory bandwidth. More...
Select from the table below to see the source code which generates the inefficient global loads and stores.

Location	Description
File: transpose.cu	
Line: 142	Global Store L2 Transactions/Access = 32.0 [5242880 L2 transactions for 163840 total executions]
Line: 142	Global Store L2 Transactions/Access = 32.0 [5242880 L2 transactions for 163840 total executions]

What is an Uncoalesced Global Store?

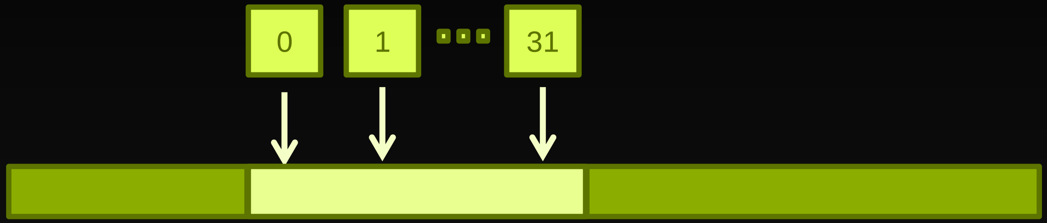
- Global memory access happens in transactions of 32 words
- *Coalesced* access:
  - A group of 32 contiguous threads (“warp”) accessing adjacent words
- *Uncoalesced* access:
  - A warp of 32 threads accessing scattered words
  - Results in 2..32 transactions

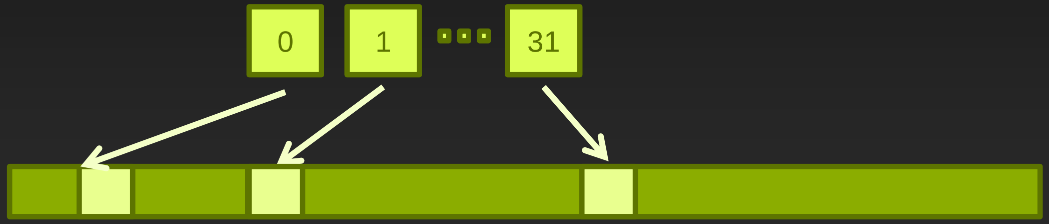




Global Memory Access Patterns

- SoA vs AoS:
  - Good: point.x[i]
  - Not so good: point[i].x
- Strided array access:
  - ~OK: x[i] = a[i+1] – a[i]
  - Slower: x[i] = a[64*i] – a[i]
- Random array access:
  - Slower: a[rand(i)]



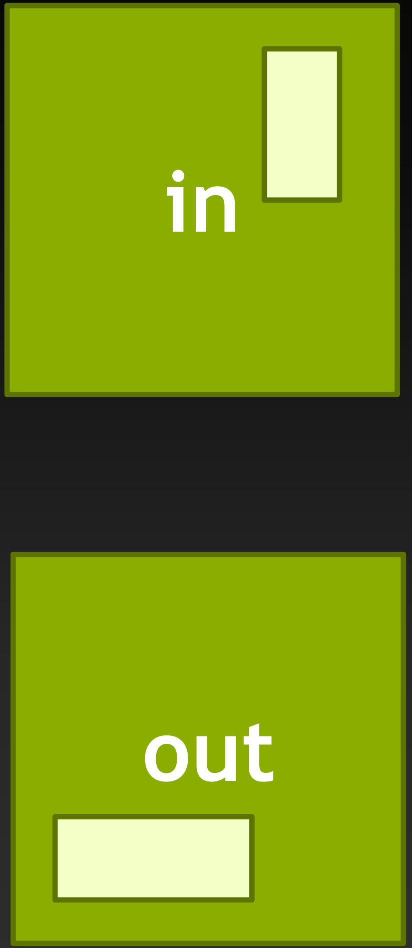


How can we improve the writes?

- Coalesced read
- Scattered write (stride N)

⇒ Process matrix tile, not single row/column, per block

⇒ Transpose matrix tile within block

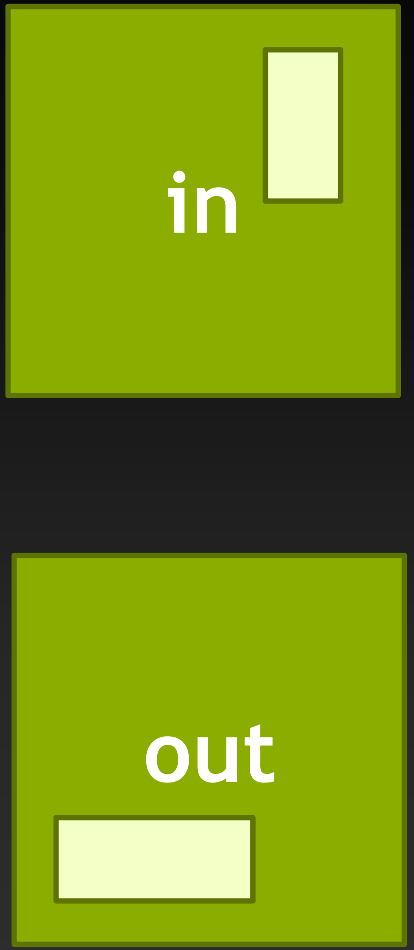


How can we improve the writes?

- Coalesced read
- Scattered write (stride N)

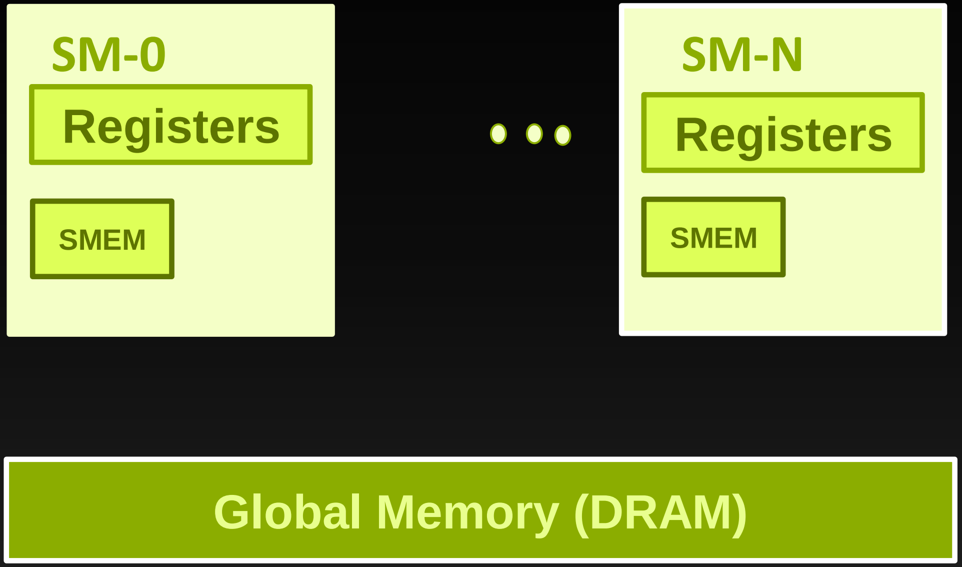
- Transpose matrix tile within block

⇒ Need threads in a block to cooperate:
use shared memory



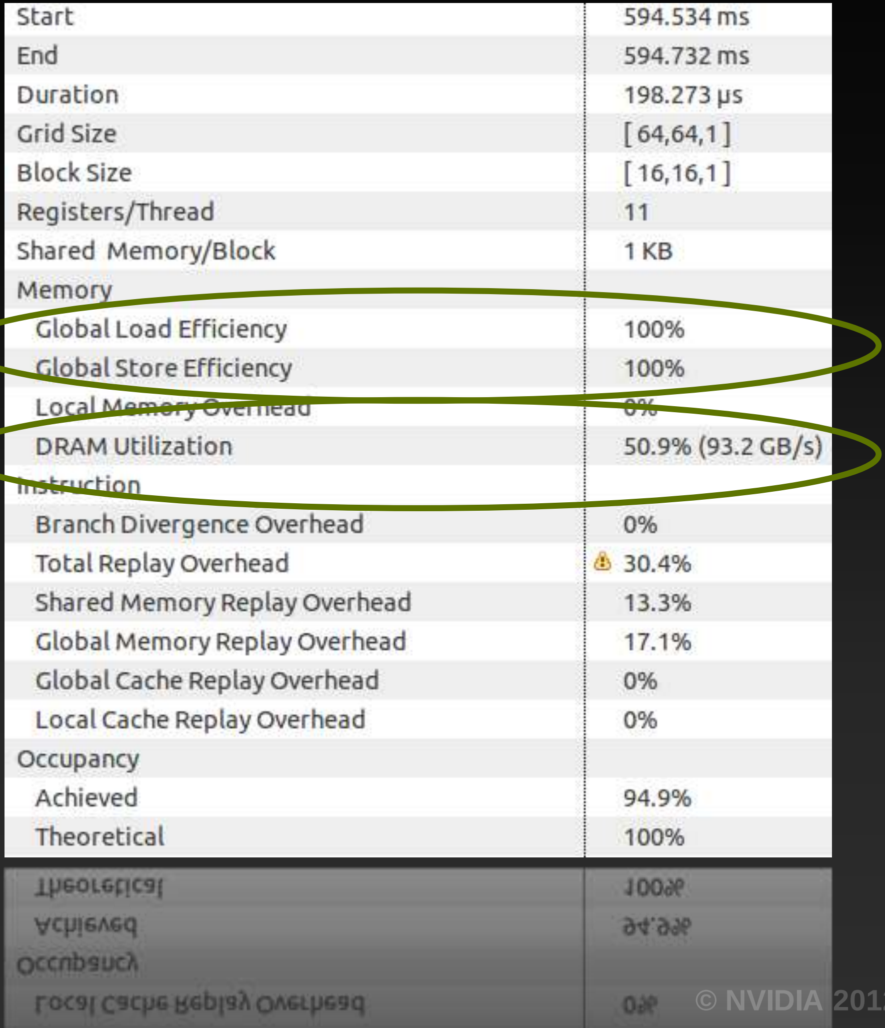
Shared memory

- Accessible by all threads in a block
- Fast compared to global memory
  - Low access latency
  - High bandwidth
- Common uses:
  - Software managed cache
  - Data layout conversion



Transpose with coalesced read/write

```
__global__ transpose(float in[], float out[])
{

  __shared__ float tile[TILE][TILE];

  int glob_in = xIndex + (yIndex)*N;
  int glob_out = xIndex + (yIndex)*N;

  tile[threadIdx.y][threadIdx.x] = in[glob_in];

  __syncthreads();

  out[glob_out] = tile[threadIdx.x][threadIdx.y];

}


grid(N/TILE, N/TILE,1)
threads(TILE, TILE, 1)
transpose<<<grid, threads>>>(in, out);
```



Start	594.534 ms
End	594.732 ms
Duration	198.273 μs
Grid Size	[64,64,1]
Block Size	[16,16,1]
Registers/Thread	11
Shared Memory/Block	1 KB
Memory	
Global Load Efficiency	100%
Global Store Efficiency	100%
Local Memory Overhead	0%
DRAM Utilization	50.9% (93.2 GB/s)
Instruction	
Branch Divergence Overhead	0%
Total Replay Overhead	⚠ 30.4%
Shared Memory Replay Overhead	13.3%
Global Memory Replay Overhead	17.1%
Global Cache Replay Overhead	0%
Local Cache Replay Overhead	0%
Occupancy	
Achieved	94.9%
Theoretical	100%

Why did our DRAM Utilization decrease?

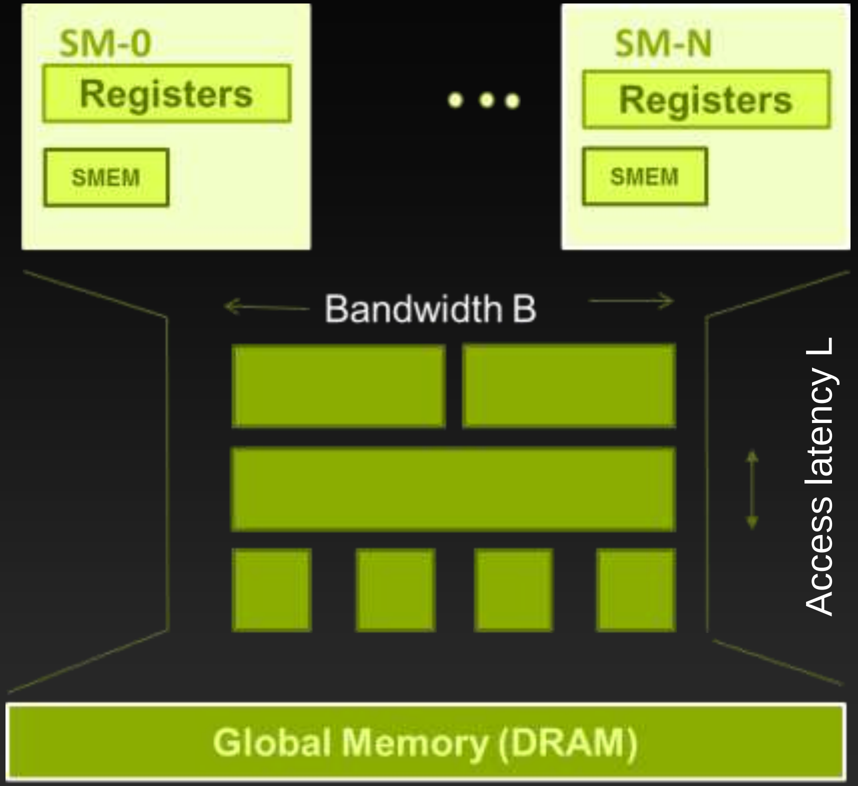
- **Goal: utilize all available memory bandwidth**
- **Little's Law:**

  # bytes in flight = latency * bandwidth

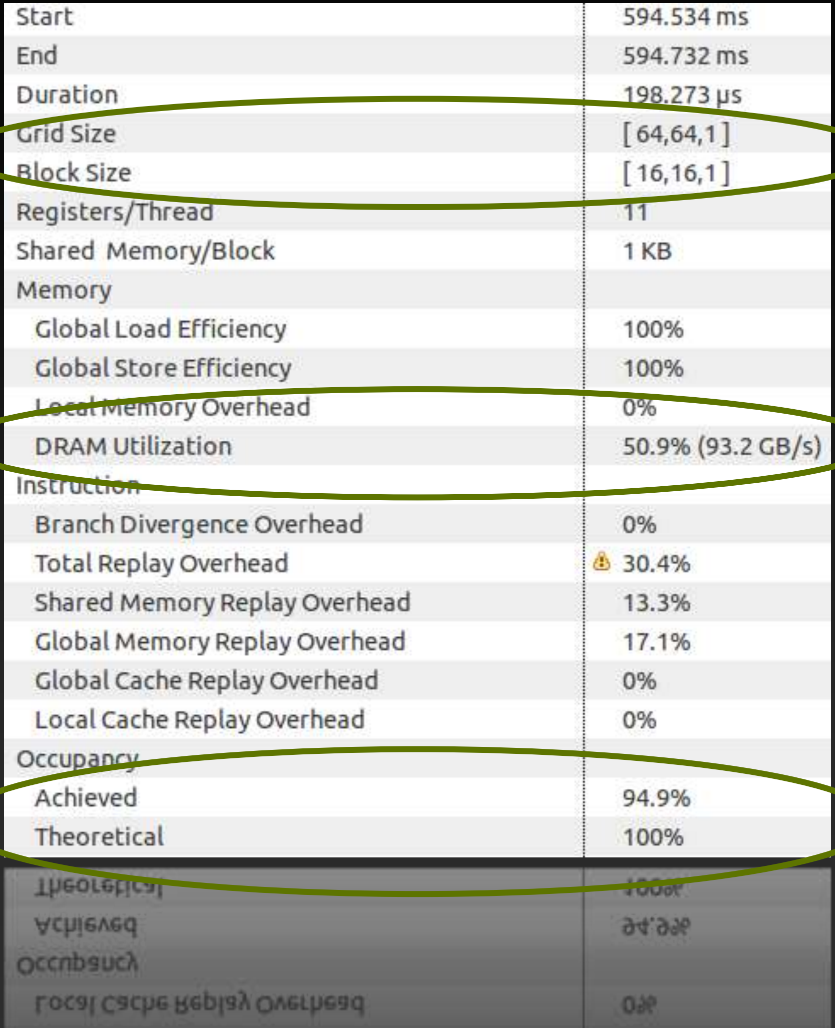
⇒ Increase parallelism (number of threads)

(or)

⇒ Reduce interval (time between requests)



Latency problems, possibly?



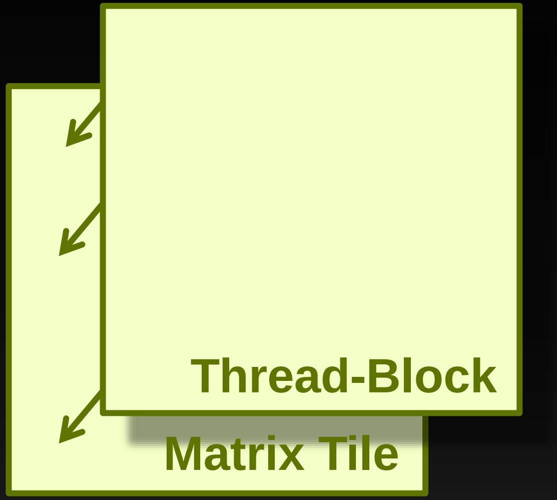
Start	594.534 ms
End	594.732 ms
Duration	198.273 μs
Grid Size	[64,64,1]
Block Size	[16,16,1]
Registers/Thread	11
Shared Memory/Block	1 KB
Memory	
Global Load Efficiency	100%
Global Store Efficiency	100%
Local Memory Overhead	0%
DRAM Utilization	50.9% (93.2 GB/s)
Instruction	
Branch Divergence Overhead	0%
Total Replay Overhead	⚠ 30.4%
Shared Memory Replay Overhead	13.3%
Global Memory Replay Overhead	17.1%
Global Cache Replay Overhead	0%
Local Cache Replay Overhead	0%
Occupancy	
Achieved	94.9%
Theoretical	100%

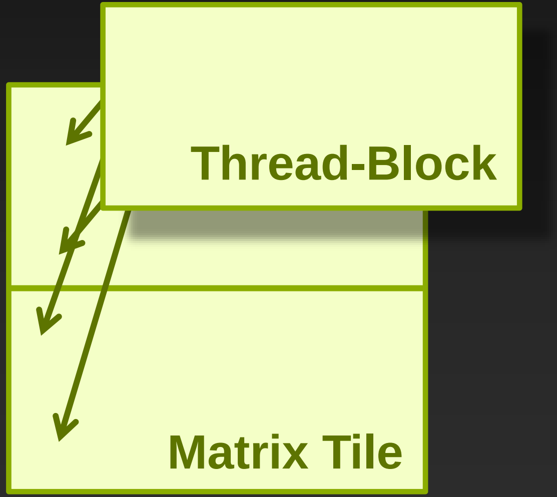
Barrier Synchronization Latency

- **Synchronization in kernel:**

```
tile[y][x] = in[in_data];
__syncthreads();
out[out_index] = tile[x][y];
```

⇒ Keep threads blocked at the barrier to a minimum





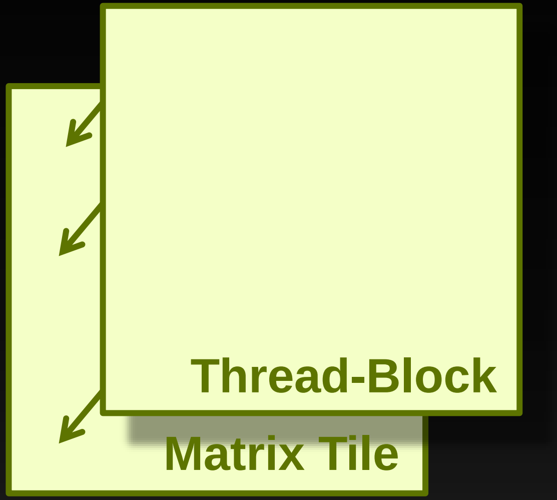
Barrier Synchronization Latency

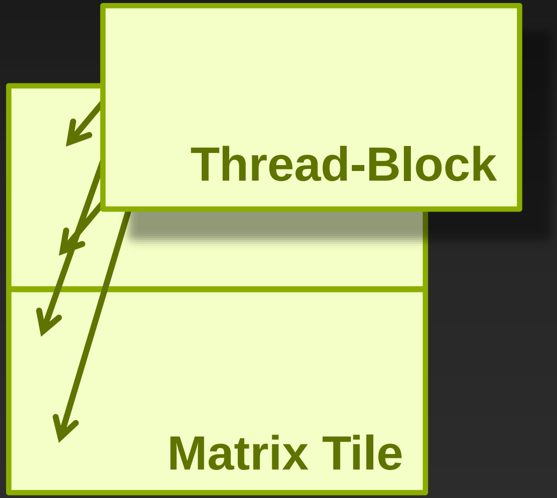
- **Synchronization in kernel:**

```
tile[y][x] = in[in_data];
__syncthreads();
out[out_index] = tile[x][y];
```

- **Keep threads blocked at the barrier to a minimum**

⇒ Use more thread blocks, but # blocks per SM is limited by # threads per block





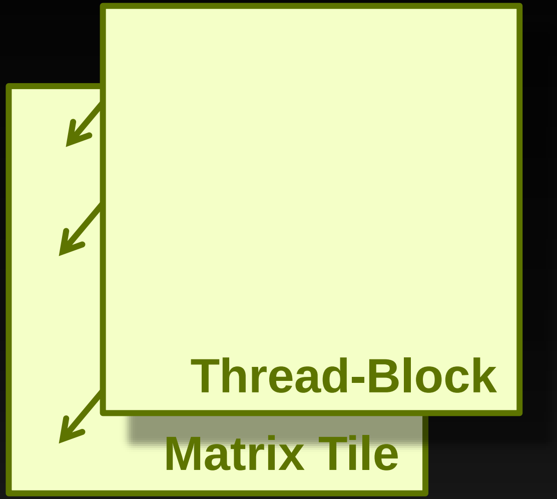
Barrier Synchronization Latency

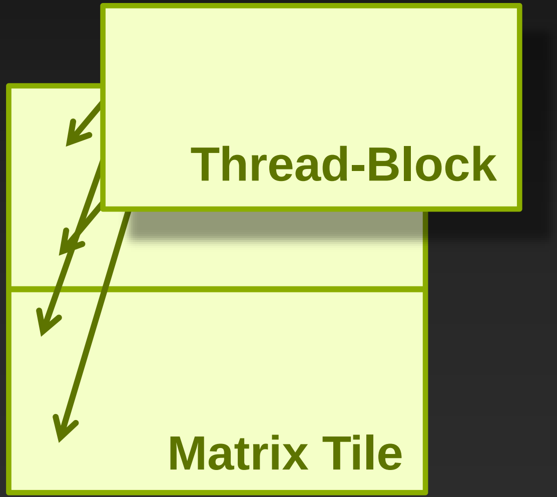
- **Synchronization in kernel:**

```
    tile[y][x] = in[in_data];
    __syncthreads();
    out[out_index] = tile[x][y];
```

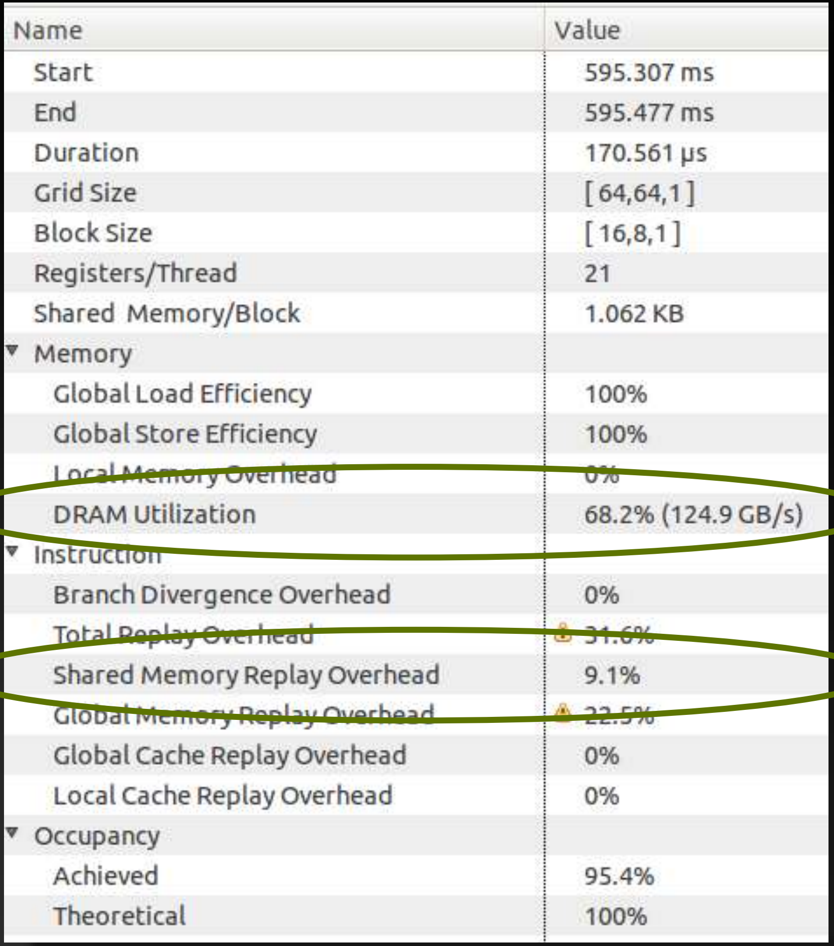
- **Use more thread blocks, but # blocks per SM is limited by # threads per block**

⇒ Solution for transpose: reduce number of threads per block





Shared Memory Replay (Bank Conflict) Latency

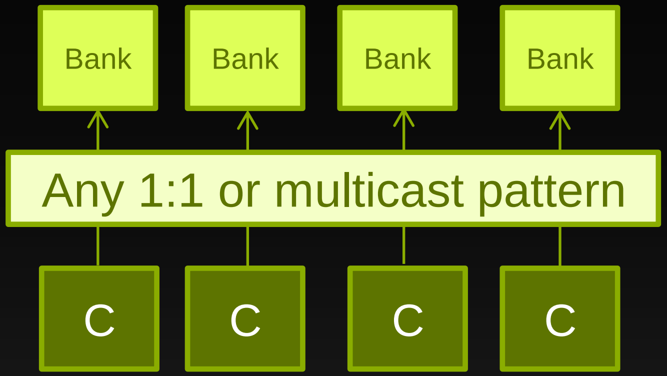


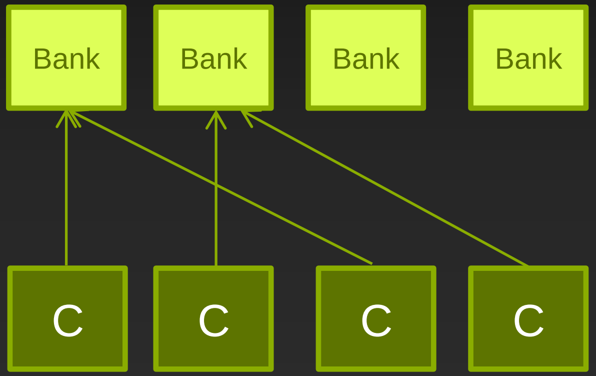
Name	Value
Start	595.307 ms
End	595.477 ms
Duration	170.561 μs
Grid Size	[64,64,1]
Block Size	[16,8,1]
Registers/Thread	21
Shared Memory/Block	1.062 KB
▾ Memory	
Global Load Efficiency	100%
Global Store Efficiency	100%
Local Memory Overhead	0%
DRAM Utilization	68.2% (124.9 GB/s)
▾ Instruction	
Branch Divergence Overhead	0%
Total Replay Overhead	⚠ 31.6%
Shared Memory Replay Overhead	9.1%
Global Memory Replay Overhead	⚠ 22.5%
Global Cache Replay Overhead	0%
Local Cache Replay Overhead	0%
▾ Occupancy	
Achieved	95.4%
Theoretical	100%

Shared Memory Organization

- Organized in 32 independent banks
- Optimal access: no two words from same bank
  - Separate banks per thread
  - Banks can multicast
- Multiple words from same bank serialize

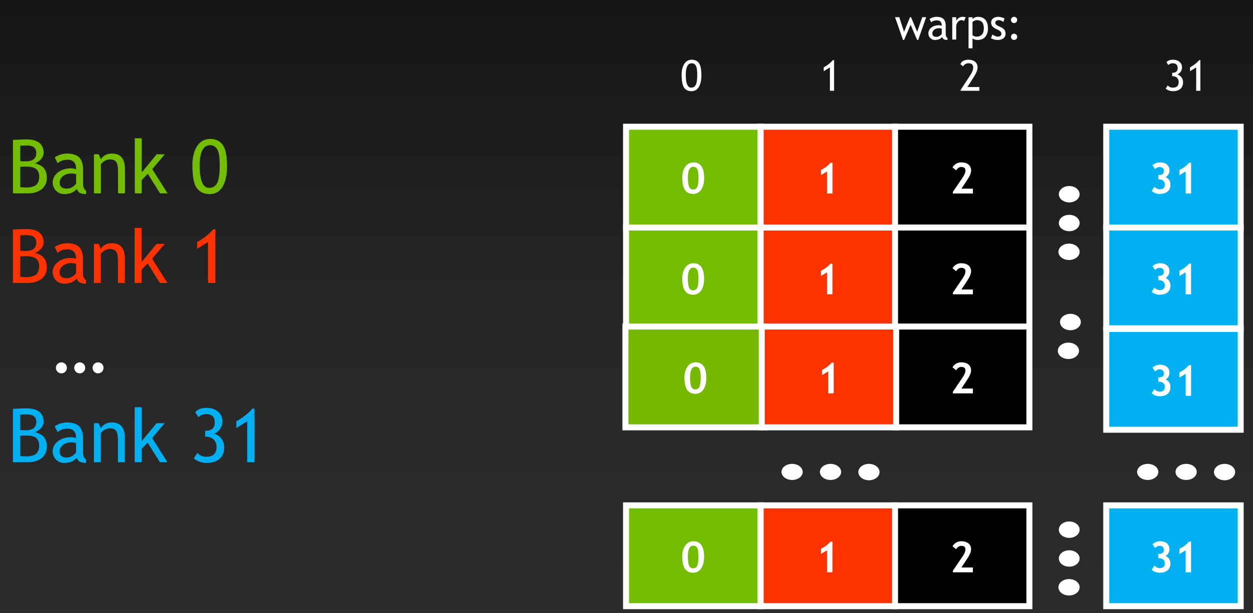
⇒ Solution for transpose: padding tile[16][16] => tile[16][17]





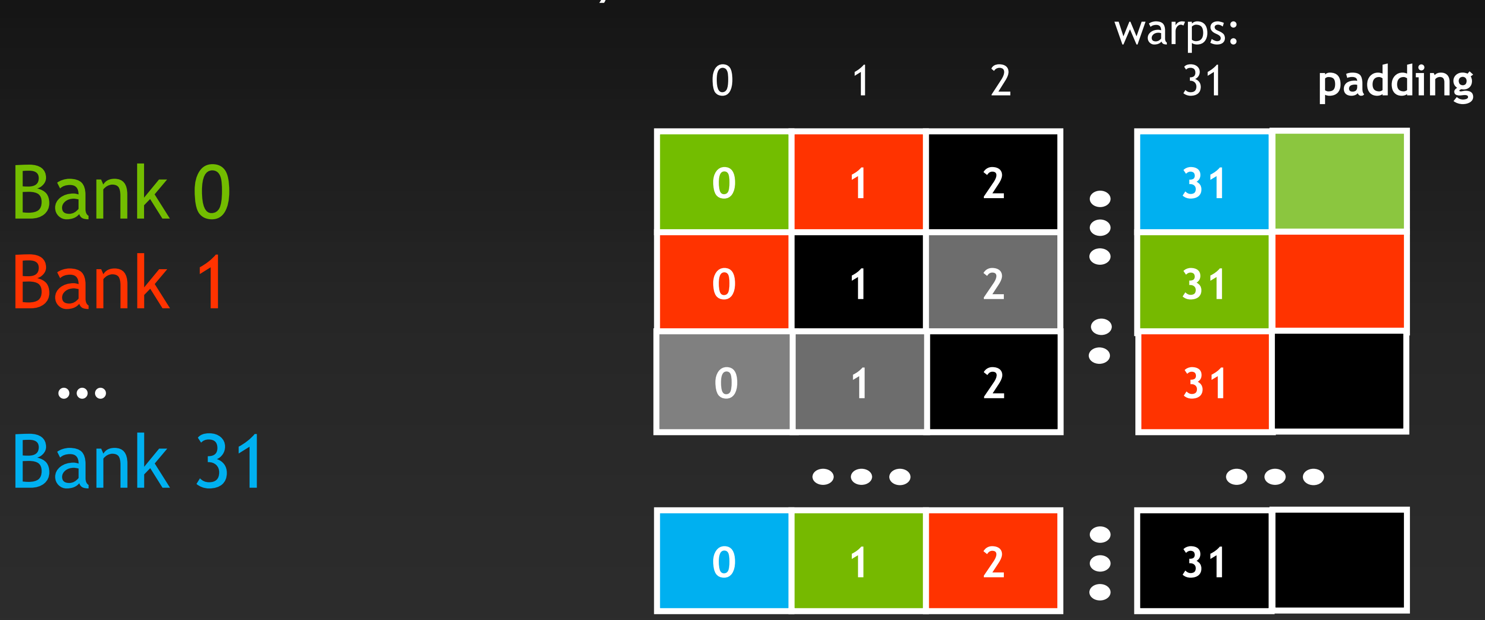
Shared Memory: Avoiding Bank Conflicts

- Example: 32x32 SMEM array
- Warp accesses a column:
  - 32-way bank conflicts (threads in a warp access the same bank)

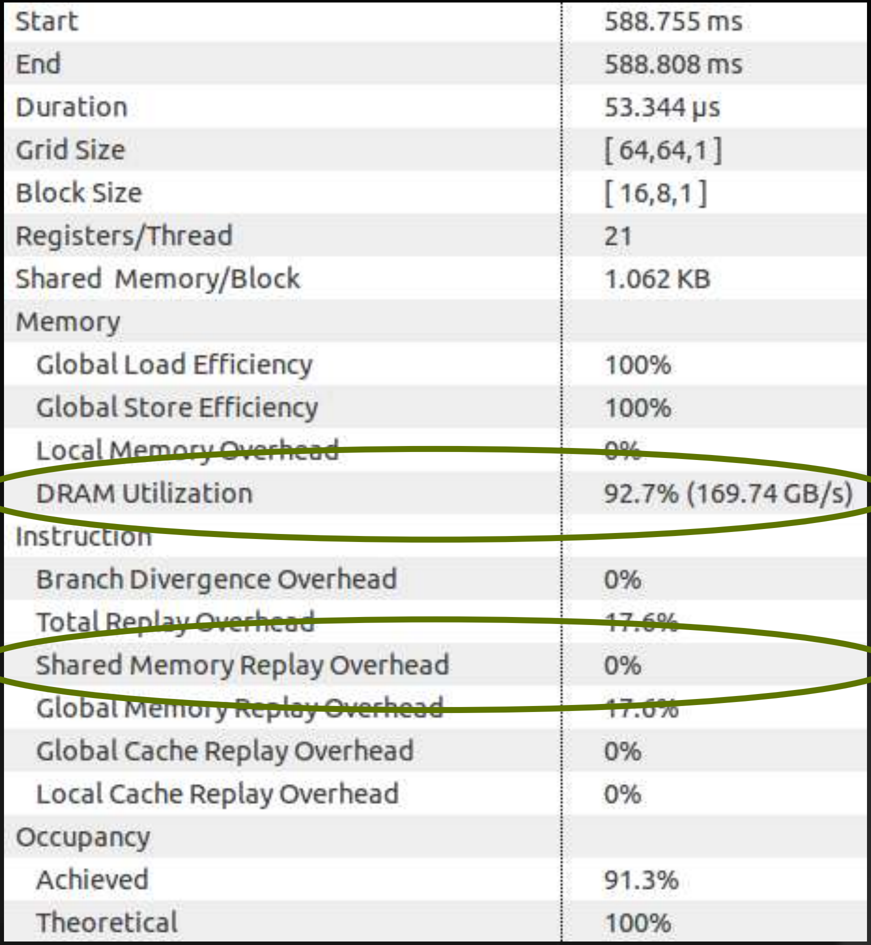


Shared Memory: Avoiding Bank Conflicts

- Add a column for padding:
  - 32x33 SMEM array
- Warp accesses a column:
  - 32 different banks, no bank conflicts



Final Solution



Start	588.755 ms
End	588.808 ms
Duration	53.344 µs
Grid Size	[64,64,1]
Block Size	[16,8,1]
Registers/Thread	21
Shared Memory/Block	1.062 KB
Memory	
Global Load Efficiency	100%
Global Store Efficiency	100%
Local Memory Overhead	0%
DRAM Utilization	92.7% (169.74 GB/s)
Instruction	
Branch Divergence Overhead	0%
Total Replay Overhead	17.6%
Shared Memory Replay Overhead	0%
Global Memory Replay Overhead	17.6%
Global Cache Replay Overhead	0%
Local Cache Replay Overhead	0%
Occupancy	
Achieved	91.3%
Theoretical	100%

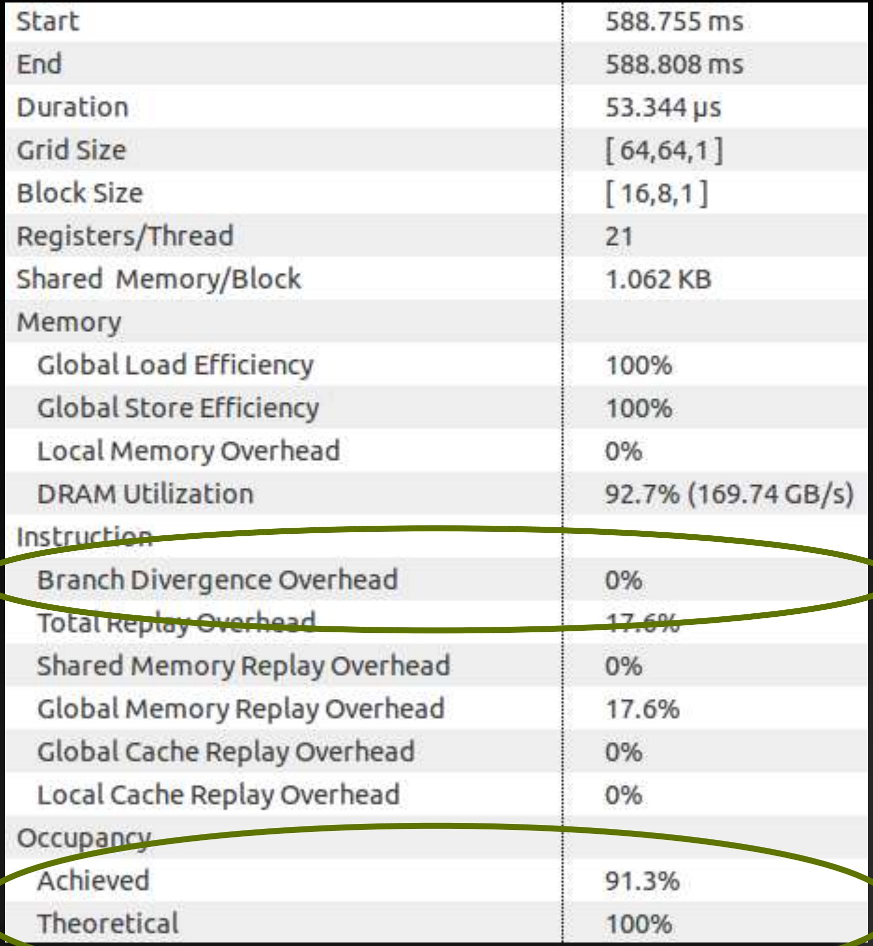
Optimization Summary

1 thread per column	12 GB/s
1 thread per element	99 GB/s
Memory accesses coalesced	93 GB/s
Latency hiding	124 GB/s
Bank conflict resolution	170 GB/s

⇒ Ready for Deployment

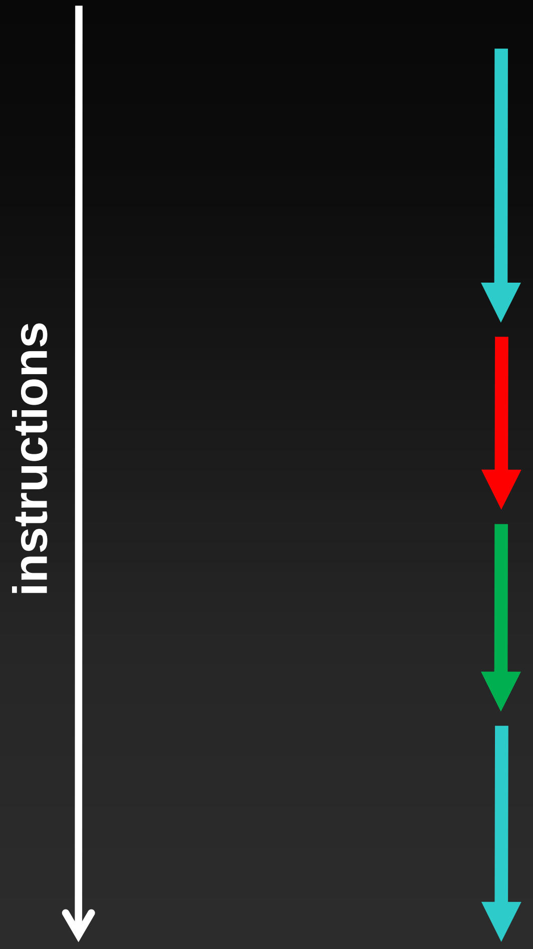
Optimization steps were profile-guided

Additional Metrics



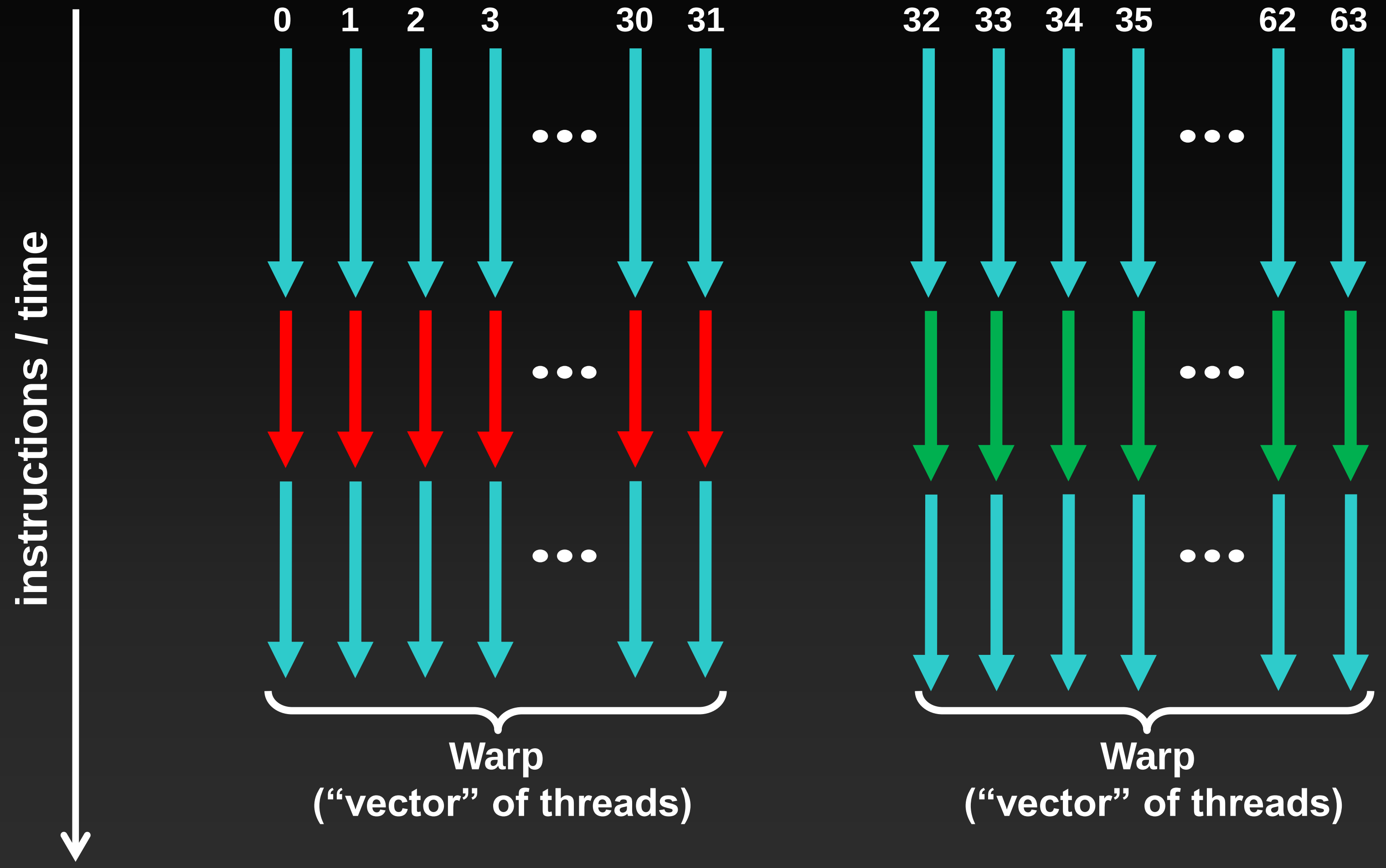
Start	588.755 ms
End	588.808 ms
Duration	53.344 μs
Grid Size	[64,64,1]
Block Size	[16,8,1]
Registers/Thread	21
Shared Memory/Block	1.062 KB
Memory	
Global Load Efficiency	100%
Global Store Efficiency	100%
Local Memory Overhead	0%
DRAM Utilization	92.7% (169.74 GB/s)
Instruction	
Branch Divergence Overhead	0%
Total Replay Overhead	17.6%
Shared Memory Replay Overhead	0%
Global Memory Replay Overhead	17.6%
Global Cache Replay Overhead	0%
Local Cache Replay Overhead	0%
Occupancy	
Achieved	91.3%
Theoretical	100%

Control Flow

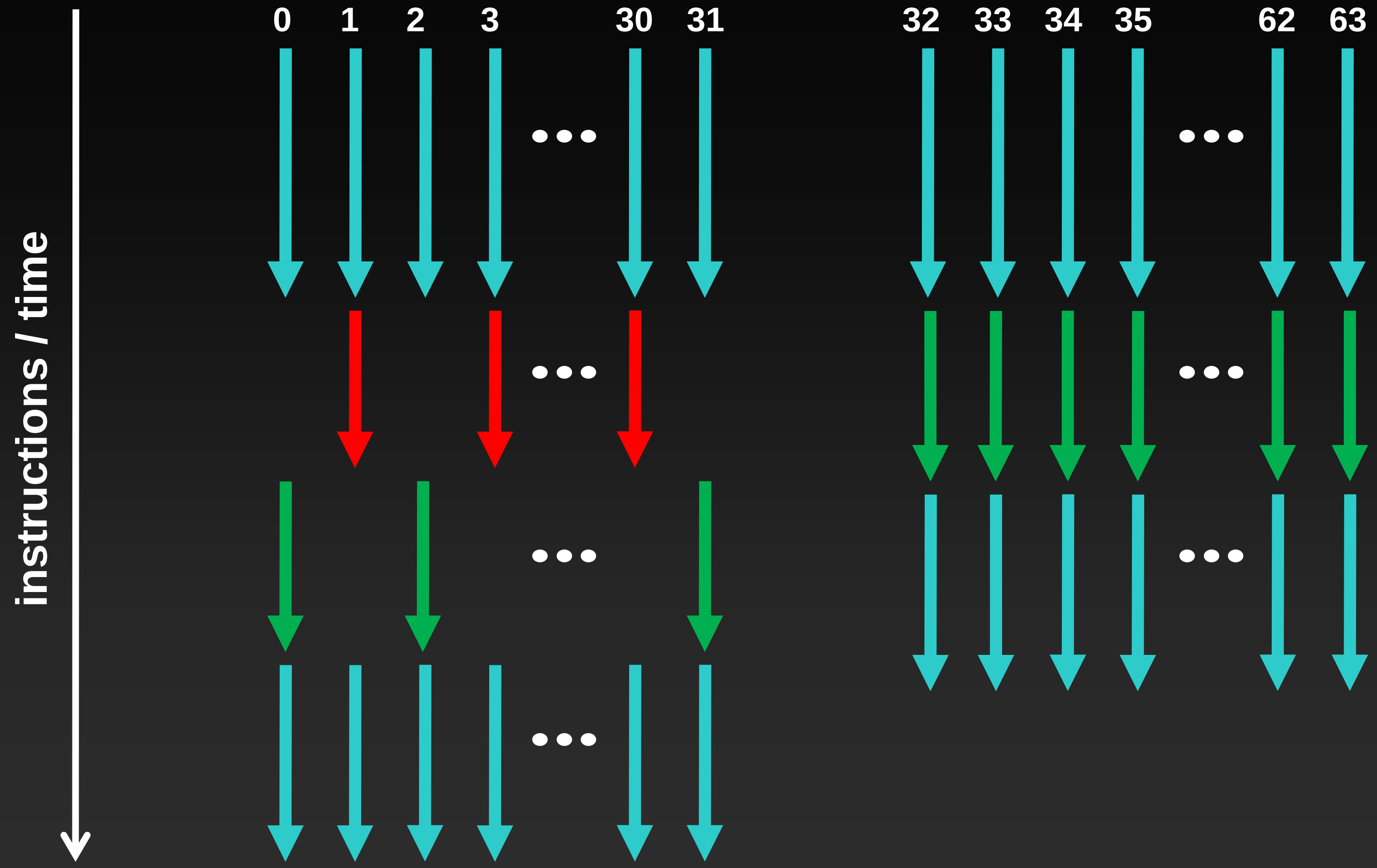


```
if ( ... )
{
    // then-clause
}
else
{
    // else-clause
}
```

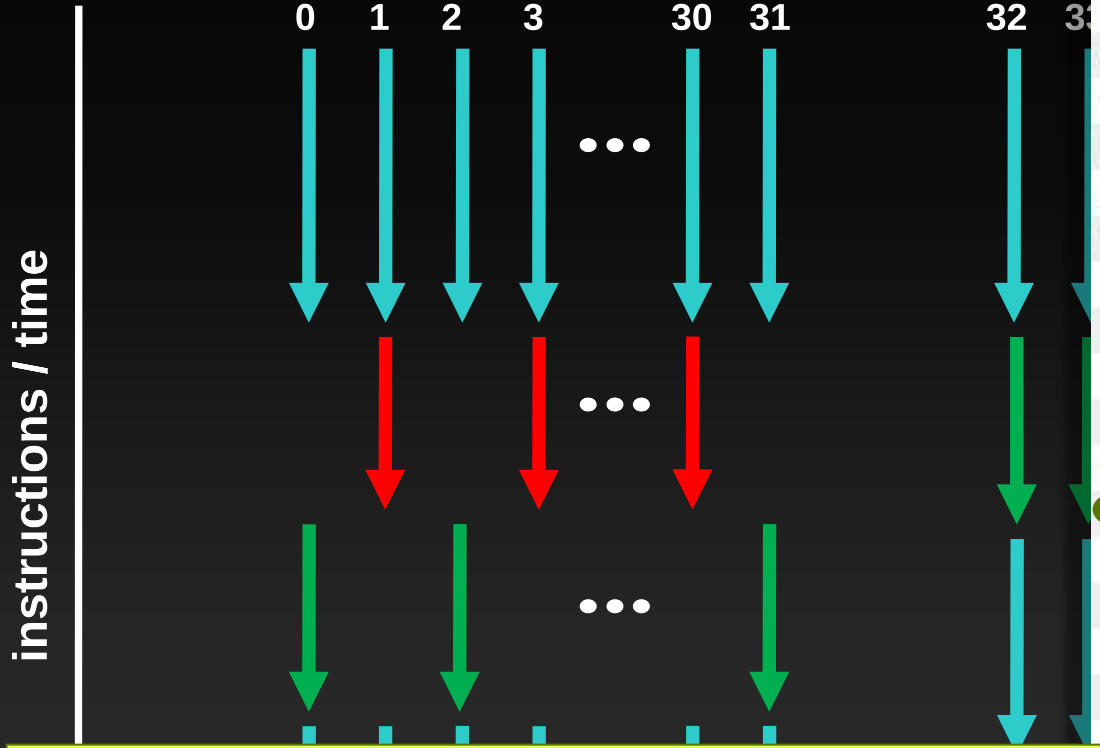
Execution within warps is coherent

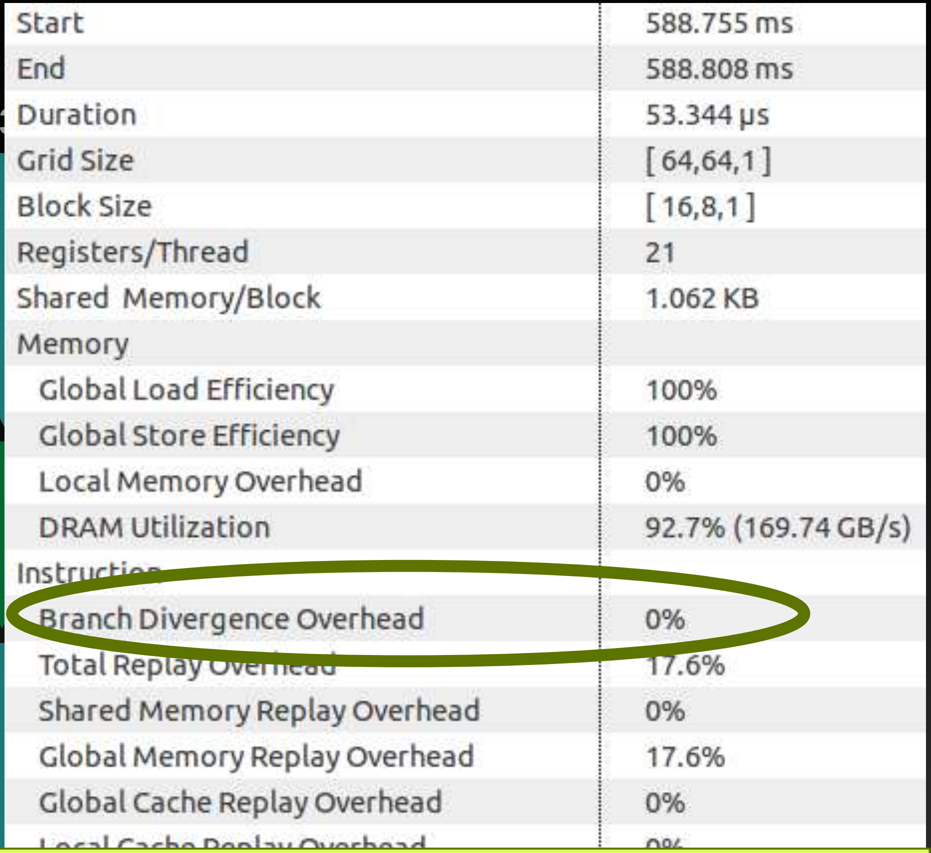


Execution diverges within a warp



Execution diverges within a warp





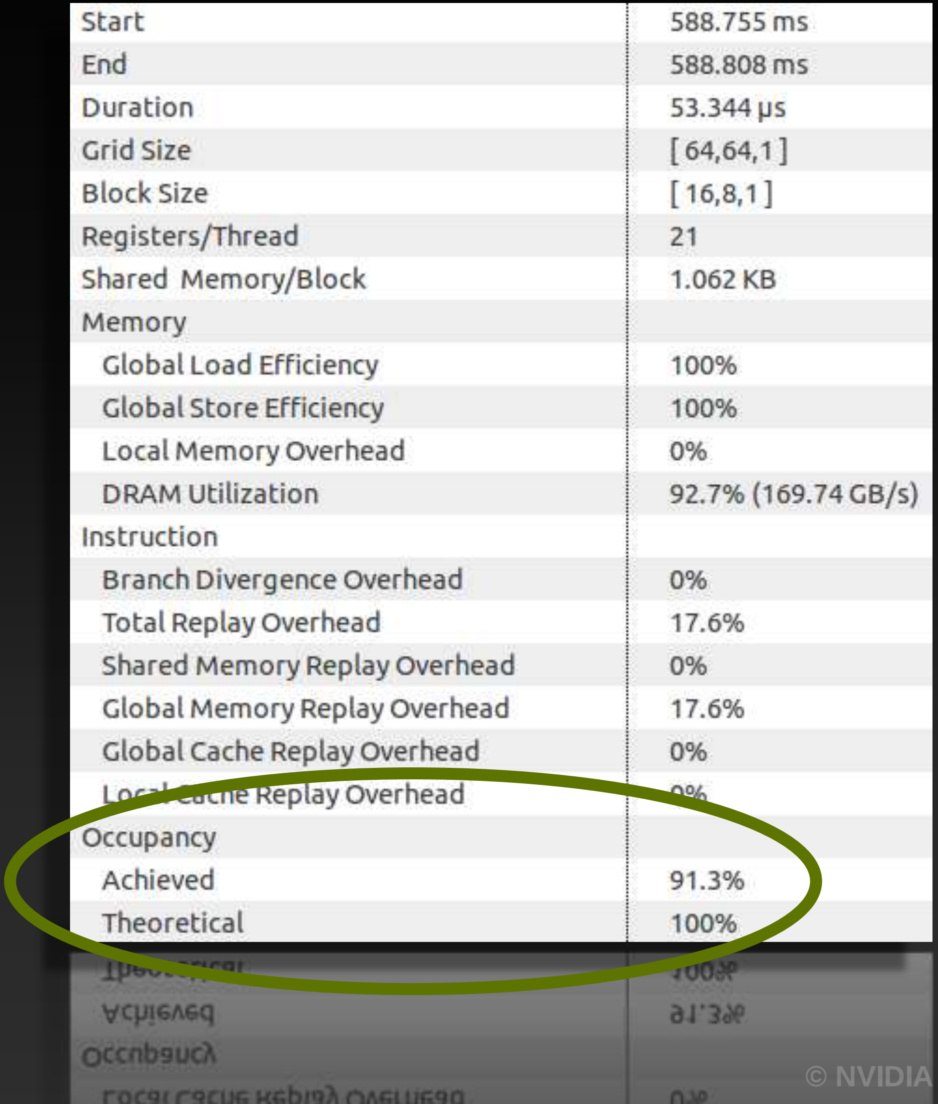
Start	588.755 ms
End	588.808 ms
Duration	53.344 μs
Grid Size	[64,64,1]
Block Size	[16,8,1]
Registers/Thread	21
Shared Memory/Block	1.062 KB
Memory	
Global Load Efficiency	100%
Global Store Efficiency	100%
Local Memory Overhead	0%
DRAM Utilization	92.7% (169.74 GB/s)
Instruction	
Branch Divergence Overhead	0%
Total Replay Overhead	17.6%
Shared Memory Replay Overhead	0%
Global Memory Replay Overhead	17.6%
Global Cache Replay Overhead	0%
Local Cache Replay Overhead	0%
Achieved	91.3%

Solution: Group threads with similar control flow
Factorize to minimize lines of divergent code

Occupancy

- Need independent threads per SM to hide latencies:
  - Memory access latencies
  - Instruction latencies
- Hardware resources determine number of threads that fit per SM

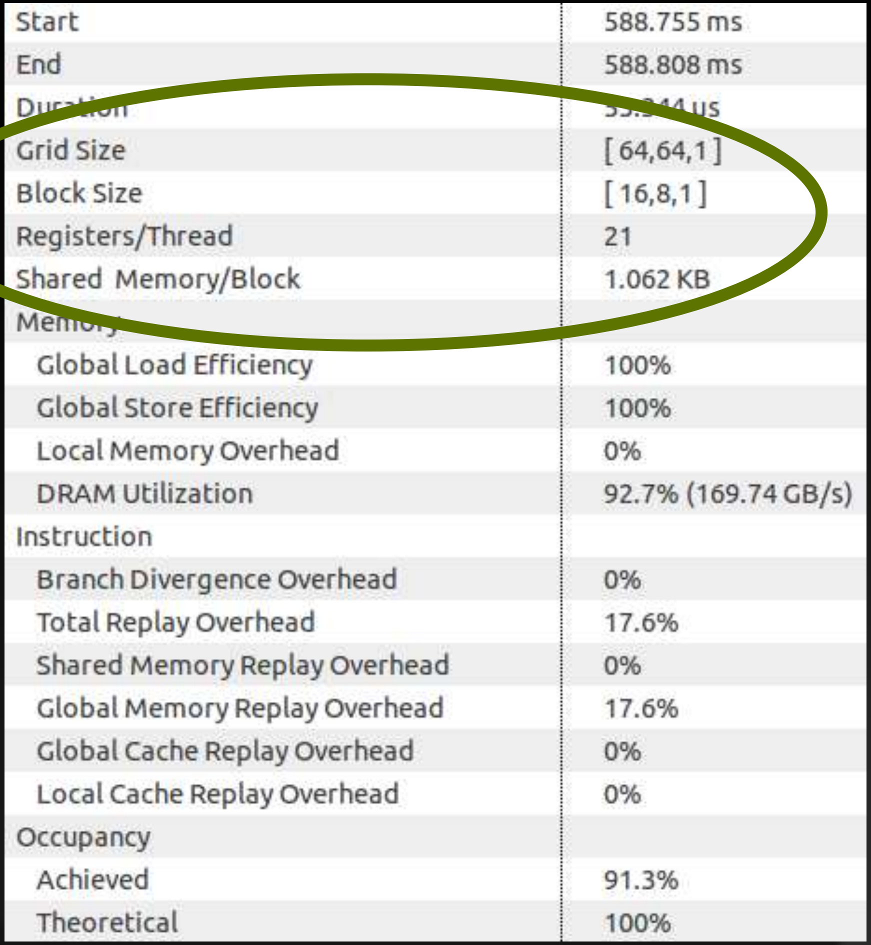
Occupancy = N_{actual} / N_{max}



Start	588.755 ms
End	588.808 ms
Duration	53.344 µs
Grid Size	[64,64,1]
Block Size	[16,8,1]
Registers/Thread	21
Shared Memory/Block	1.062 KB
Memory	
Global Load Efficiency	100%
Global Store Efficiency	100%
Local Memory Overhead	0%
DRAM Utilization	92.7% (169.74 GB/s)
Instruction	
Branch Divergence Overhead	0%
Total Replay Overhead	17.6%
Shared Memory Replay Overhead	0%
Global Memory Replay Overhead	17.6%
Global Cache Replay Overhead	0%
Local Cache Replay Overhead	0%
Occupancy	
Achieved	91.3%
Theoretical	100%

Occupancy

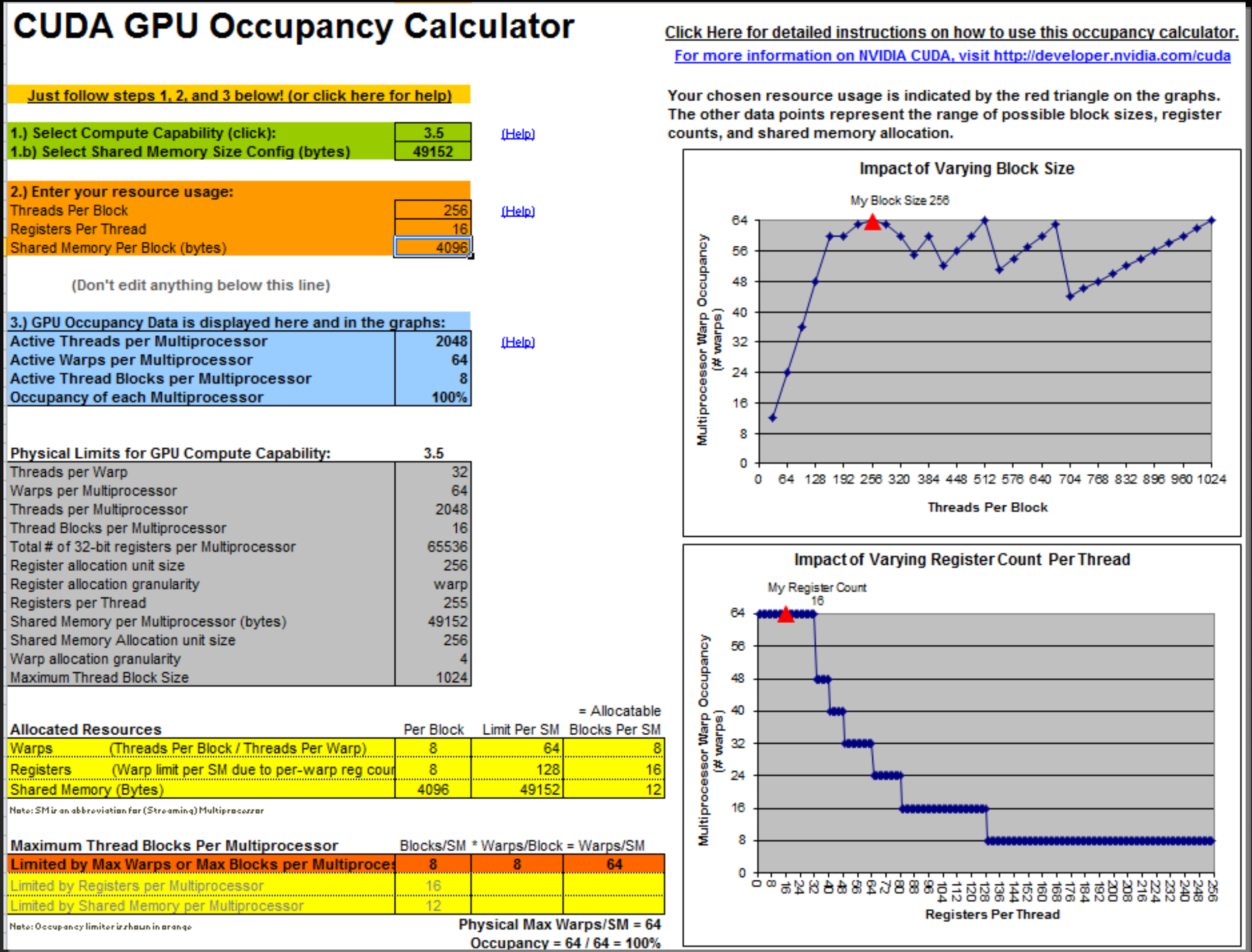
- Limiting resources:
  - Number of threads
  - Number of registers per thread
  - Number of blocks
  - Amount of shared memory per block
- Don’t need for 100% occupancy for maximum performance



Start	588.755 ms
End	588.808 ms
Duration	53.344 us
Grid Size	[64,64,1]
Block Size	[16,8,1]
Registers/Thread	21
Shared Memory/Block	1.062 KB
Memory	
Global Load Efficiency	100%
Global Store Efficiency	100%
Local Memory Overhead	0%
DRAM Utilization	92.7% (169.74 GB/s)
Instruction	
Branch Divergence Overhead	0%
Total Replay Overhead	17.6%
Shared Memory Replay Overhead	0%
Global Memory Replay Overhead	17.6%
Global Cache Replay Overhead	0%
Local Cache Replay Overhead	0%
Occupancy	
Achieved	91.3%
Theoretical	100%

CUDA Occupancy Calculator

- **Analyze effect of resource consumption on occupancy**



CUDA GPU Occupancy Calculator

Click Here for detailed instructions on how to use this occupancy calculator.
For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	3.5	(Help)
1.b) Select Shared Memory Size Config (bytes)	49152	

2.) Enter your resource usage:		
Threads Per Block	256	(Help)
Registers Per Thread	16	
Shared Memory Per Block (bytes)	4096	

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	2048	(Help)
Active Warps per Multiprocessor	64	
Active Thread Blocks per Multiprocessor	8	
Occupancy of each Multiprocessor	100%	

Physical Limits for GPU Compute Capability:	3.5
Threads per Warp	32
Warps per Multiprocessor	64
Threads per Multiprocessor	2048
Thread Blocks per Multiprocessor	16
Total # of 32-bit registers per Multiprocessor	65536
Register allocation unit size	256
Register allocation granularity	warp
Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	49152
Shared Memory Allocation unit size	256
Warp allocation granularity	4
Maximum Thread Block Size	1024

				= Allocatable
Allocated Resources		Per Block	Limit Per SM	Blocks Per SM
Warps	(Threads Per Block / Threads Per Warp)	8	64	8
Registers	(Warp limit per SM due to per-warp reg cour	8	128	16
Shared Memory (Bytes)		4096	49152	12

Note: SM is an abbreviation for (Streaming) Multiprocessor

Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiproces	8	8	64
Limited by Registers per Multiprocessor	16		
Limited by Shared Memory per Multiprocessor	12		

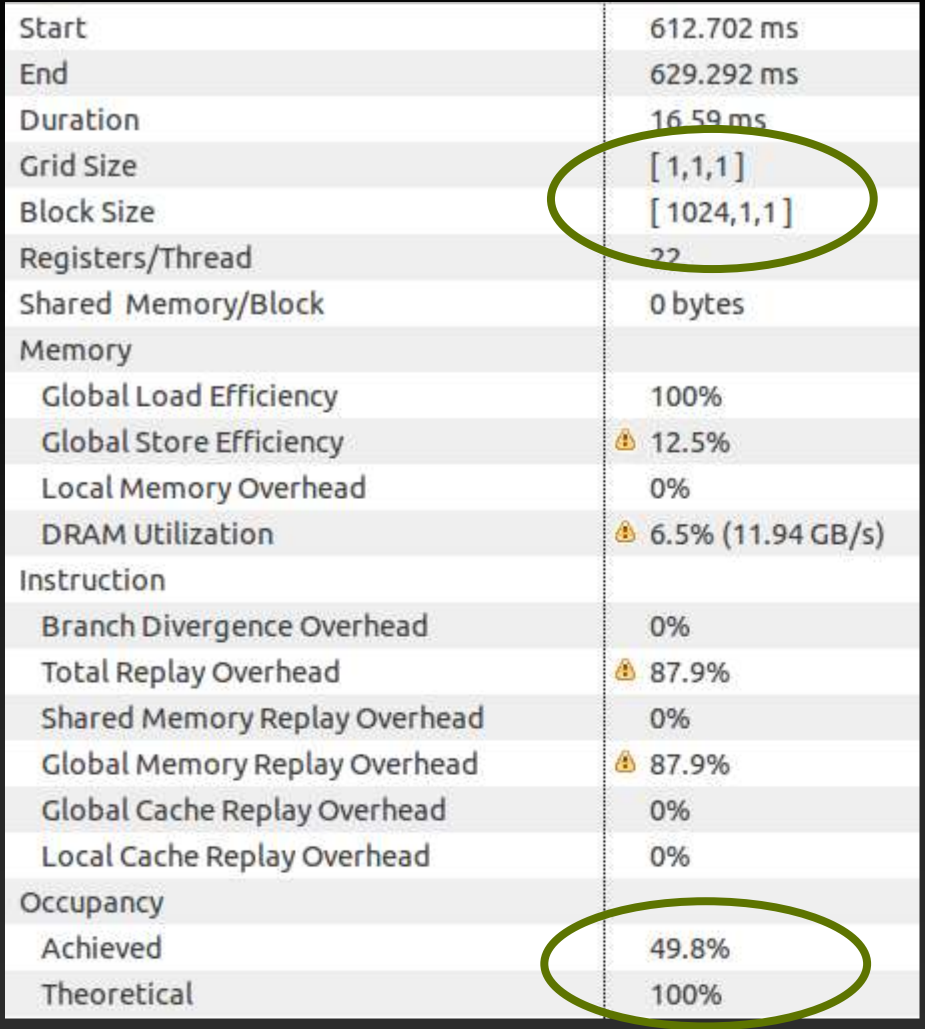
Note: Occupancy limiter is shown in orange

Physical Max Warps/SM = 64
Occupancy = 64 / 64 = 100%

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

Occupancy Example

- **Occupancy here is limited by grid size and number of threads per block**



Start	612.702 ms
End	629.292 ms
Duration	16.59 ms
Grid Size	[1,1,1]
Block Size	[1024,1,1]
Registers/Thread	22
Shared Memory/Block	0 bytes
Memory	
Global Load Efficiency	100%
Global Store Efficiency	⚠ 12.5%
Local Memory Overhead	0%
DRAM Utilization	⚠ 6.5% (11.94 GB/s)
Instruction	
Branch Divergence Overhead	0%
Total Replay Overhead	⚠ 87.9%
Shared Memory Replay Overhead	0%
Global Memory Replay Overhead	⚠ 87.9%
Global Cache Replay Overhead	0%
Local Cache Replay Overhead	0%
Occupancy	
Achieved	49.8%
Theoretical	100%

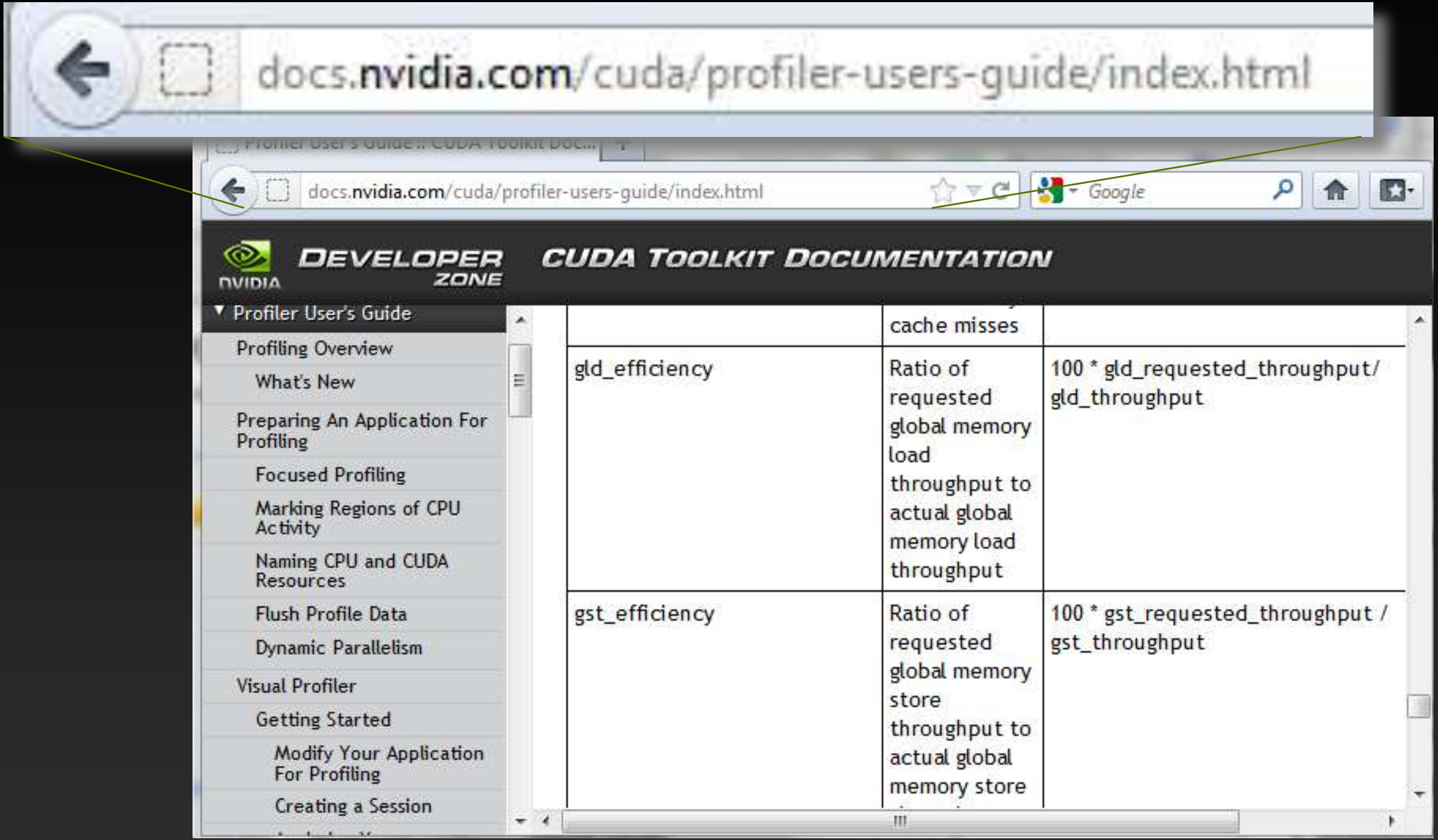
Alternative profiling: nvprof

```
%nvprof  --print-gpu-trace ./transpose

Profiling result:
   Start  Duration Grid Size  Block Size  Regs*  Size     Throughput   Name
577.11ms  874.57us         -          -      -   4.19MB  4.80GB/s     [CUDA memcpy HtoD]
598.45ms    1.67ms    (1 1 1)  (1024 1 1)   22     -          -        transposeNaive(float*,
600.12ms    1.67ms    (1 1 1)  (1024 1 1)   22     -          -        transposeNaive(float*,
601.79ms    1.67ms    (1 1 1)  (1024 1 1)   22     -          -        transposeNaive(float*,
```

```
nvprof --print-gpu-trace  --aggregate-mode-off --events sm_cta_launched ./transpose

Profiling result:
Device            Event Name,      Kernel,                       Values
  0               sm_cta_launched, transposeNaive(float*, ..), 76 73 72 72 73 74 75 73 73 72 73 73 72 73
```

- **Command-Line Profiler**
- **Access to hardware counters**
  - **List of supported counters: --query-events**

Alternative profiling: nvprof

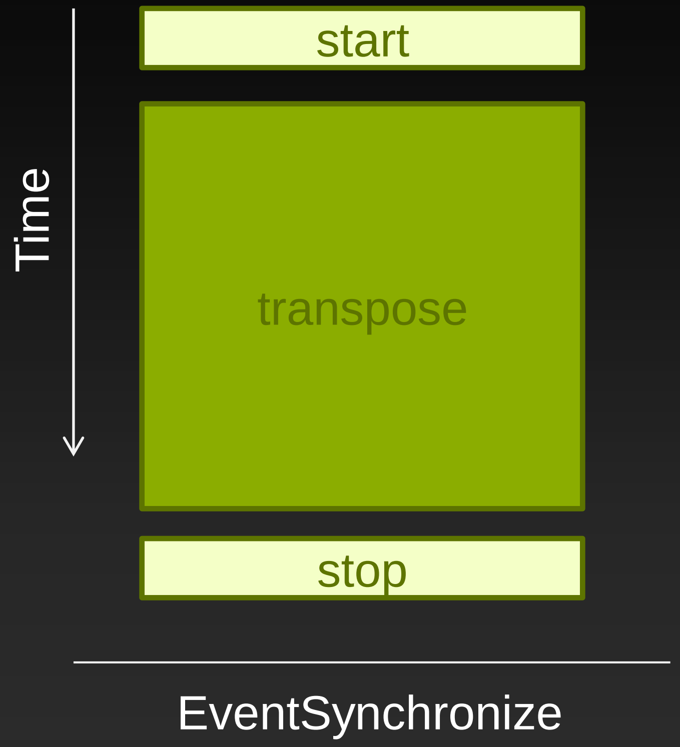


docs.nvidia.com/cuda/profiler-users-guide/index.html

	cache misses	
gld_efficiency	Ratio of requested global memory load throughput to actual global memory load throughput	100 * gld_requested_throughput/ gld_throughput
gst_efficiency	Ratio of requested global memory store throughput to actual global memory store	100 * gst_requested_throughput / gst_throughput

Alternative profiling: instrumentation

```
cudaEventRecord(start, 0);

transpose<<<grid, threads>>>(..);

cudaEventRecord(stop,0);

cudaEventSynchronize(stop);

cudaEventElapsedTime(&time, start, stop);
```



KEPLER HIGHLIGHTS

KEPLER HIGHLIGHTS

Leveraging fine-grained parallelism

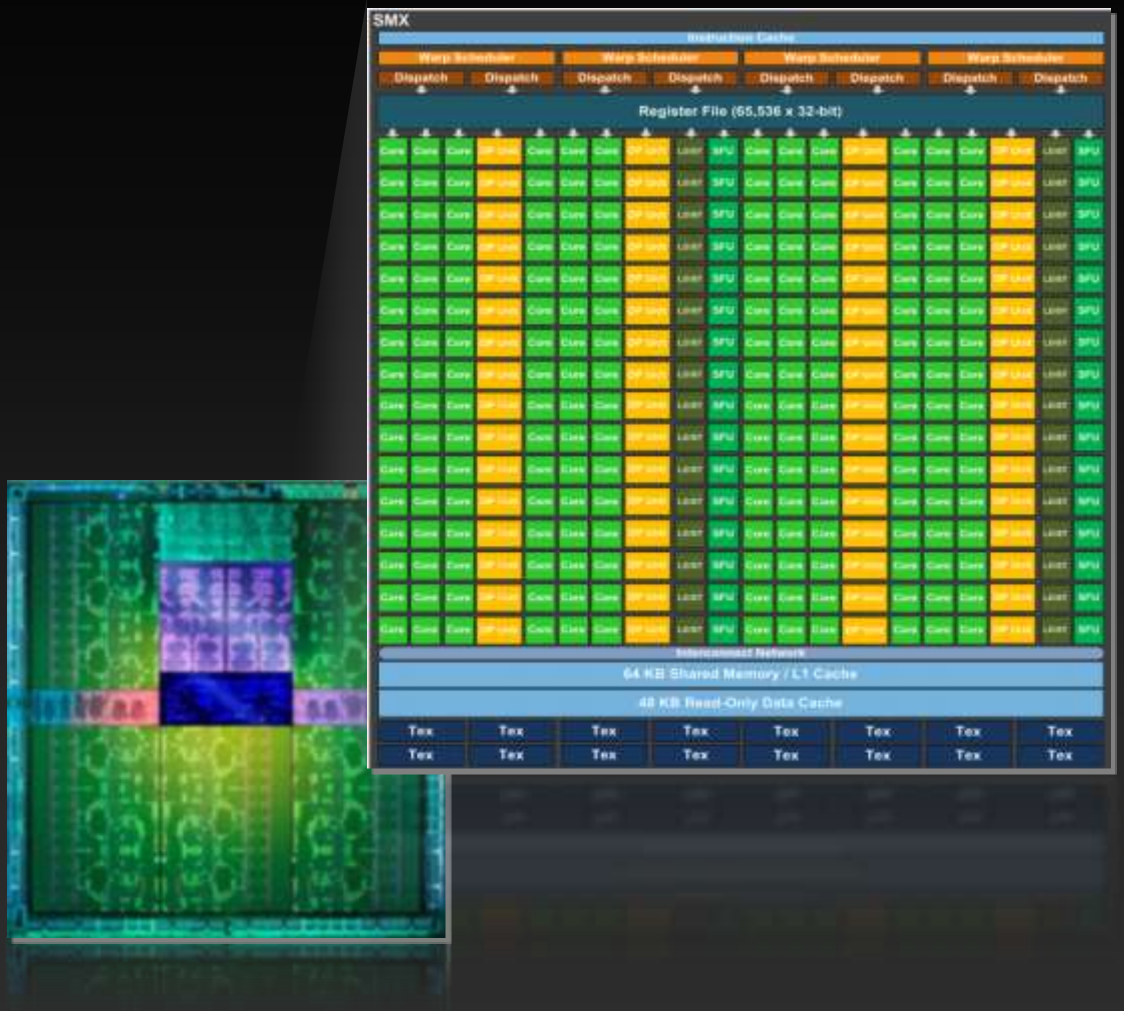
KEPLER HIGHLIGHTS

Leveraging fine-grained parallelism:
SMX

Kepler Streaming Multiprocessor (SMX)

Per SMX:

- 192 SP CUDA Cores
- 64 DP CUDA Cores
- 4 warp schedulers
  - Up to 2048 concurrent threads
  - One or two instructions issued per scheduler per clock from a single warp


SMX
Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit)
Core
DP Unit
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
48 KB Read-Only Data Cache
Tex

Exposing Sufficient Parallelism

- What SMX ultimately needs:
  - Sufficient number of independent instructions
  - Kepler GK110 is “wider” than Fermi or GK104; needs more parallelism
- Two ways to increase parallelism:
  - More independent instructions (ILP) within a thread (warp)
  - More concurrent threads (warps)

ILP vs. TLP

- SMX can leverage available Instruction-Level Parallelism more or less interchangeably with Thread-Level Parallelism
  - Much better at this than Fermi
- Sometimes easier to increase ILP than to increase TLP
  - E.g. # of threads may be limited by algorithm or by HW resource limits
  - But if each thread has some degree of independent operations to do, Kepler SMX can leverage that (e.g. a small loop that is unrolled)
- In fact, some degree of ILP is actually *required* to approach theoretical max Instructions Per Clock (IPC)

KEPLER HIGHLIGHTS

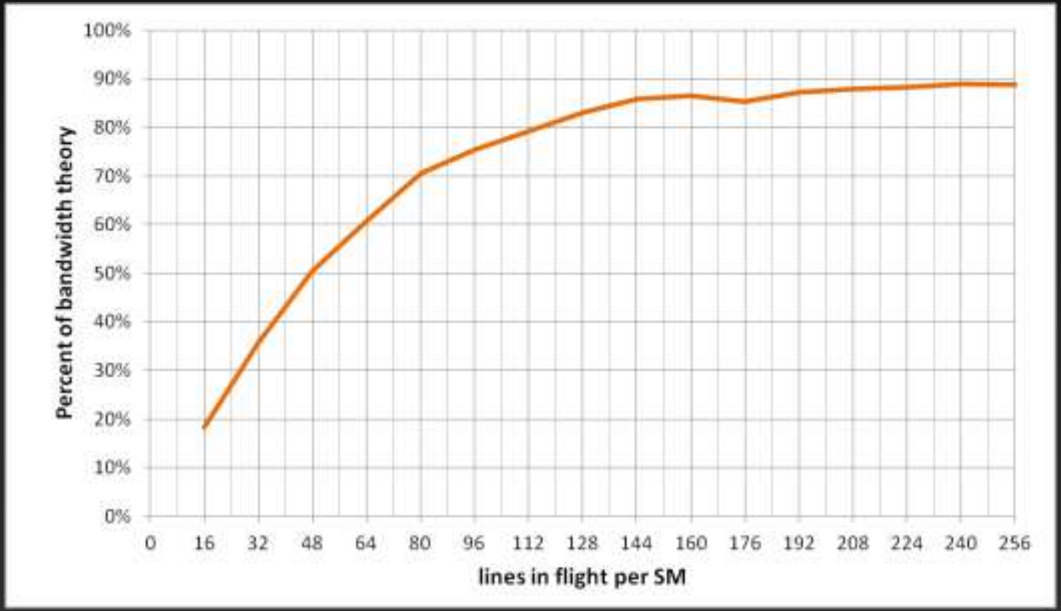
Leveraging fine-grained parallelism:
Memory bandwidth

Exposing Sufficient Parallelism

- What memory system hardware ultimately needs:
  - Sufficient requests in flight to saturate bandwidth
- Two ways to increase parallelism:
  - More independent accesses within a thread (warp)
  - More concurrent threads (warps)

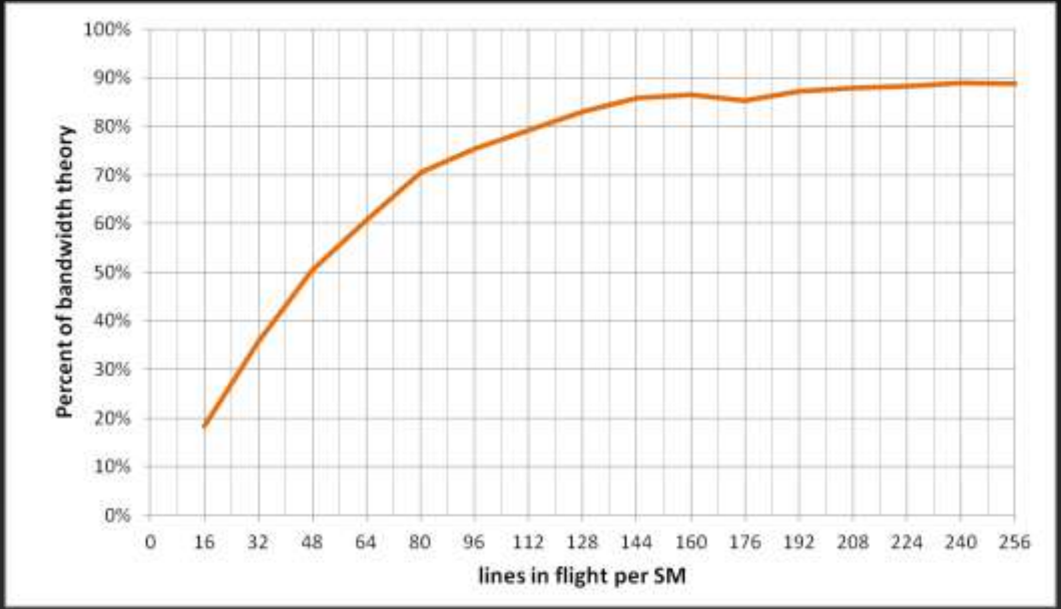
Memory-Level Parallelism = Bandwidth

- Achieved Kepler memory throughput
  - Shown as a function of number of concurrent requests per SM with 128-byte lines



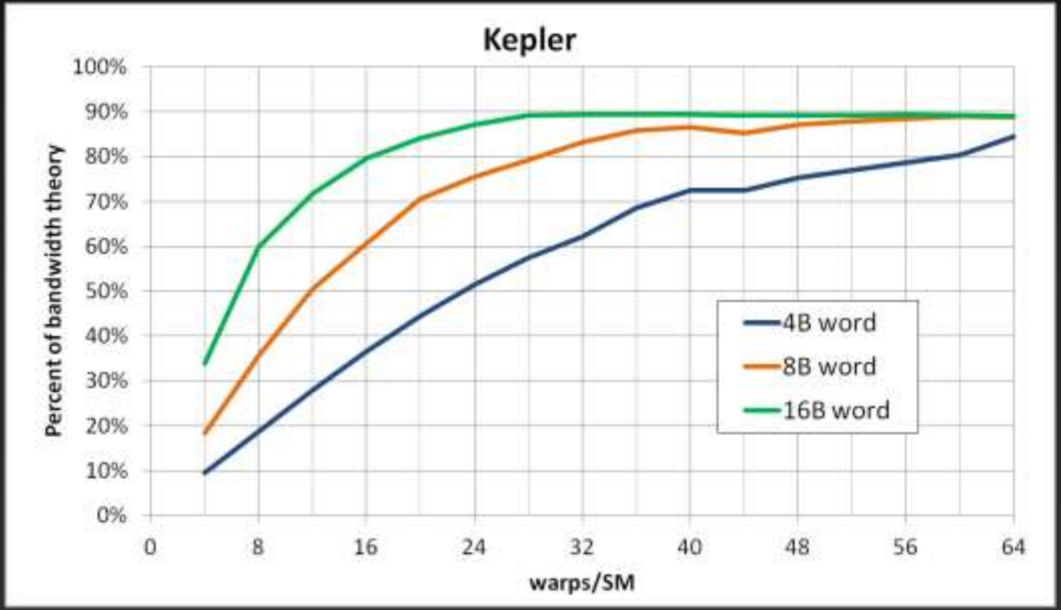
Memory-Level Parallelism = Bandwidth

- In order to saturate memory bandwidth, SM must issue enough independent memory requests concurrently



Elements per Thread and Performance

- Experiment: vary size of accesses by threads of a warp, check performance
  - Memcpy kernel: each warp has 2 concurrent requests (one write and the read following it)



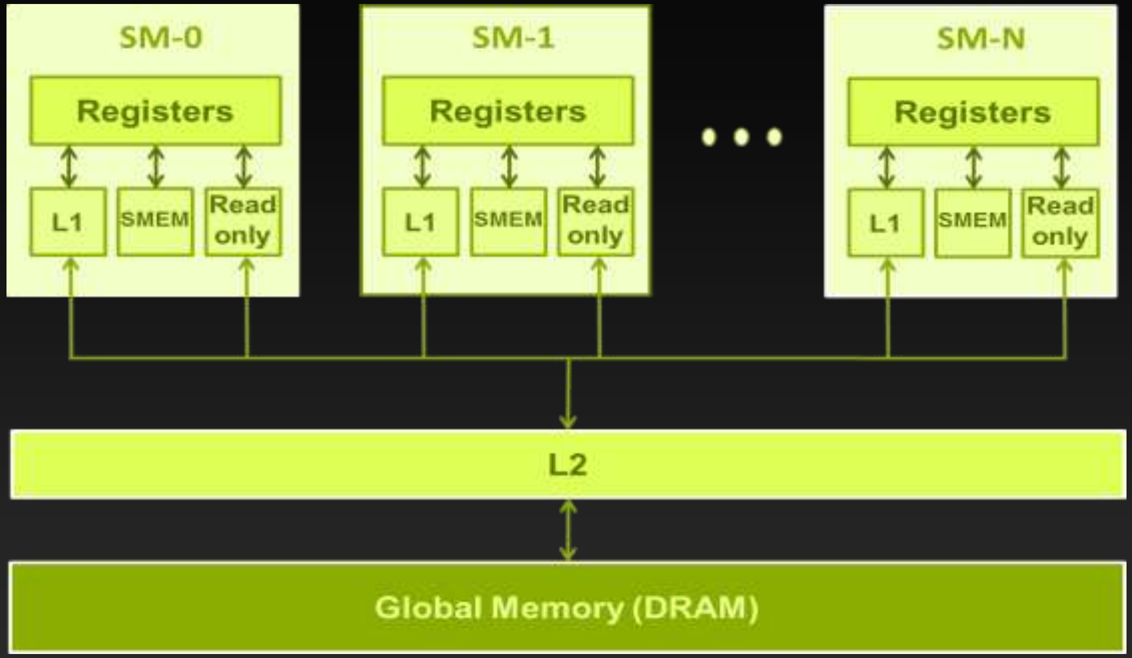
Accesses by a warp:

4B words:	1 line
8B words:	2 lines
16B words:	4 lines

To achieve same throughput at lower occupancy or with smaller words, need more independent requests per warp

A note about caches

- L1 and L2 caches
  - Ignore in software design
  - Thousands of concurrent threads – cache blocking difficult at best
- Read-only Data Cache
  - Shared with texture pipeline
  - Useful for uncoalesced reads
  - Handled by compiler when `const __restrict__` is used



KEPLER HIGHLIGHTS

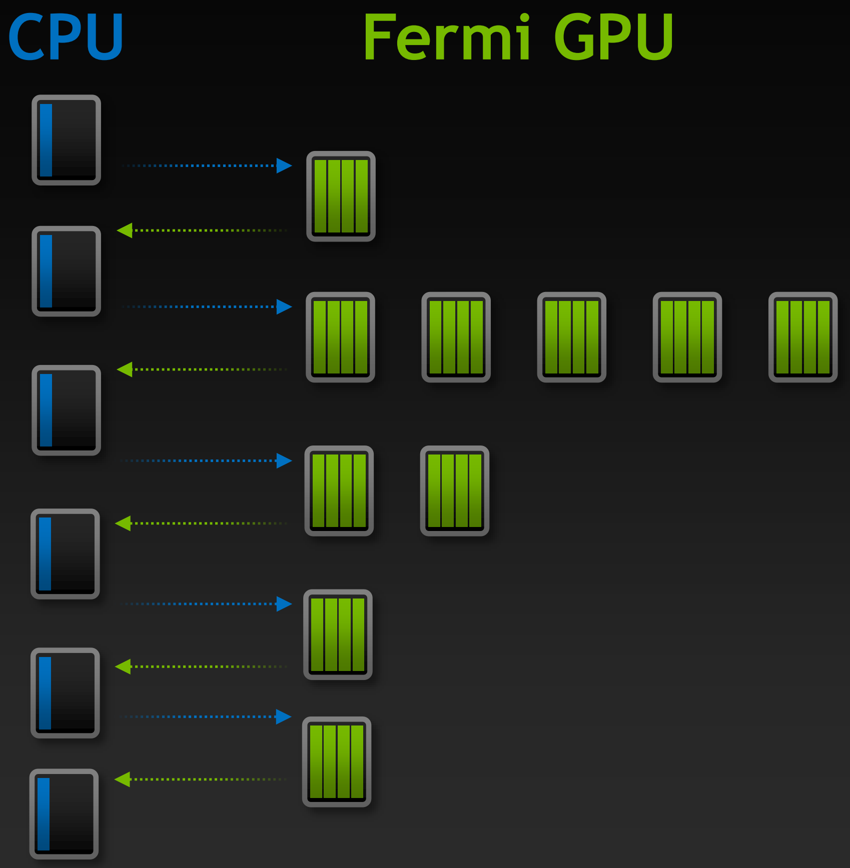
Leveraging coarse-grained parallelism

KEPLER HIGHLIGHTS

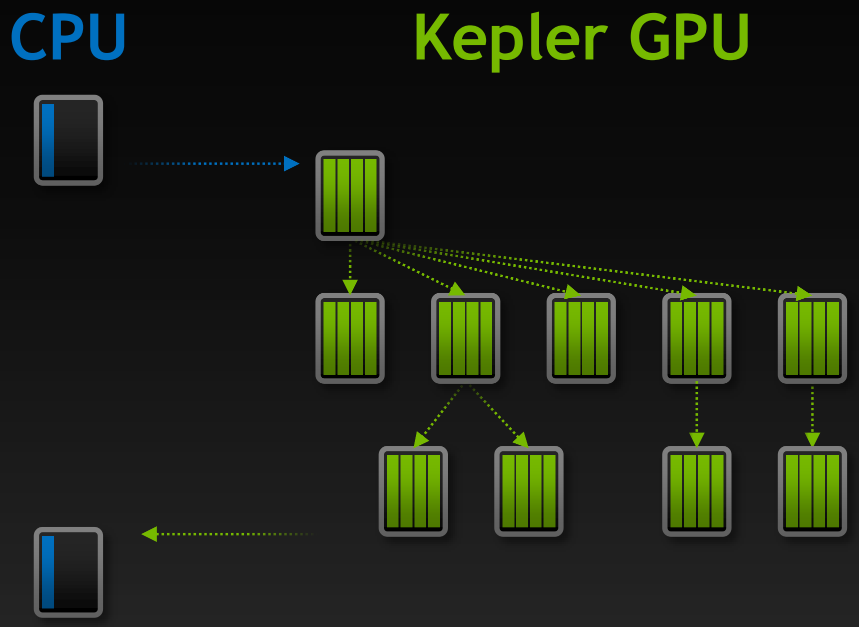
Leveraging coarse-grained parallelism:

Dynamic Parallelism

Dynamic Parallelism



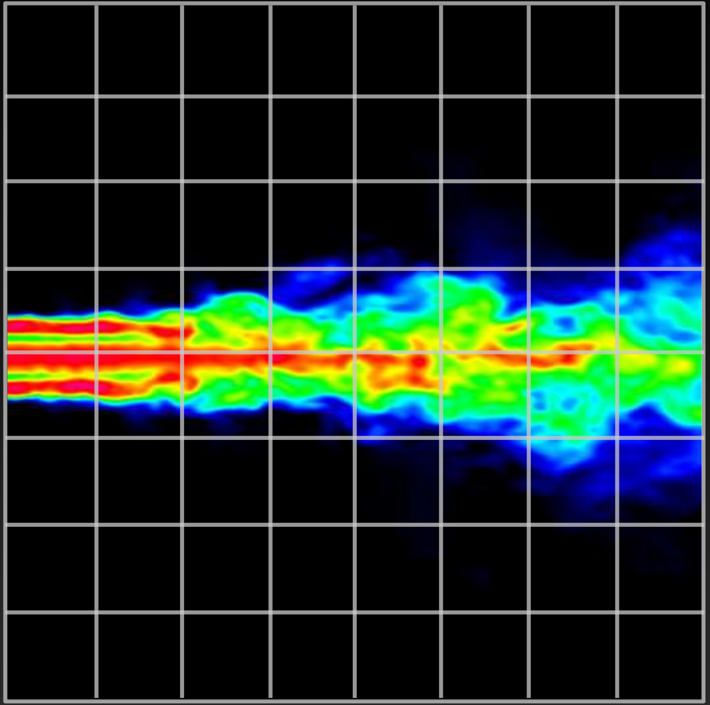
GPU as Co-Processor



Autonomous, Dynamic Parallelism

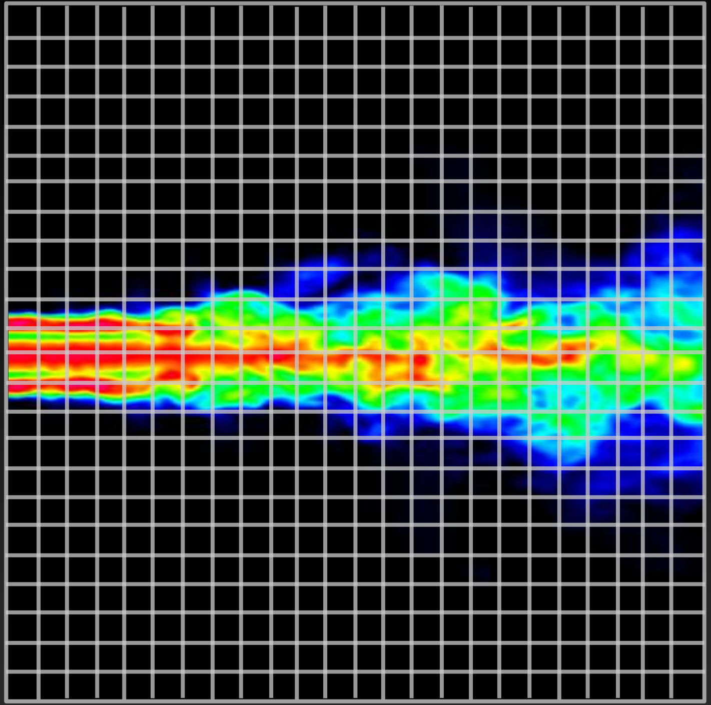
Dynamic Parallelism

Coarse grid



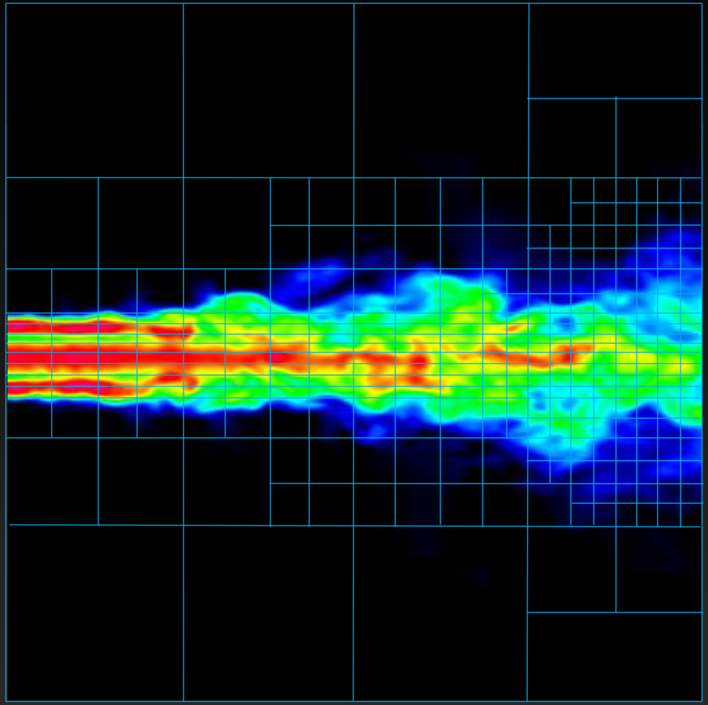
Higher Performance
Lower Accuracy

Fine grid



Lower Performance
Higher Accuracy

Dynamic grid



Target performance where accuracy is required

Supported on GK110 GPUs

Dynamic Parallelism

- Kernel launches grids
- Syntax is identical to host
- CUDA Runtime functions in `cudadevrt` library

```
__global__ void childKernel()
{
    printf("Hello %d", threadIdx.x);
}

__global__ void parentKernel()
{
   childKernel<<<1,10>>>();
   cudaDeviceSynchronize();
   printf("World!\n");
}
```

```
int main(int argc, char *argv[])
{
  parentKernel<<<1,1>>>();
  cudaDeviceSynchronize();
  return 0;
}
```

Dynamic Parallelism :: nested parallelism

- Return traffic to the host after each algorithm step is not required to be a good case for Dynamic Parallelism
  - We often illustrate Dynamic Parallelism that way, but that's just one example

- Look for cases of general nested parallelism as well
  - E.g., apps that don't have enough parallelism exposed at any one place, even though in aggregate there is much more

Dynamic (Nested) Parallelism Example

```
void f(void)
{
    for (int i = 0 ; i < 12 ; i++)
        v[i].doSomething();
}

V::doSomething(void)
{
    for (int j = 0 ; j < 100 ; j++)
        x[j].innerSomething();
}

X::innerSomething(void)
{
    for (int k = 0 ; k < 29 ; k++)
        y[k].evaluate();
}
```

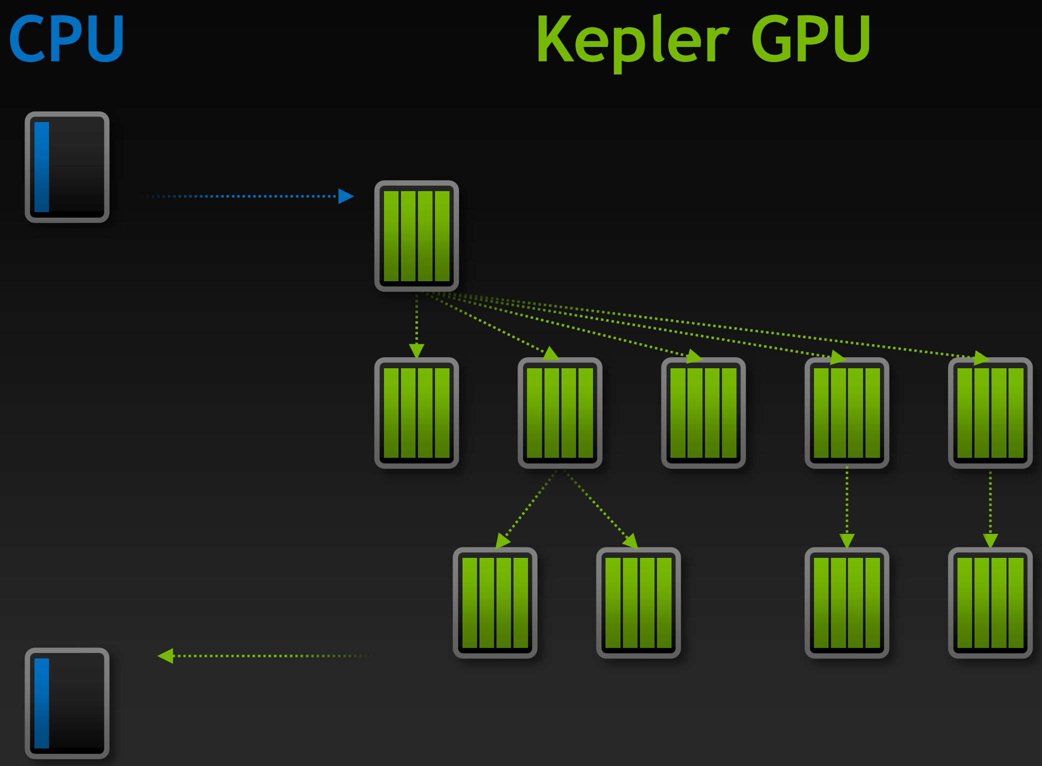
- `evaluate()` is called a total of 34800 times
- But parallelism is only exposed as 29 calls at a time
- Choices: flatten C++ hierarchy
  - Lose abstraction
  - What if functions are virtual?
- Dynamic Parallelism makes this much simpler

Dynamic (Nested) Parallelism Example

```
void f(void)
{
    for (int i = 0 ; i < 12 ; i++)
        v[i].doSomething();
}

V::doSomething(void)
{
    for (int j = 0 ; j < 100 ; j++)
        x[j].innerSomething();
}

X::innerSomething(void)
{
    for (int k = 0 ; k < 29 ; k++)
        y[k].evaluate();
}
```

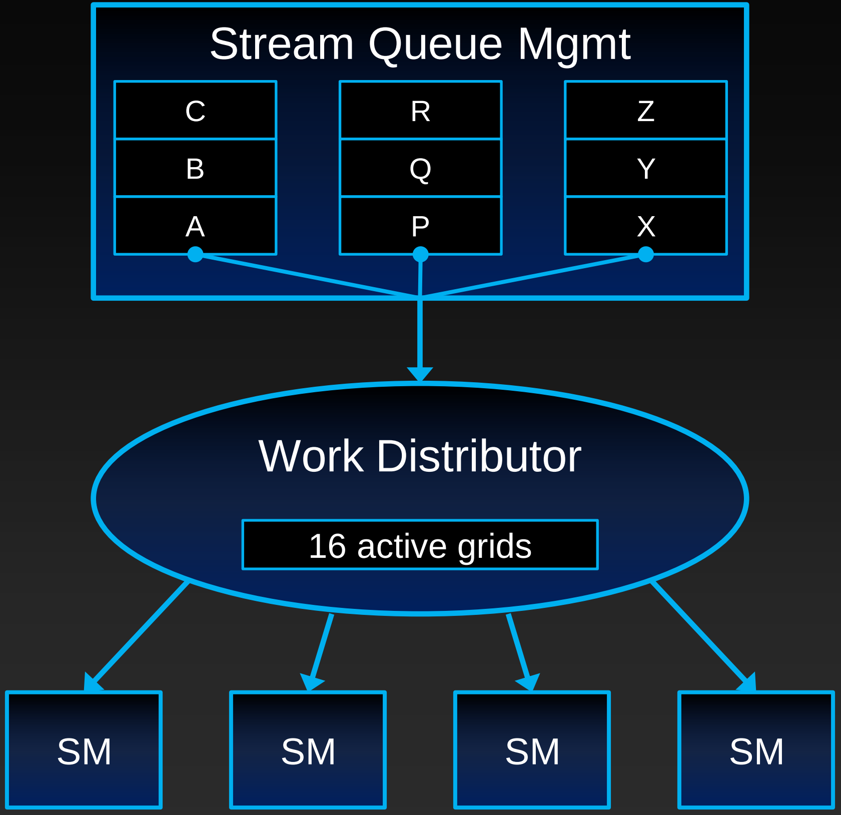
```
void f(void)
{
    V::doSomething_krnl<<<1,12>>>(v);
}

__global__ V::doSomething_krnl(V *v)
{
    X::innerSomething_krnl<<<1,100>>>
        (v[threadIdx.x].x);
}

__global__ X::innerSomething_krnl(X *x)
{
    Y::evaluate_krnl<<<1,29>>>
        (x[threadIdx.x].y);
}
```

Dynamic Parallelism

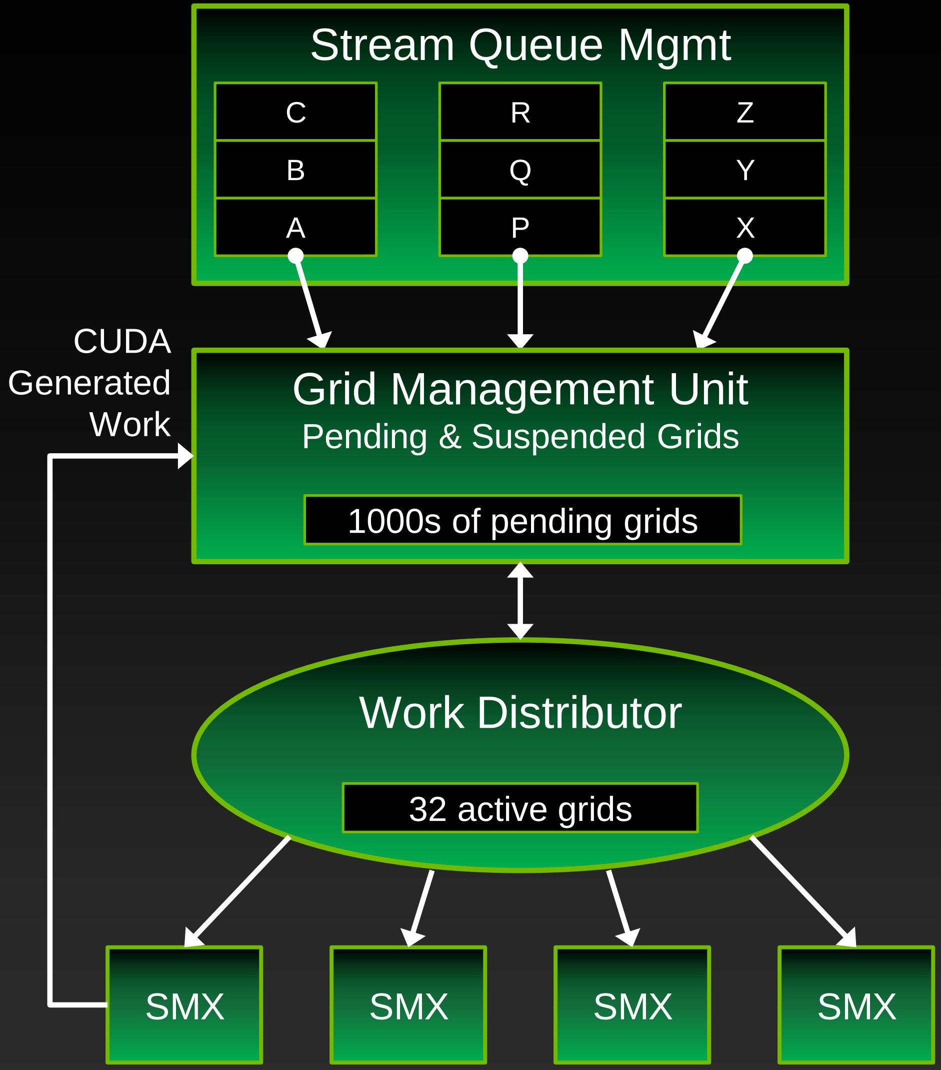


Autonomous, Dynamic Parallelism

Grid Management



Fermi



Kepler GK110

KEPLER HIGHLIGHTS

Leveraging coarse-grained parallelism:

Hyper-Q

Hyper-Q Enables Efficient Scheduling

- **Grid Management Unit selects most appropriate task from up to 32 hardware queues (CUDA streams)**
- **Improves scheduling of concurrently executed grids**
- **Particularly interesting for MPI applications when combined with CUDA Proxy, but *not limited to MPI applications***

CUDA Proxy

- Linux-only experimental feature in CUDA 5.0
  - Fully supported on Cray systems in CUDA 5.0
  - Full production support to be expanded in upcoming CUDA release
- No application modifications necessary
  - Launch proxy daemon with `nvidia-proxy-server-control –d`
  - CUDA driver automatically detects running daemon and routes GPU accesses through it
- Combines requests from several processes into one GPU context (shared virtual memory space, concurrent kernels possible, etc.)

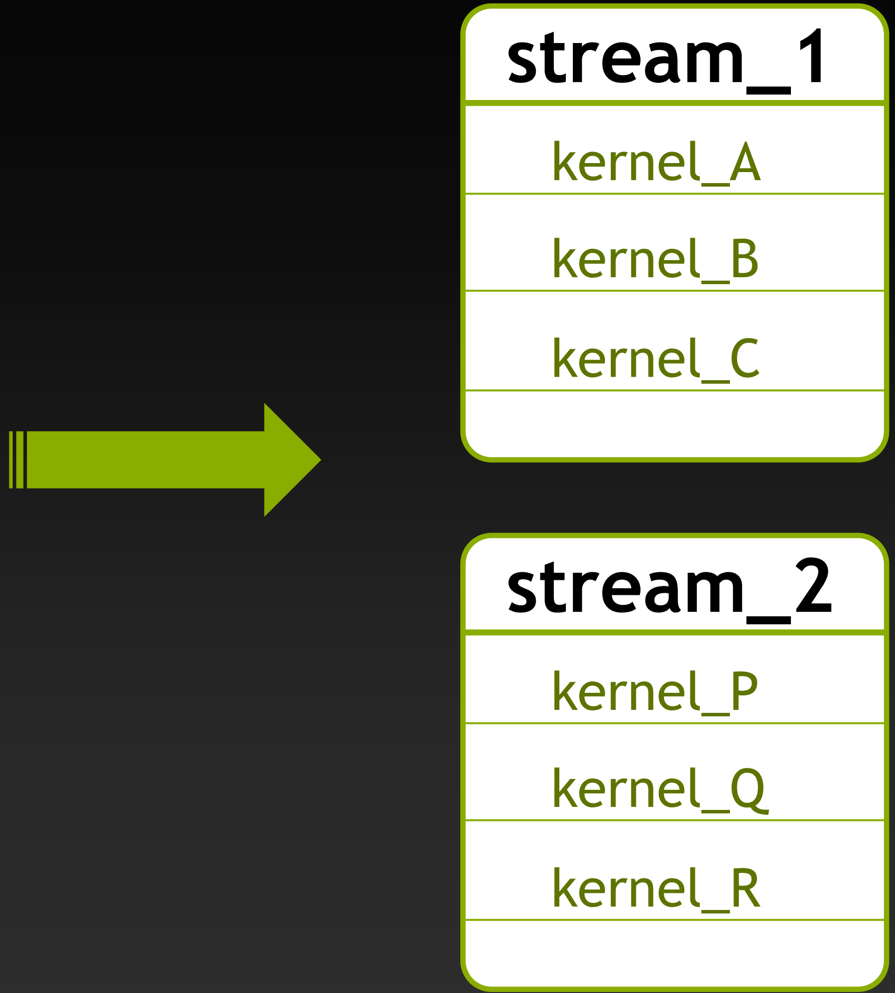
Hyper-Q for non-MPI apps

- One process: No proxy required!
  - Automatically utilized
  - One or many host threads no problem
  - Just need multiple CUDA streams
  - Removes false dependencies among CUDA streams that reduce effective concurrency on Fermi and GK104 GPUs

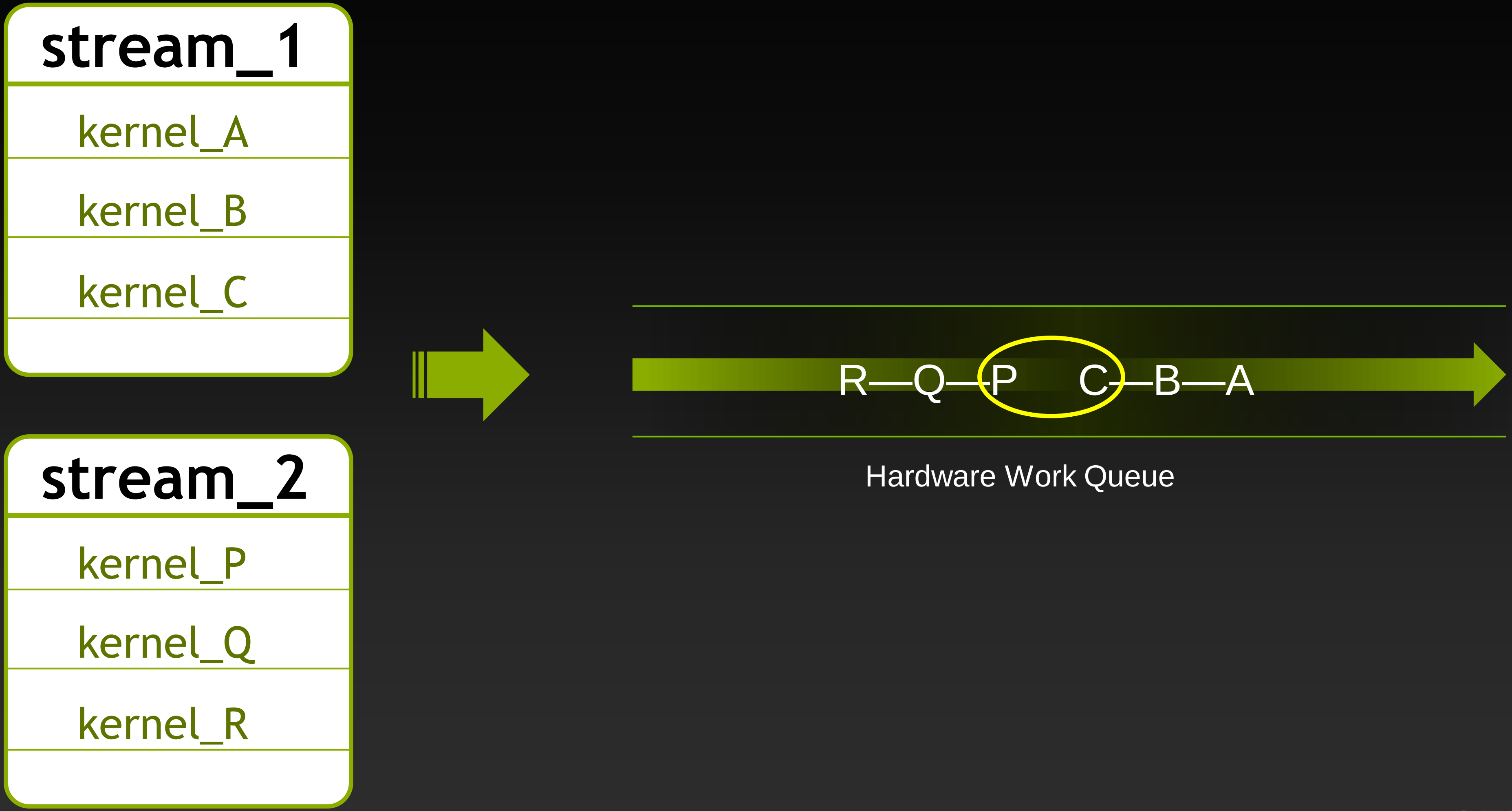
- Multi-process: Use CUDA Proxy, even if not MPI
  - Though currently experimental, proxy still interesting for task-level parallelism across processes from the same user
  - MPI is not required for proxy – it’s just a common case for HPC

Stream Dependencies Example

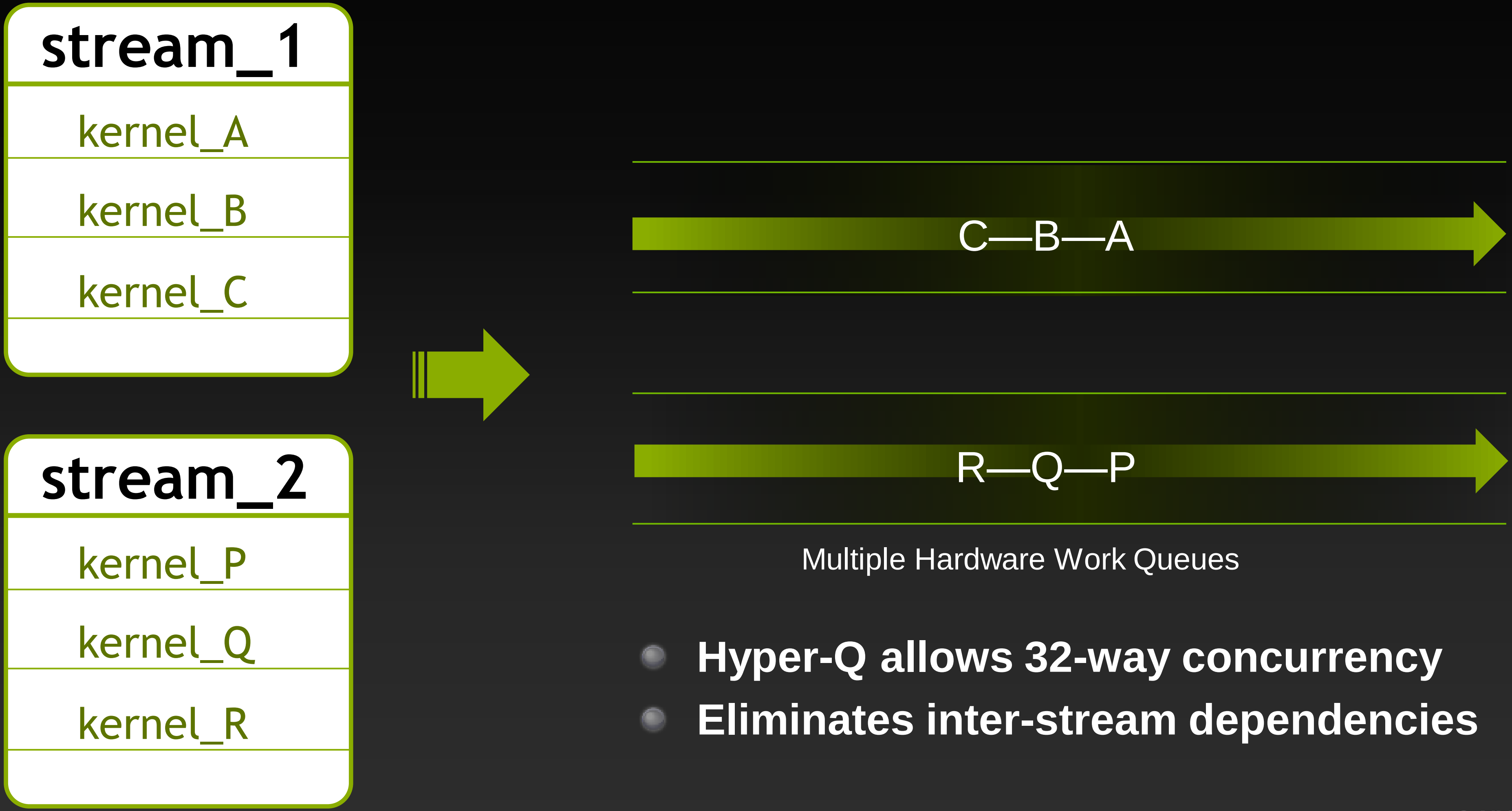
```
void foo(void)
{
    kernel_A<<<g,b,s, stream_1>>>();
    kernel_B<<<g,b,s, stream_1>>>();
    kernel_C<<<g,b,s, stream_1>>>();
}

void bar(void)
{
    kernel_P<<<g,b,s, stream_2>>>();
    kernel_Q<<<g,b,s, stream_2>>>();
    kernel_R<<<g,b,s, stream_2>>>();
}
```



Stream Dependencies without Hyper-Q

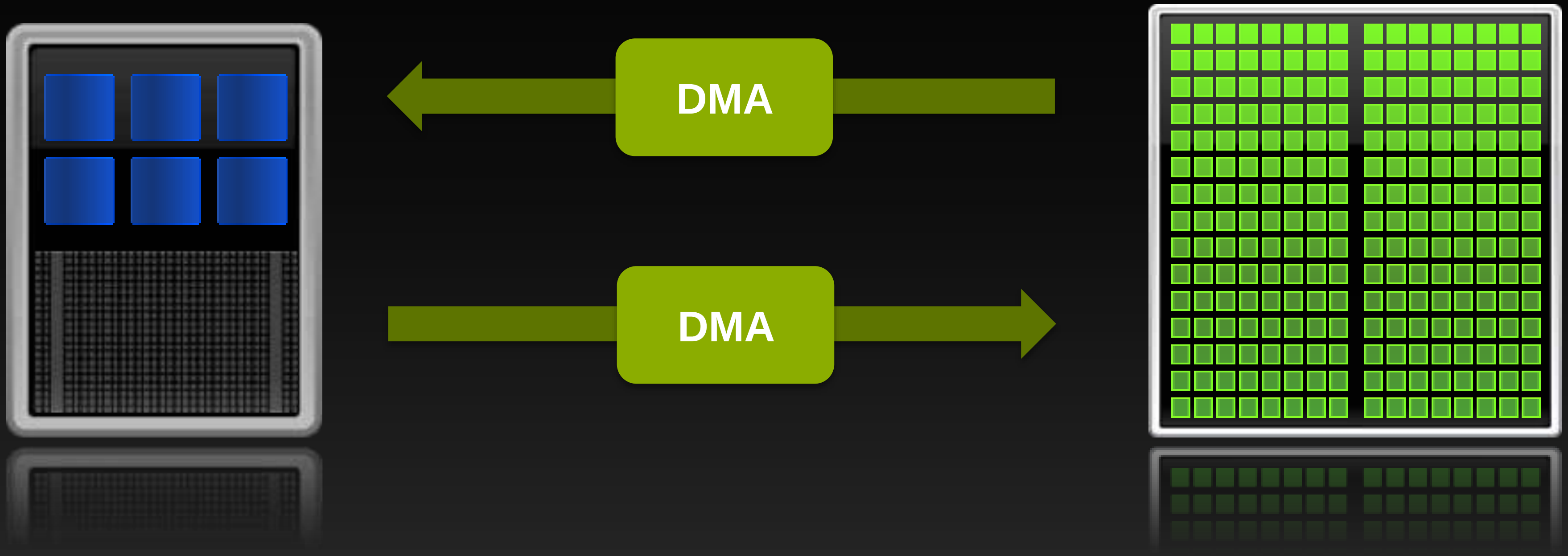


Stream Dependencies with Hyper-Q



- **Hyper-Q allows 32-way concurrency**
- **Eliminates inter-stream dependencies**

Hyper-Q Example: Building a Pipeline

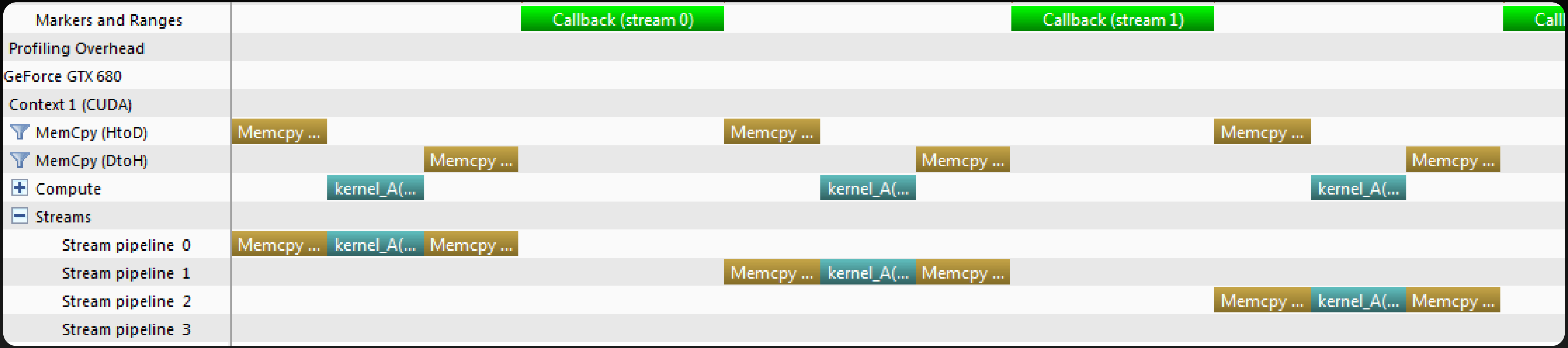


- Heterogeneous system: overlap work and data movement
- Kepler + CUDA 5: Hyper-Q and CPU Callbacks

Pipeline Code

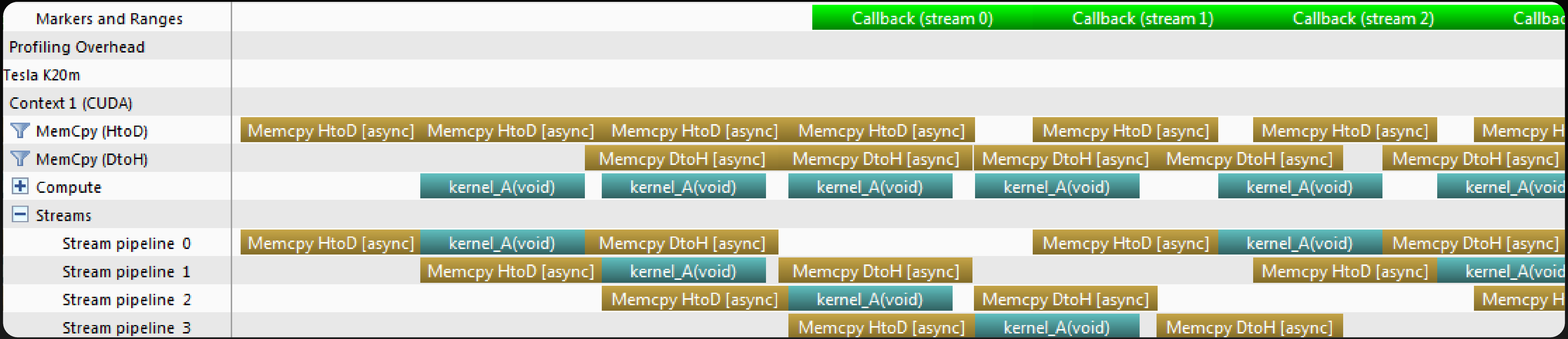
```
for (unsigned int i = 0 ; i < nIterations ; ++i)
{
    // Copy data from host to device
    chk(cudaMemcpyAsync(d_data, h_data, cpybytes, cudaMemcpyHostToDevice,
                        *r_streams.active()));

    // Launch device kernel A
    kernel_A<<<gdim, bdim, 0, *r_streams.active()>>>();

    // Copy data from device to host
    chk(cudaMemcpyAsync(h_data, d_data, cpybytes, cudaMemcpyDeviceToHost,
                        *r_streams.active()));

    // Launch host post-process
    chk(cudaStreamAddCallback(*r_streams.active(), cpu_callback,
                              r_streamids.active(), 0));

    // Rotate streams
    r_streams.rotate(); r_streamids.rotate();
}
```

Pipeline Without Hyper-Q



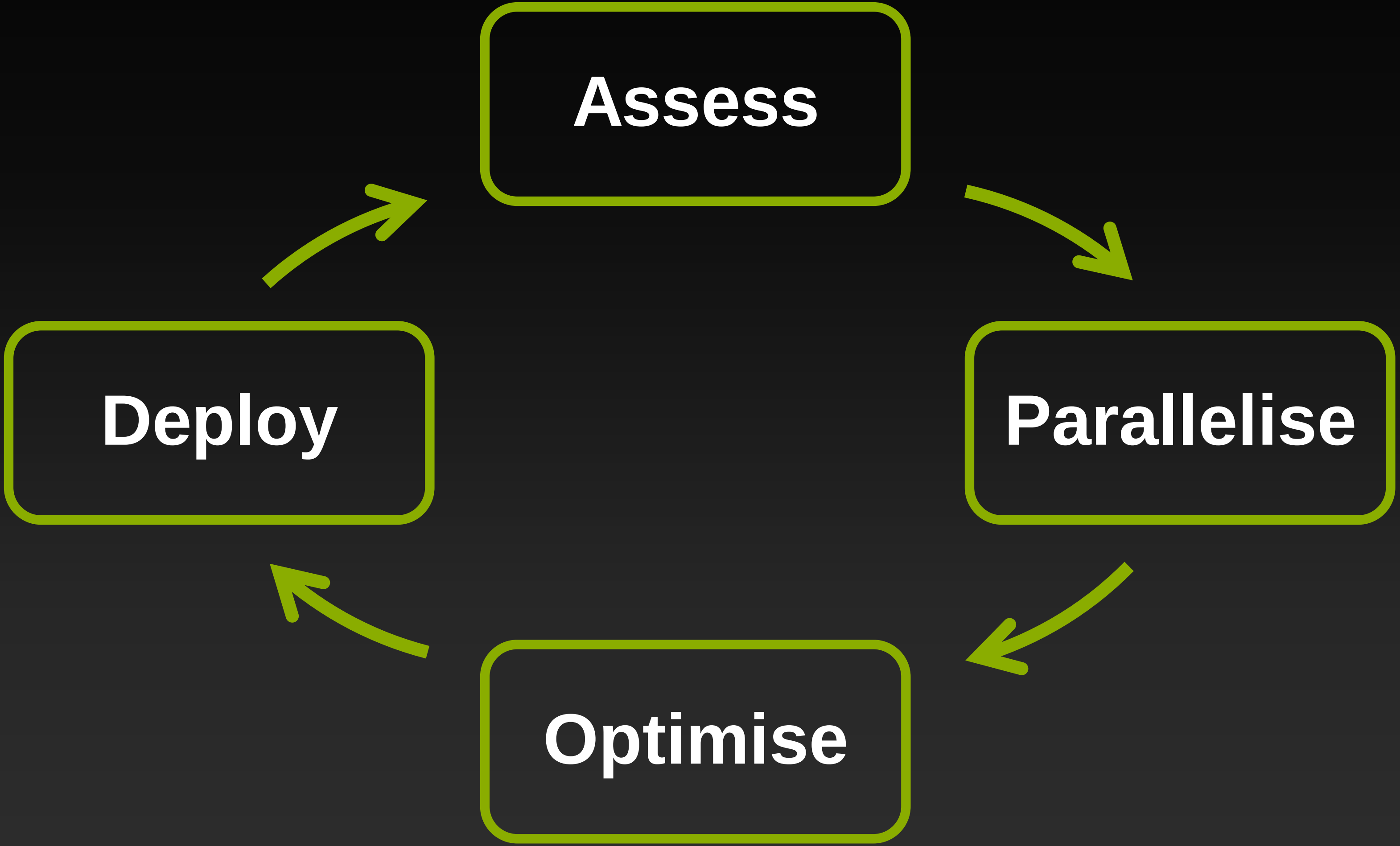
- **False dependencies prevent overlap**
- **Breadth-first launch gives overlap, but more complex code**

Pipeline With Hyper-Q



- Full overlap of all engines
- Simple to program

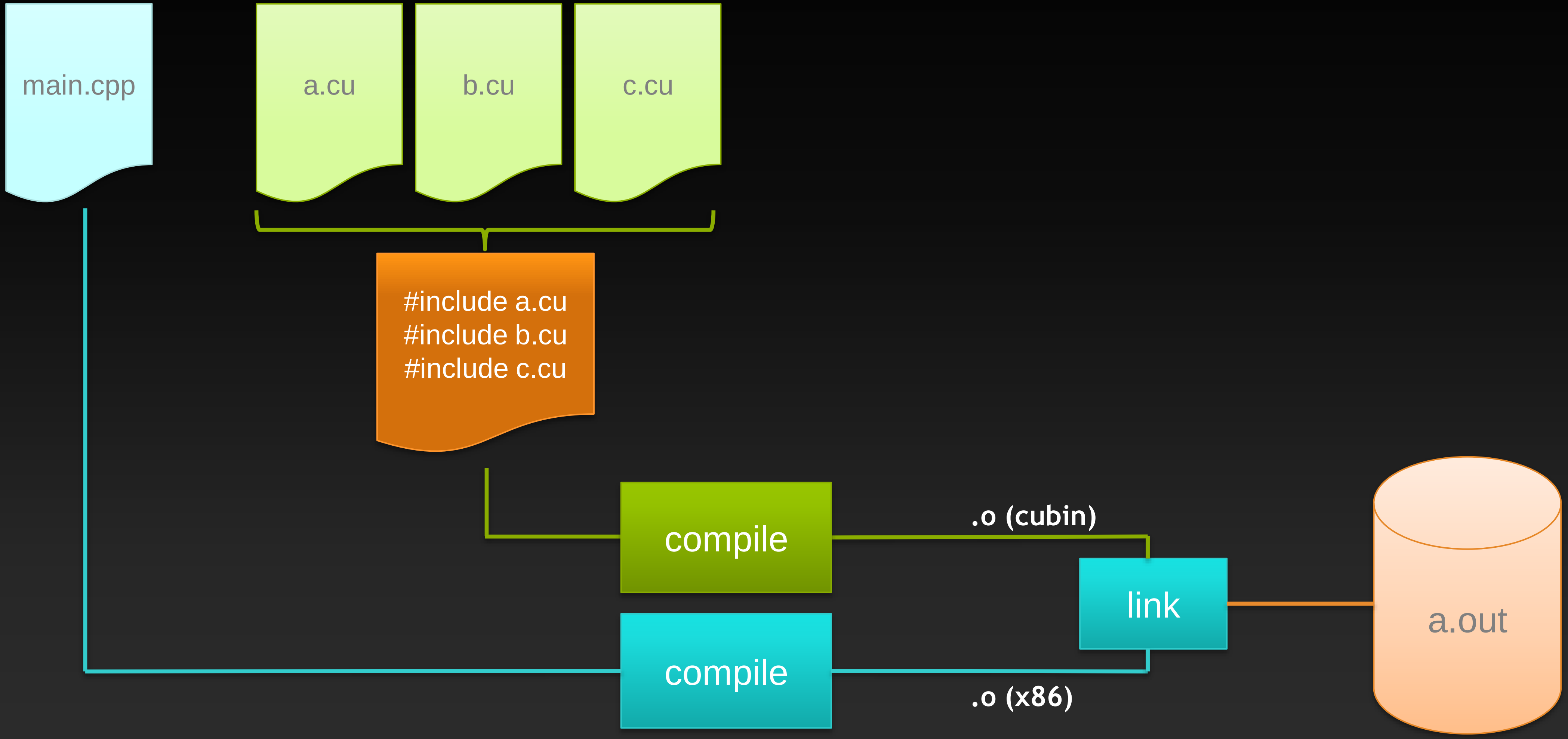
APOD: A Systematic Path to Performance



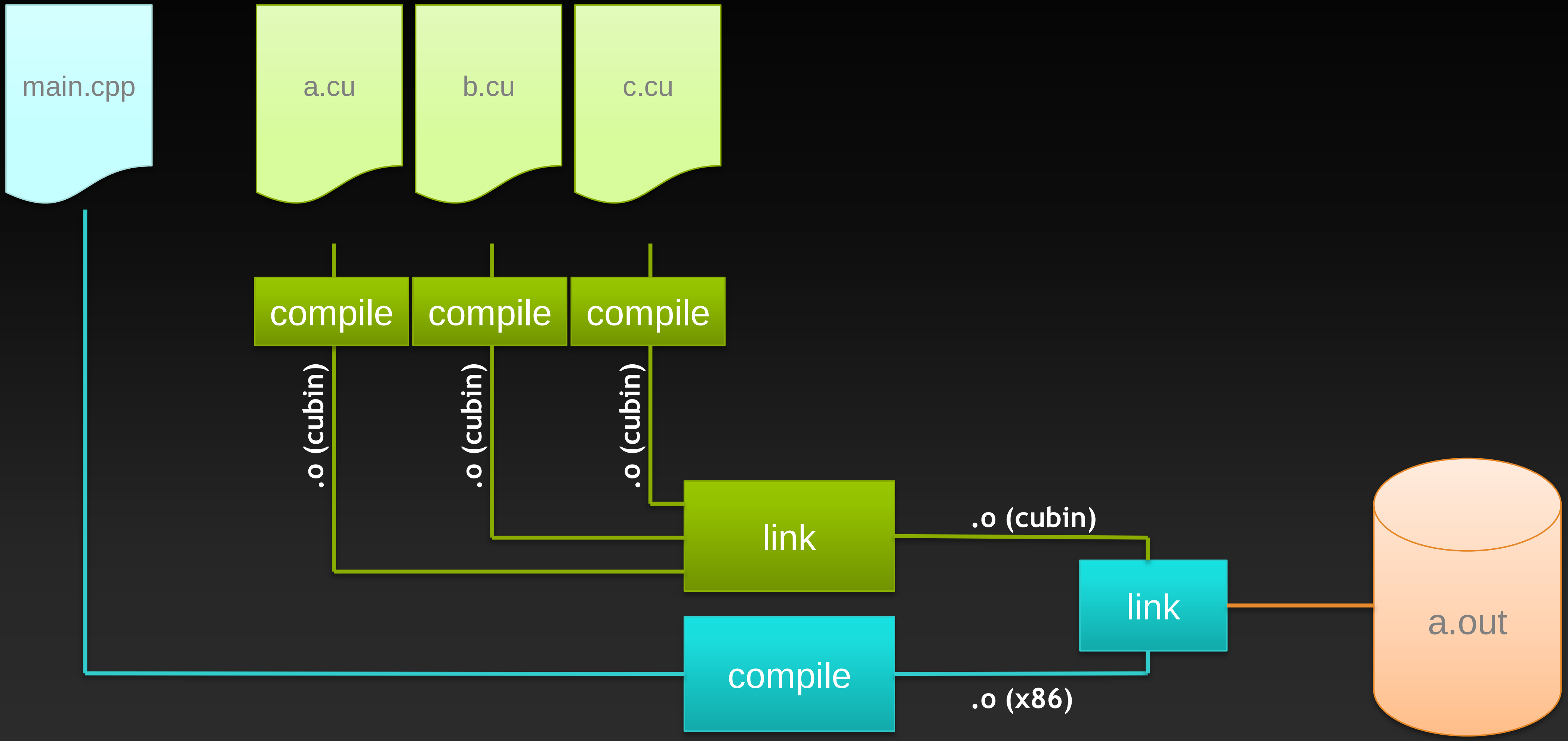
CUDA 5 HIGHLIGHTS: FERMI + KEPLER

Separate Compilation and Linking

Whole-Program Compilation (CUDA 4.2)



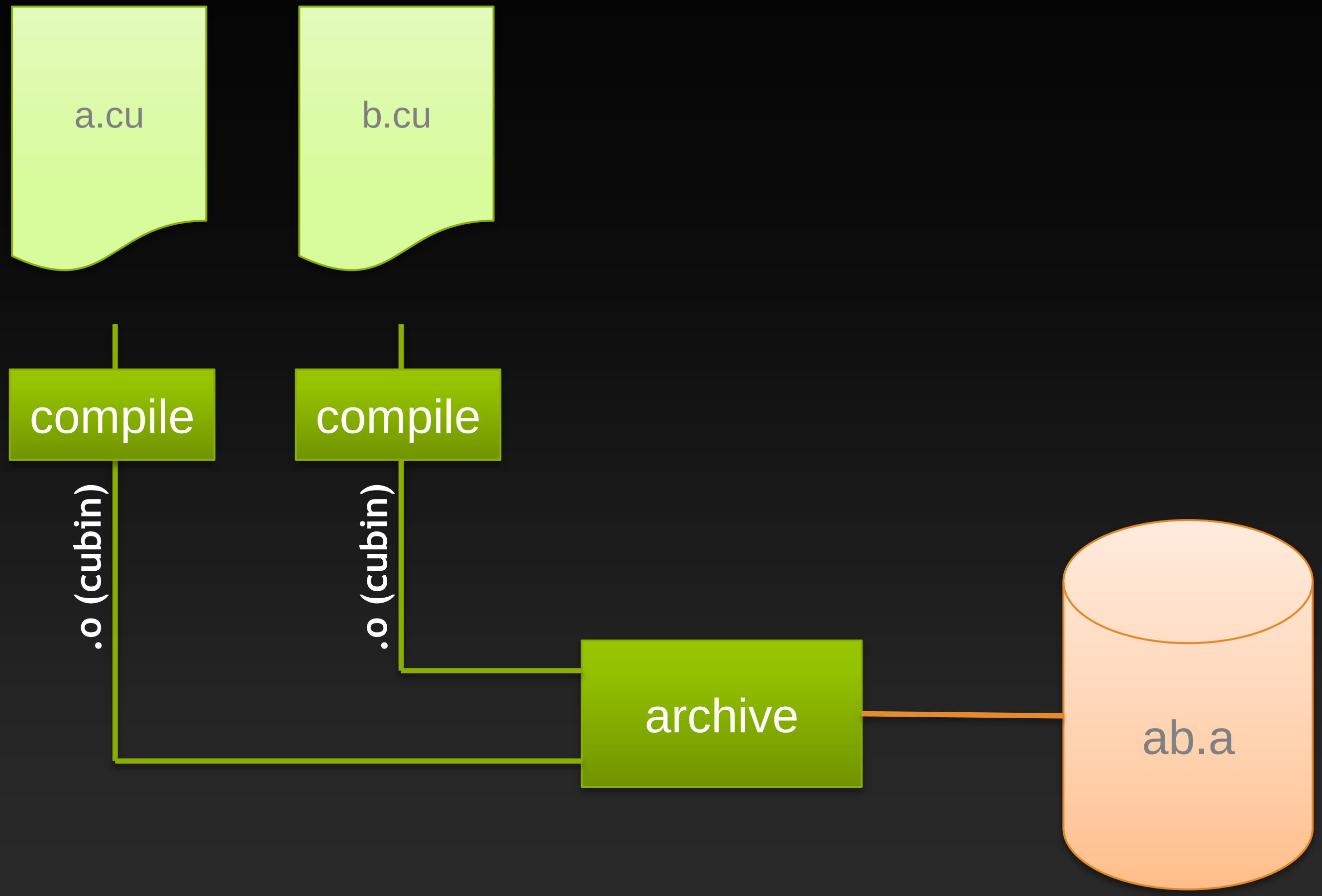
Separate Compilation & Linking (CUDA 5)



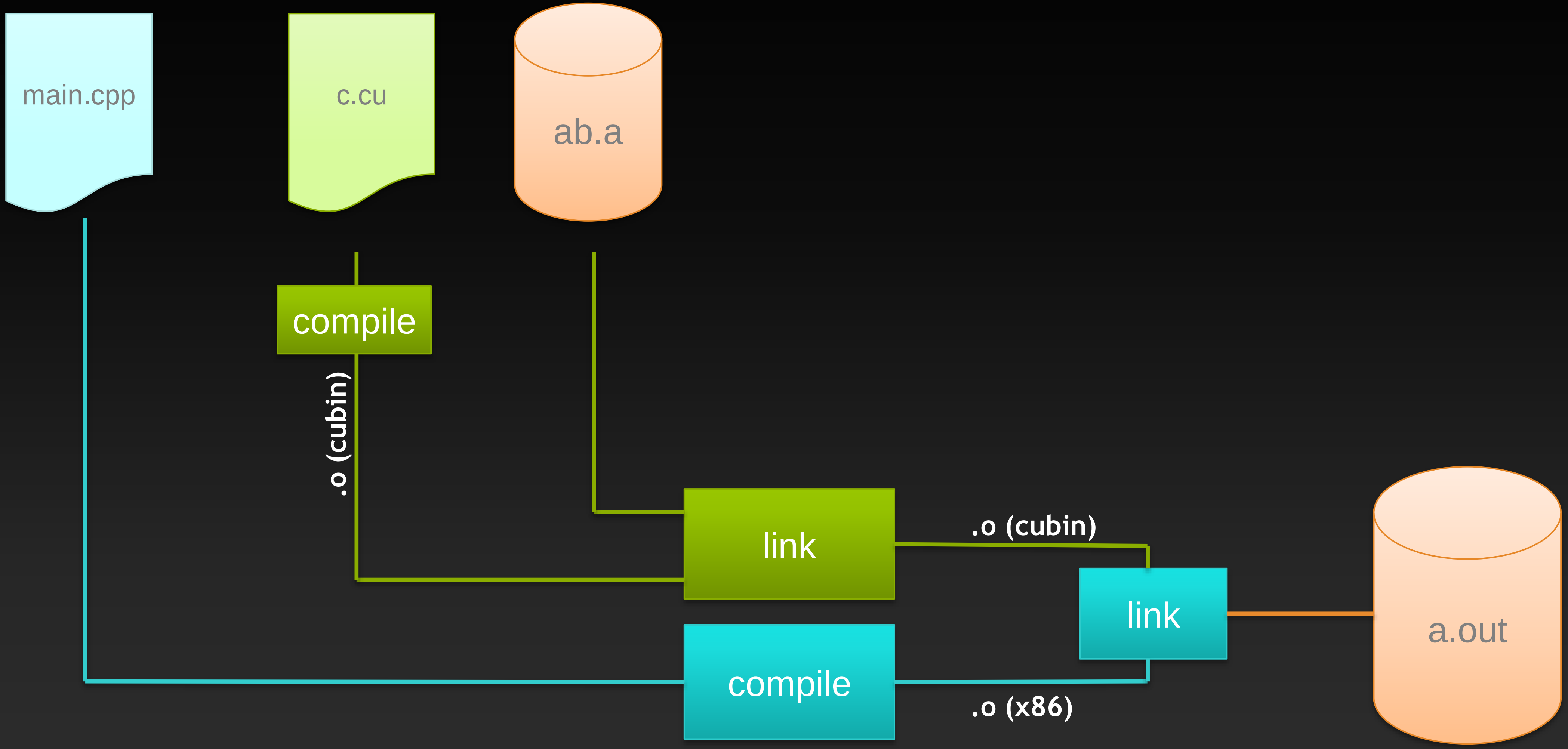
Separate Compilation: Benefits

- Simpler code reuse
  - No need to `#include` everything
  - `extern` attribute respected
- Reduced build time
  - Incremental compilation
- GPU-callable libraries
  - 3rd party or custom
  - e.g., `libcublas_device.a`

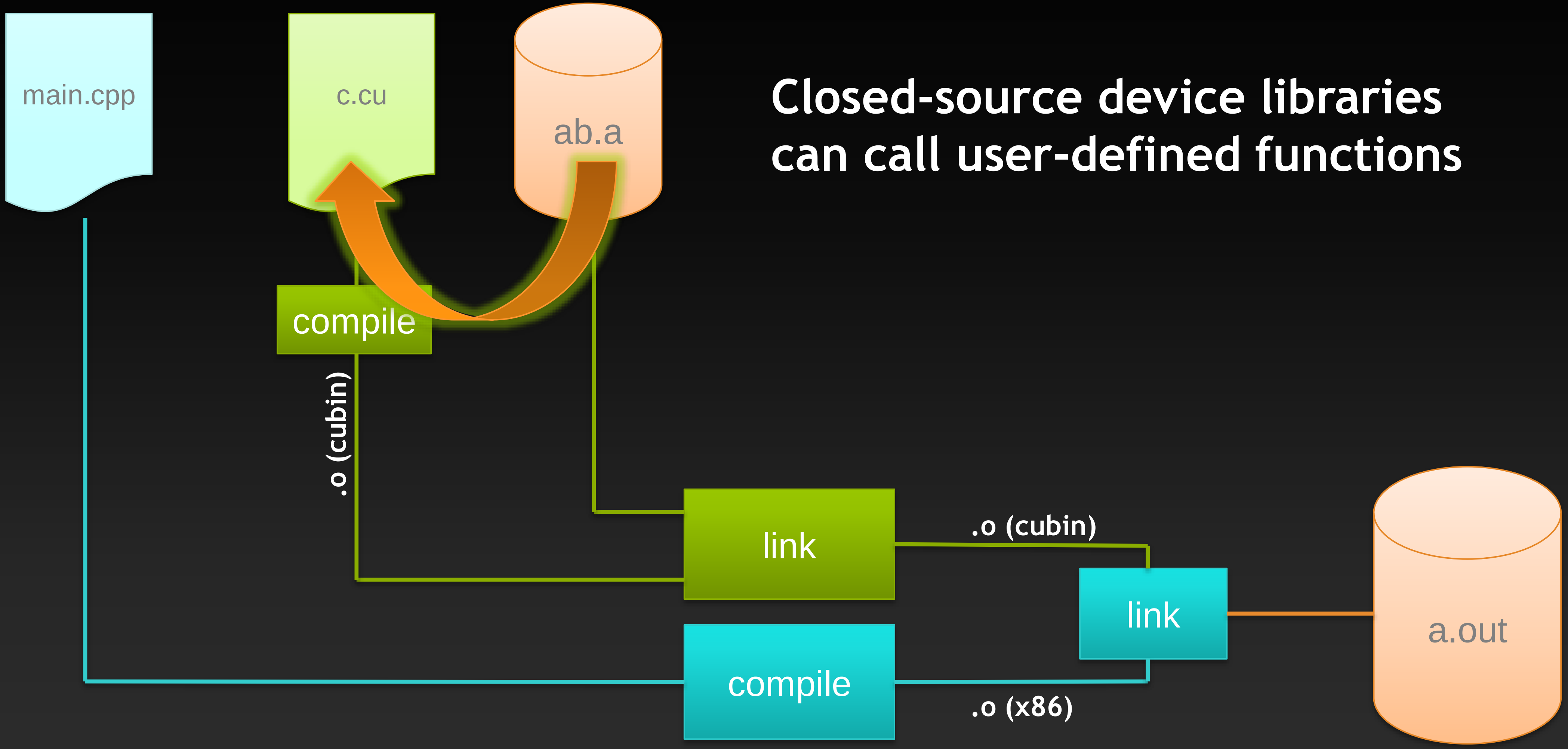
GPU-callable Libraries



GPU-callable Libraries



GPU Callbacks



Closed-source device libraries can call user-defined functions

Device Linker Invocation

- Optional link step for device code:

```
nvcc -arch=sm_20 -dc a.cu b.cu
nvcc -arch=sm_20 -dlink a.o b.o -o link.o
g++ a.o b.o link.o -L<path> -lcudart
```

- Link device-runtime library for dynamic parallelism

```
nvcc -arch=sm_35 -dc a.cu b.cu
nvcc -arch=sm_35 -dlink a.o b.o -lcudadevrt -o link.o
g++ a.o b.o link.o -L<path> -lcudadevrt -lcudart
```

- Link is at cubin level (PTX link is not yet supported)

cuda-memcheck

Functional correctness checking suite

- Memory access check (default)
  - cuda-memcheck ./a.out
- Memory leak detection
  - cuda-memcheck --leak-check ./a.out
  - Requires cudaDeviceReset()
- Shared memory hazard check
  - cuda-memcheck --tool racecheck ./a.out

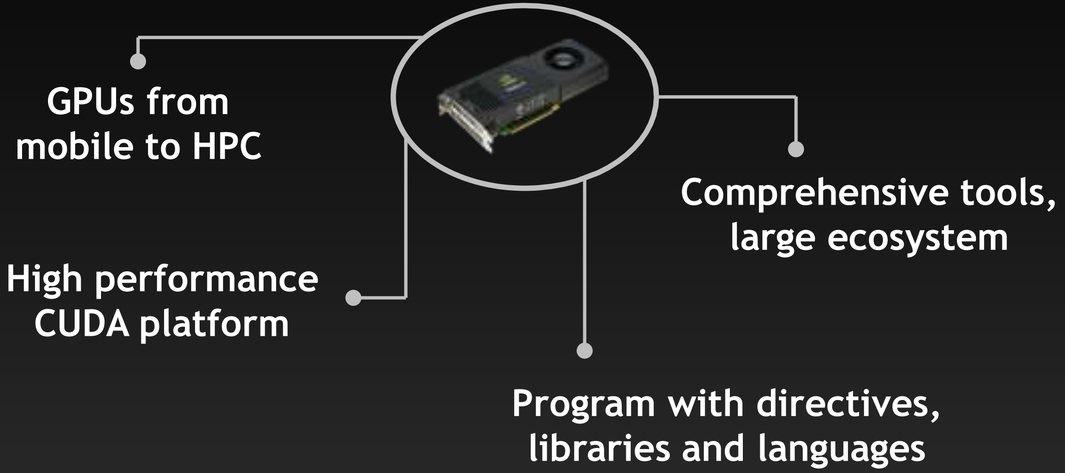
Additional Information

nvidia.com/cuda

nvidia.com/kepler

docs.nvidia.com/cuda

gputechconf.com


GPUs from mobile to HPC
High performance CUDA platform
Comprehensive tools, large ecosystem
Program with directives, libraries and languages

Additional Information: GTC

- Kepler architecture:
  - GTC 2012 S0642: Inside Kepler
- Assessing performance limiters:
  - GTC 2012 S0514: GPU Performance Analysis and Optimization
- Profiling tools:
  - GTC 2012 S0419: CUDA Performance Tools
  - GTC 2012 S0420: Nsight IDE for Linux and Mac
- GPU computing webinars:
  - developer.nvidia.com/gpu-computing-webinars

GPU TECHNOLOGY CONFERENCE
GPU TECHNOLOGY CONFERENCE 2013
MARCH 18-21, 2013 | SAN JOSE, CALIFORNIA

GTC On-Demand
Discover the Power of GPU Computing!
Relive all of the Sessions from GTC 2012
EXPLORE NOW

http://www.gputechconf.com/gtcnew/on-demand-gtc.php