Fermi Hardware & Performance Tips

Dr. Timo Stich (tstich@nvidia.com)
NVIDIA | Developer Technology Engineer

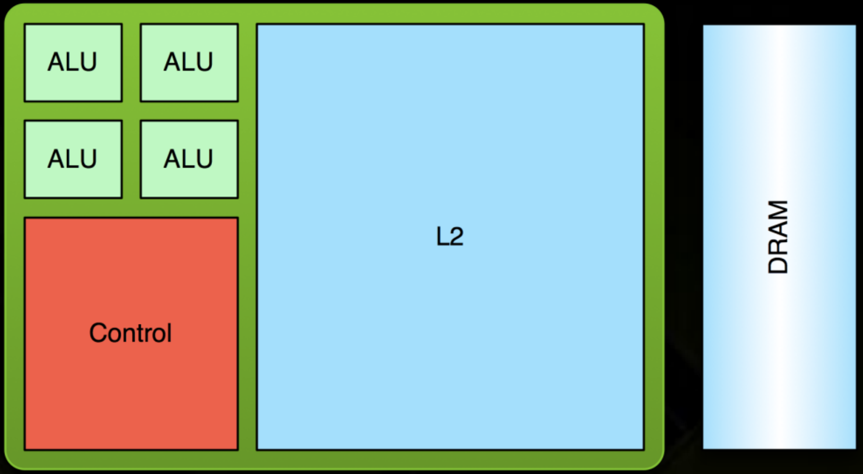
Outline

- Fermi Hardware Overview
  - SM Architecture
  - Memories & Caches
  - Instructions & Flow Control
  - CPU-GPU interaction
- Tools & Libraries
  - Profiler
  - Debugger
  - FFT, BLAS, Sparse Math, Rand etc

GPU Computing

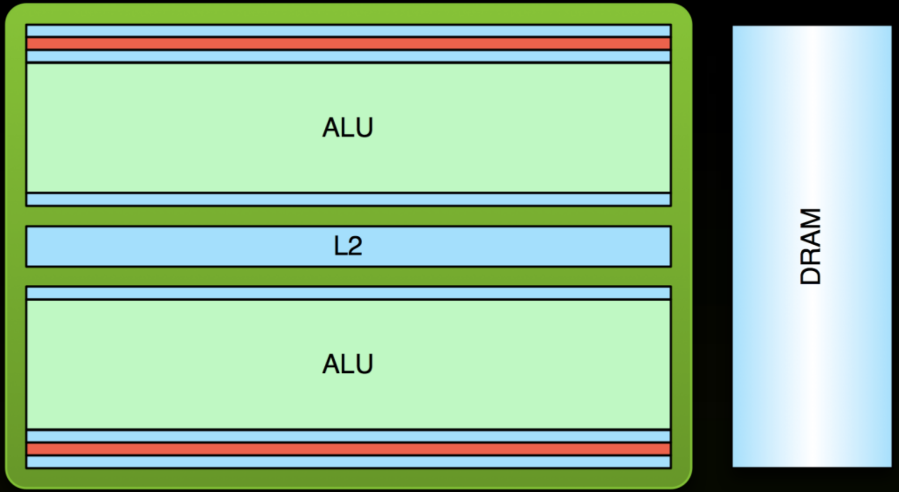
THE BIG PICTURE

Differences between CPU & GPUs



CPU

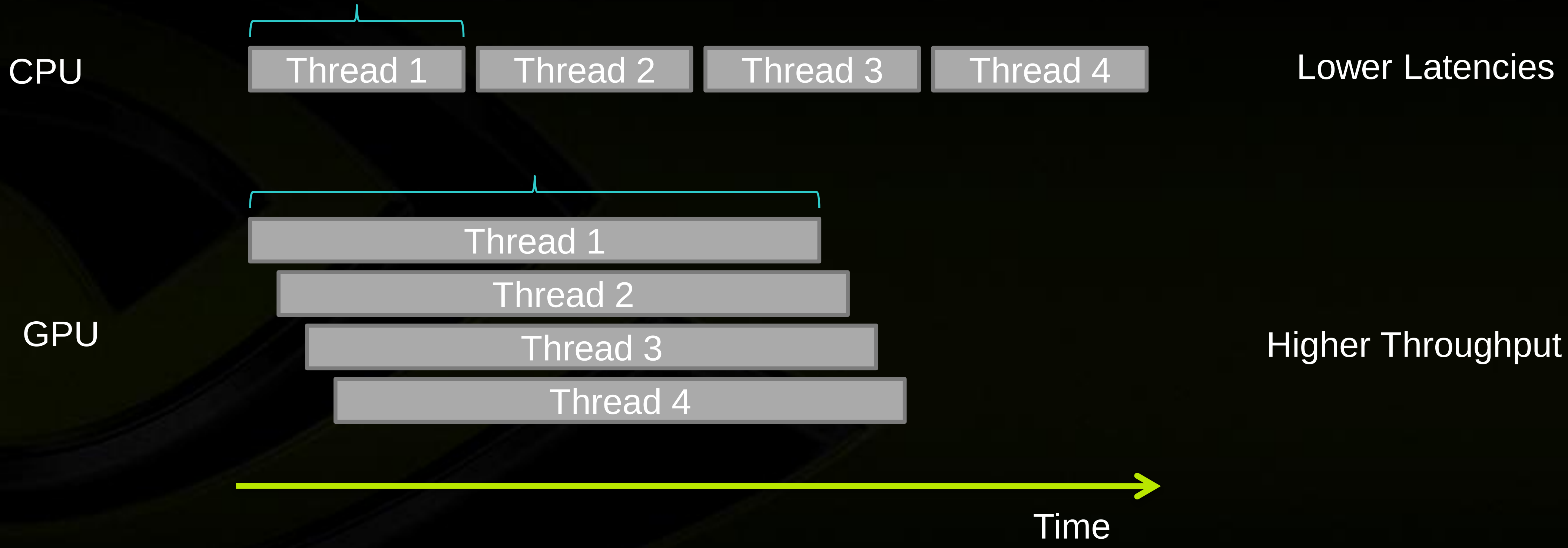
- Optimized for low-latency, serial computation
- Huge caches
- Control logic for out-of-order and speculative execution



GPU

- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- More transistors dedicated to computation

CPU vs GPU: The Thread POV



CPU

Thread 1

Thread 2

Thread 3

Thread 4

Lower Latencies

GPU

Thread 1

Thread 2

Thread 3

Thread 4

Higher Throughput

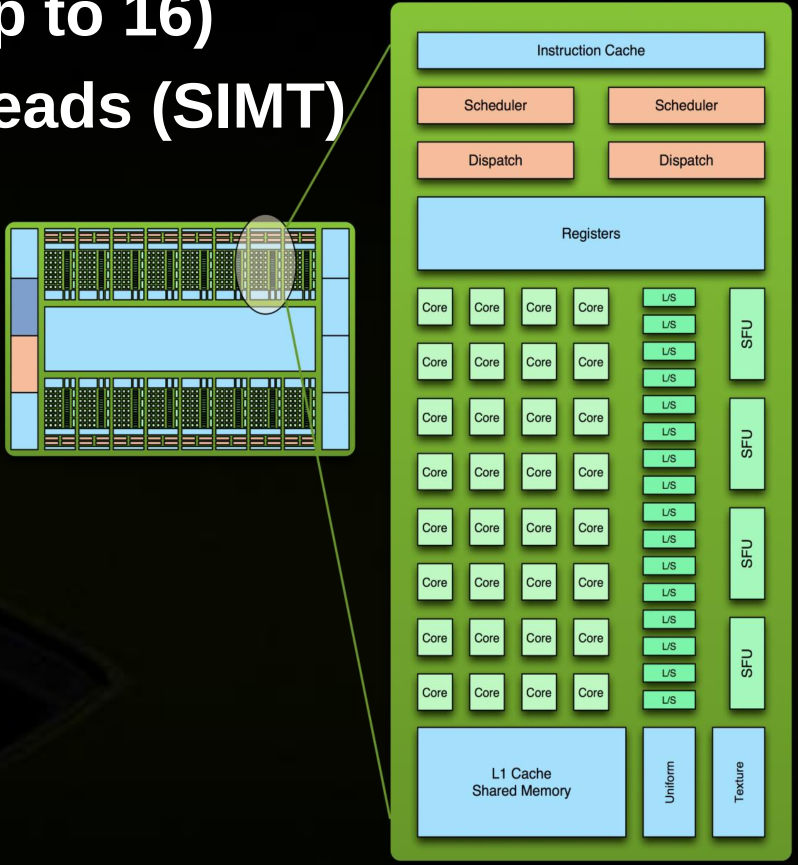
Time

CUDA Parallel Programming

FERMI HARDWARE REVIEW

Fermi SM Architecture

- GPU is structured into SMs (up to 16)
- Single instruction multiple threads (SIMT)
  - Executes warps of 32 threads
  - 1 to 48 warps per SM
  - 1 to 8 blocks per SM
  - Can issue 2 warps/cycle
- 32 CUDA cores per SM
  - 32 FP32 FMA units
  - 16 FP64 FMA units
- 64KB configurable
  L1 $ / shared memory



Warps

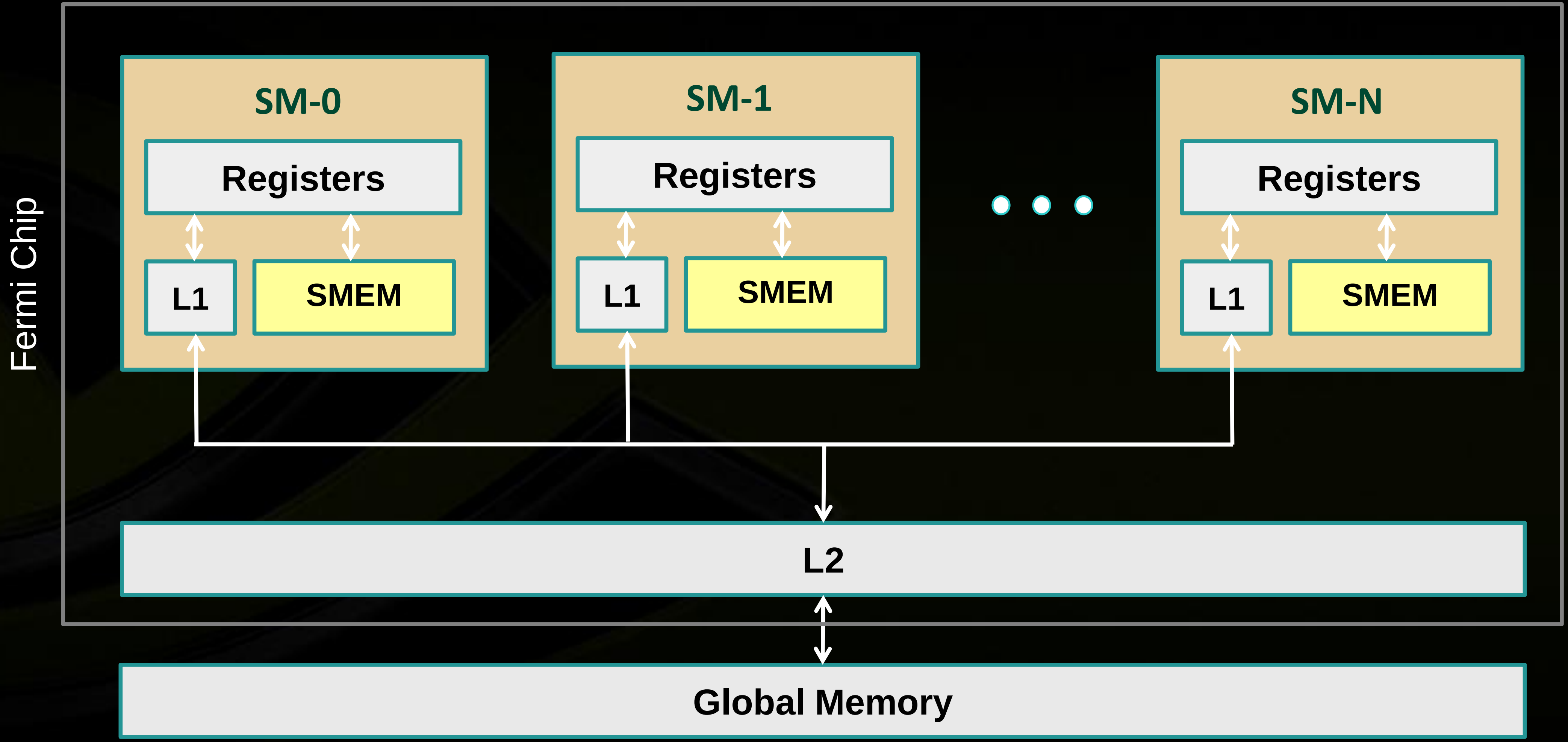
- Group of 32 Threads
  - Threads execute in lock-step (similar to Vectorprocessor)
- Divergence within warps is supported on the HW level
  - May result in serialization/instruction replays

Warp	ThreadIdx				
0	0	1	...	30	31
1	32	33	...	62	63

Fermi Memory Hierarchy Review

- Local storage
  - Each thread has own local storage
  - Mostly registers (managed by the compiler)
- Shared memory / L1
  - Program configurable: 16KB shared / 48 KB L1   OR   48KB shared / 16KB L1
  - Shared memory is accessible by the threads in the same threadblock
  - Very low latency
  - Very high throughput: 1+ TB/s aggregate
- L2
  - All accesses to global memory go through L2, including copies to/from CPU host
- Global memory
  - Accessible by all threads as well as host (CPU)
  - High latency (400-800 cycles)
  - Throughput: up to 177 GB/s

Fermi Memory Hierarchy Review



Fermi GMEM Operations

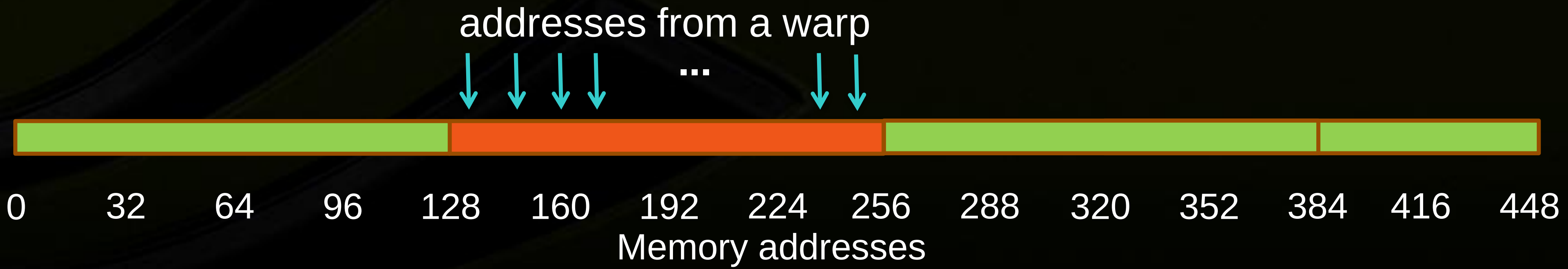
- Two types of loads:
  - Caching
    - Default mode
    - Attempts to hit in L1, then L2, then GMEM
    - Load granularity is 128-byte line
  - Non-caching
    - Compile with *–Xptxas –dlcm=cg* option to nvcc
    - Attempts to hit in L2, then GMEM
      - Do not hit in L1, invalidate the line if it's in L1 already
    - Load granularity is 32-bytes
- Stores:
  - Invalidate L1, write-back for L2

Load Operation

- Memory operations are issued per warp (32 threads)
  - Just like all other instructions
  - Prior to Fermi, memory issues were per half-warp
- Operation:
  - Threads in a warp provide memory addresses
  - Determine which lines/segments are needed
  - Request the needed lines/segments

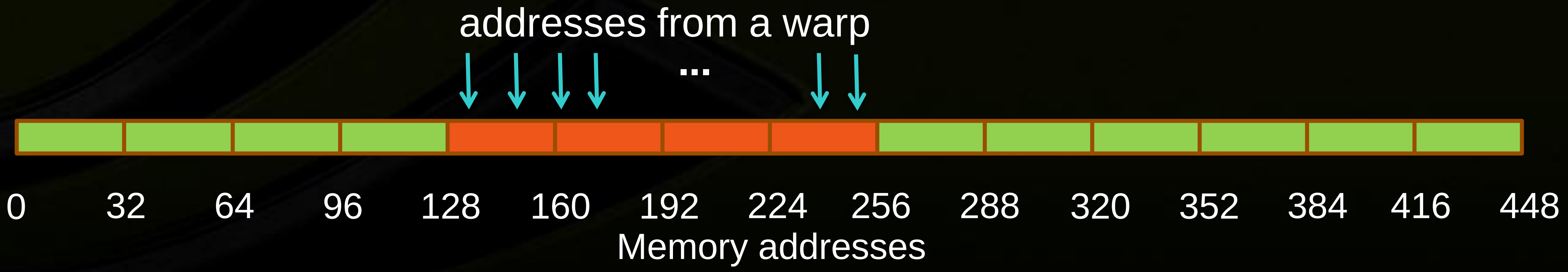
Caching Load

- Warp requests 32 aligned, consecutive 4-byte words
- Addresses fall within 1 cache-line
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 100%



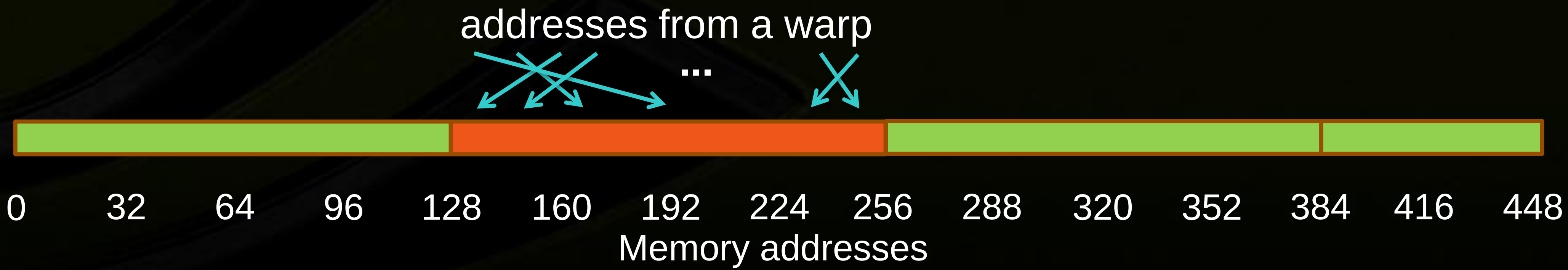
Non-caching Load

- Warp requests 32 aligned, consecutive 4-byte words
- Addresses fall within 4 segments
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 100%



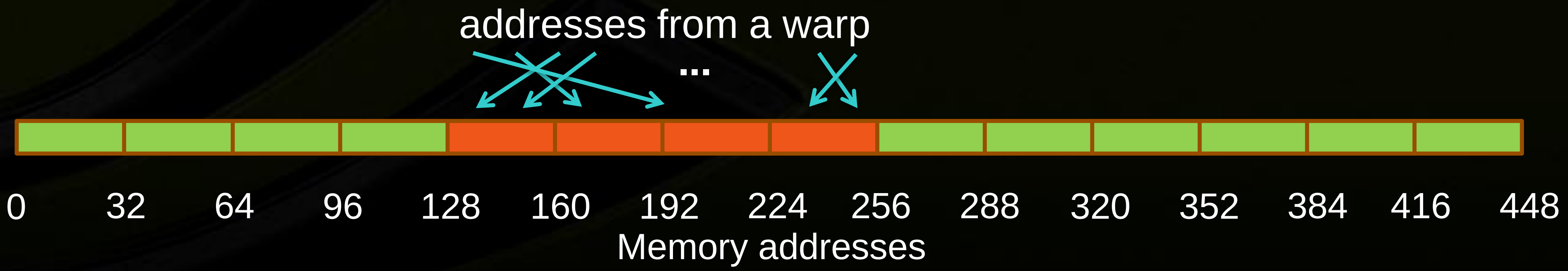
Caching Load

- **Warp requests 32 aligned, permuted 4-byte words**
- **Addresses fall within 1 cache-line**
  - **Warp needs 128 bytes**
  - **128 bytes move across the bus on a miss**
  - **Bus utilization: 100%**



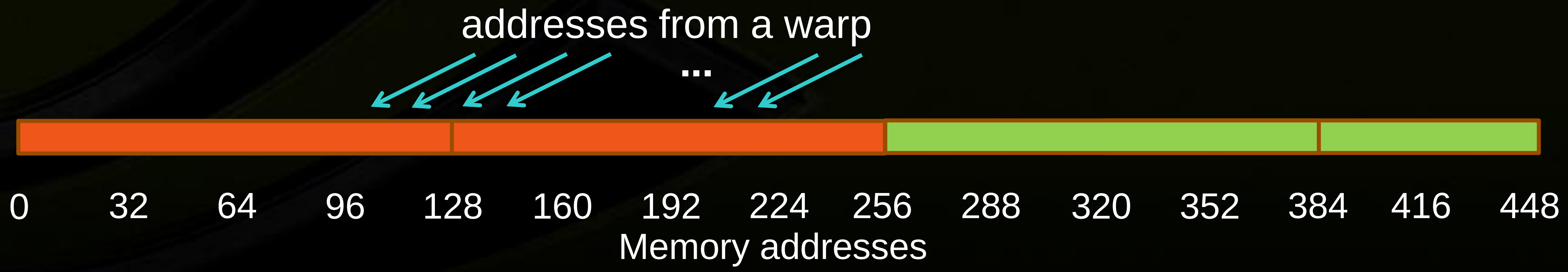
Non-caching Load

- Warp requests 32 aligned, permuted 4-byte words
- Addresses fall within 4 segments
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 100%



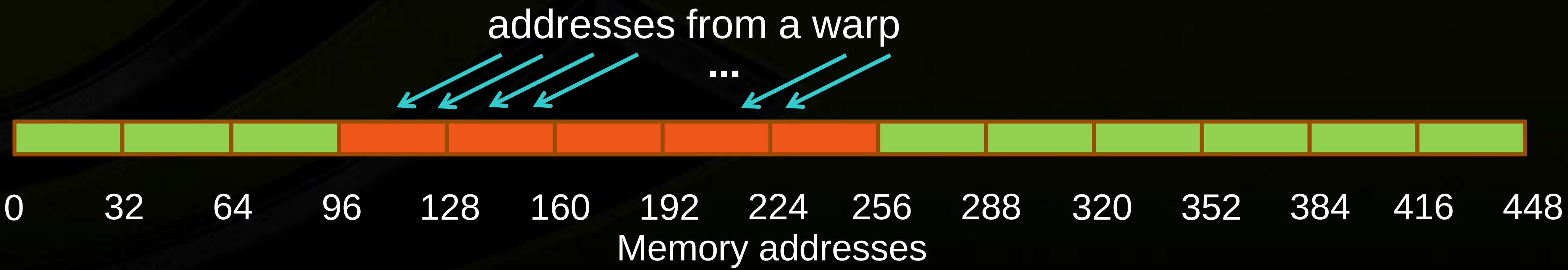
Caching Load

- **Warp requests 32 misaligned, consecutive 4-byte words**
- **Addresses fall within 2 cache-lines**
  - **Warp needs 128 bytes**
  - **256 bytes move across the bus on misses**
  - **Bus utilization: 50%**



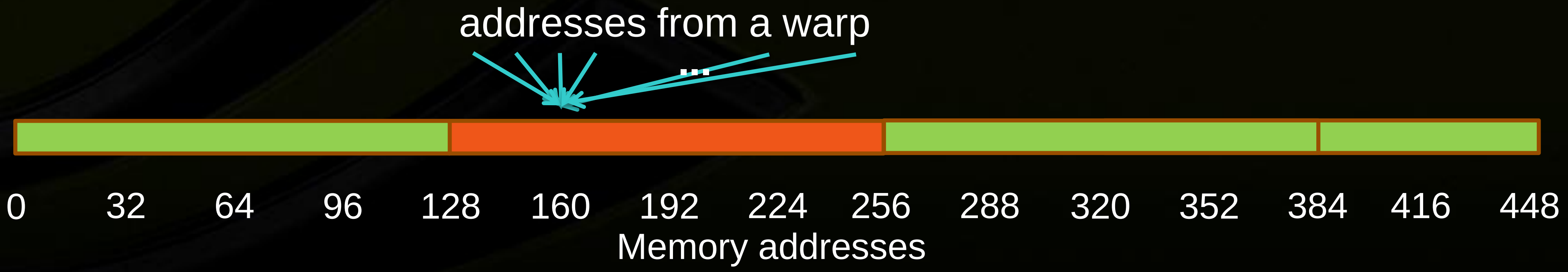
Non-caching Load

- Warp requests 32 misaligned, consecutive 4-byte words
- Addresses fall within at most 5 segments
  - Warp needs 128 bytes
  - 256 bytes move across the bus on misses
  - Bus utilization: at least 80%
    - Some misaligned patterns will fall within 4 segments, so 100% utilization



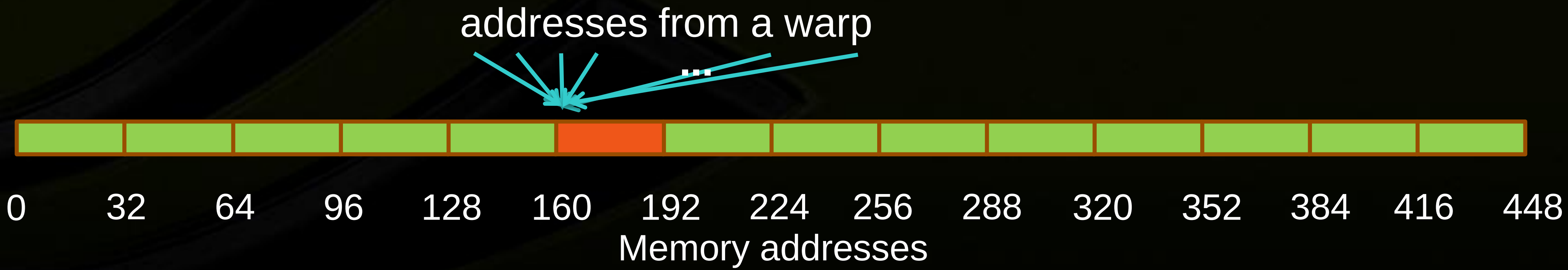
Caching Load

- All threads in a warp request the same 4-byte word
- Addresses fall within a single cache-line
  - Warp needs 4 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 3.125%



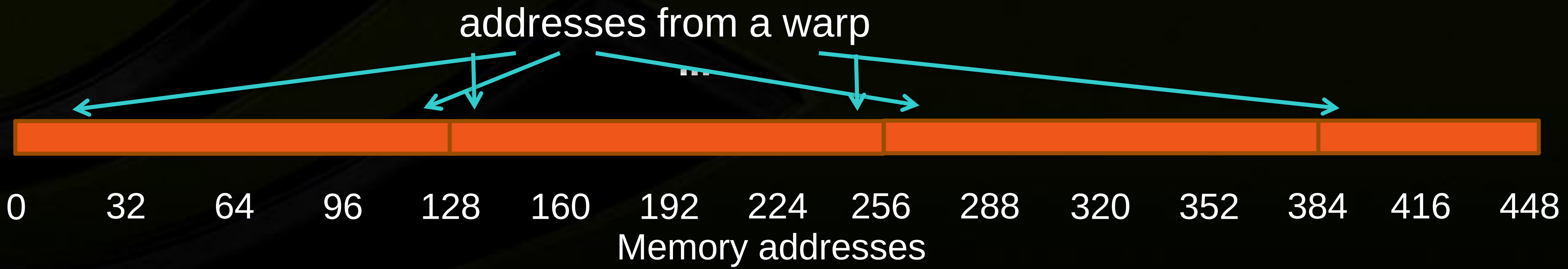
Non-caching Load

- All threads in a warp request the same 4-byte word
- Addresses fall within a single segment
  - Warp needs 4 bytes
  - 32 bytes move across the bus on a miss
  - Bus utilization: 12.5%



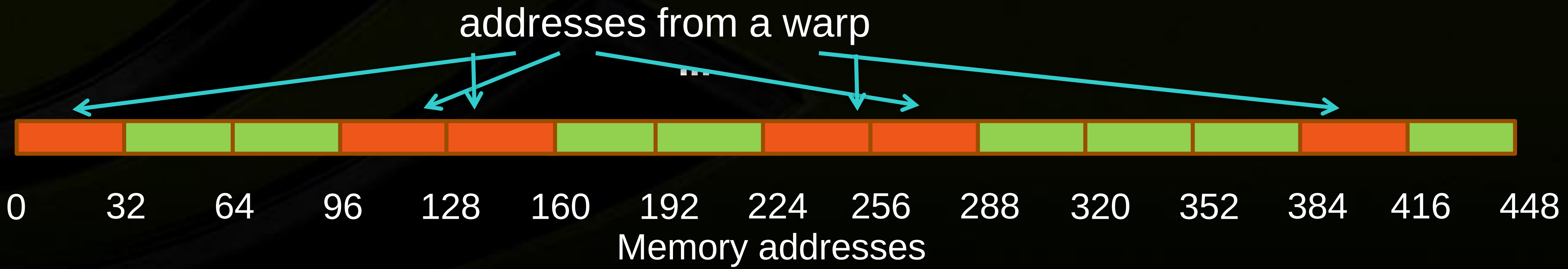
Caching Load

- Warp requests 32 scattered 4-byte words
- Addresses fall within *N* cache-lines
  - Warp needs 128 bytes
  - *N**128 bytes move across the bus on a miss
  - Bus utilization: 128 / (*N**128)



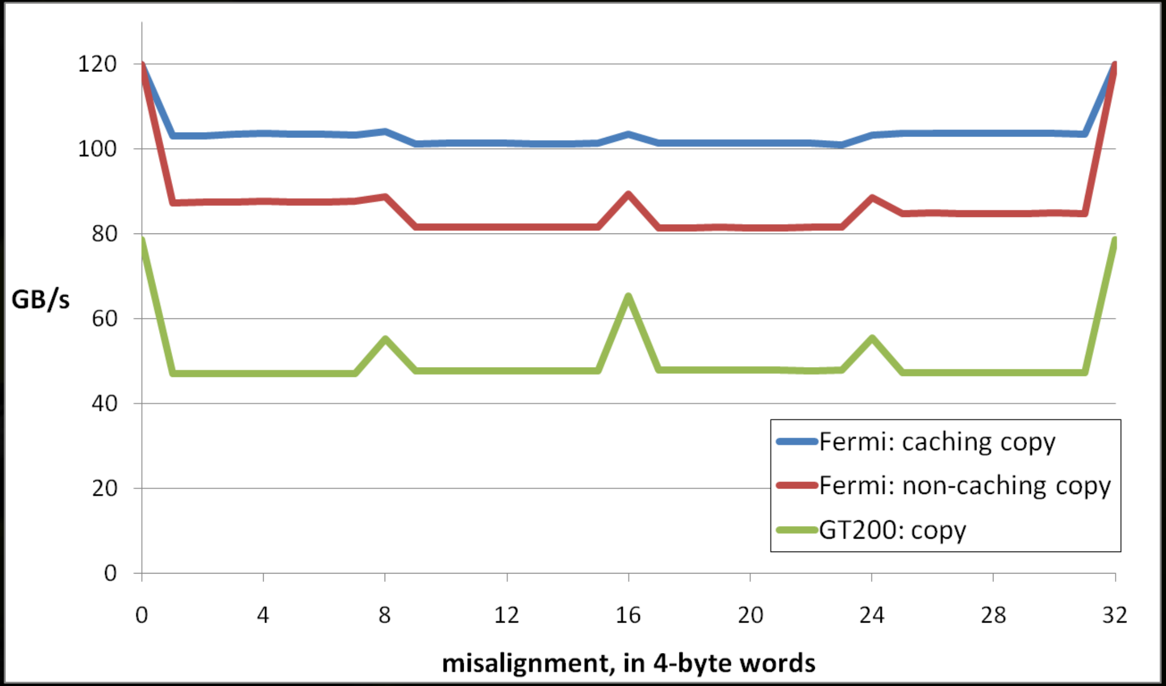
Non-caching Load

- Warp requests 32 scattered 4-byte words
- Addresses fall within *N* segments
  - Warp needs 128 bytes
  - *N**32 bytes move across the bus on a miss
  - Bus utilization: 128 / (*N**32)



Impact of Address Alignment

- **Warps should access aligned regions for maximum memory throughput**
  - **Fermi L1 can help for misaligned loads if several warps are accessing a contiguous region**
  - **ECC further significantly reduces misaligned store throughput**



Experiment:

- Copy 16MB of floats
- 256 threads/block

Greatest throughput drop:

- GT200: 40%
- Fermi:
  - CA loads: 15%
  - CG loads: 32%

GMEM Optimization Guidelines

- Strive for perfect coalescing
  - Align starting address (may require padding)
  - A warp should access within a contiguous region
- Have enough concurrent accesses to saturate the bus
  - Process several elements per thread
    - Multiple loads get pipelined
    - Indexing calculations can often be reused
  - Launch enough threads to maximize throughput
    - Latency is hidden by switching threads (warps)
- Try L1 and caching configurations to see which one works best
  - Caching vs non-caching loads (compiler option)
  - 16KB vs 48KB L1 (CUDA call)

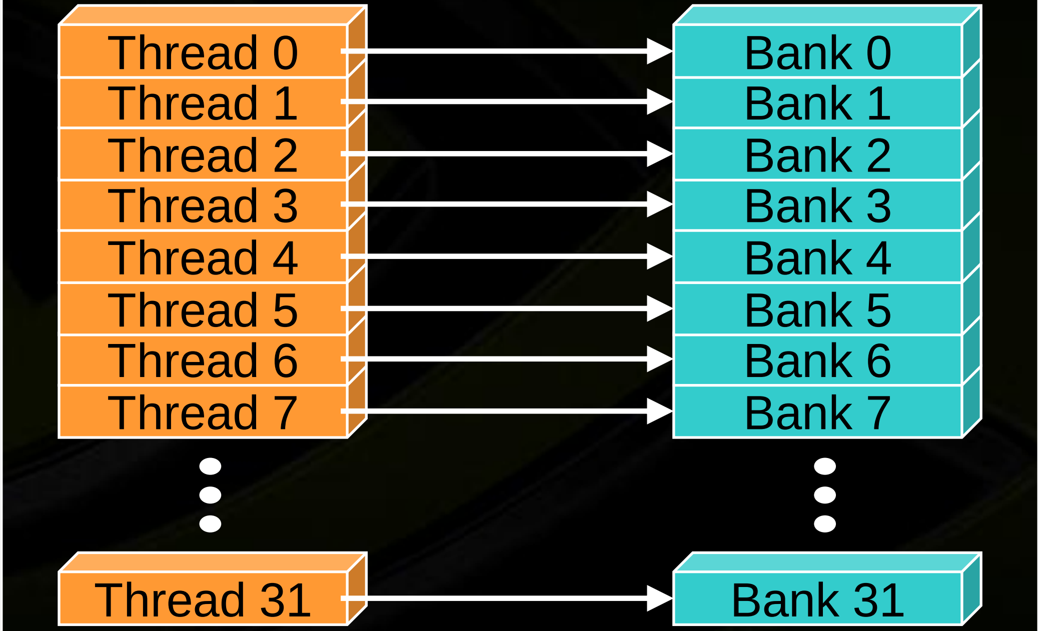
Shared Memory

Shared Memory

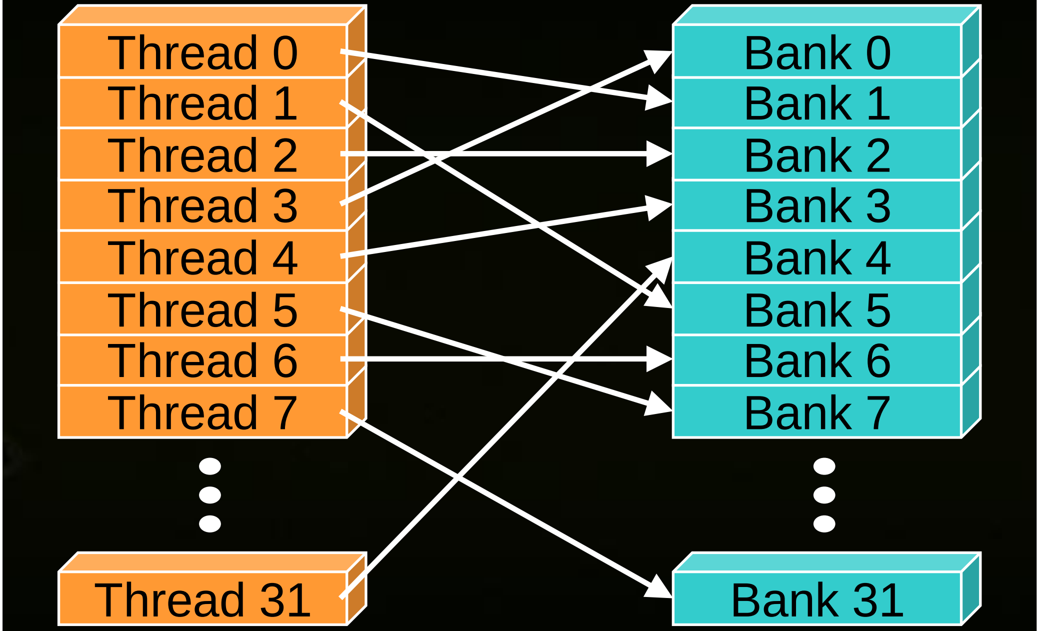
- Uses:
  - Inter-thread communication within a block
  - Cache data to reduce redundant global memory accesses
  - Use it to improve global memory access patterns
- Organization:
  - 32 banks, 4-byte wide banks
  - Successive 4-byte words belong to different banks
- Performance:
  - 4 bytes per bank per 2 clocks per multiprocessor
  - smem accesses are issued per 32 threads (warp)
    - per 16-threads for GPUs prior to Fermi
  - serialization: if *n* threads of 32 access different 4-byte words in the same bank, *n* accesses are executed serially
  - multicast: *n* threads access the same word in one fetch
    - Could be different bytes within the same word
    - Prior to Fermi, only broadcast was available, sub-word accesses within the same bank caused serialization

Bank Addressing Examples

- No Bank Conflicts

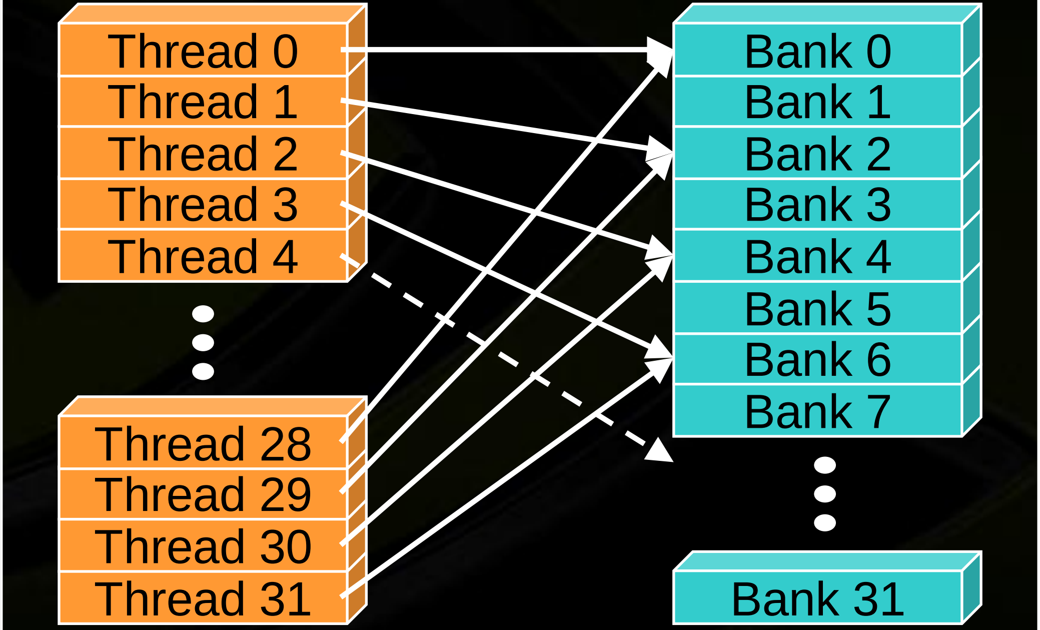


- No Bank Conflicts

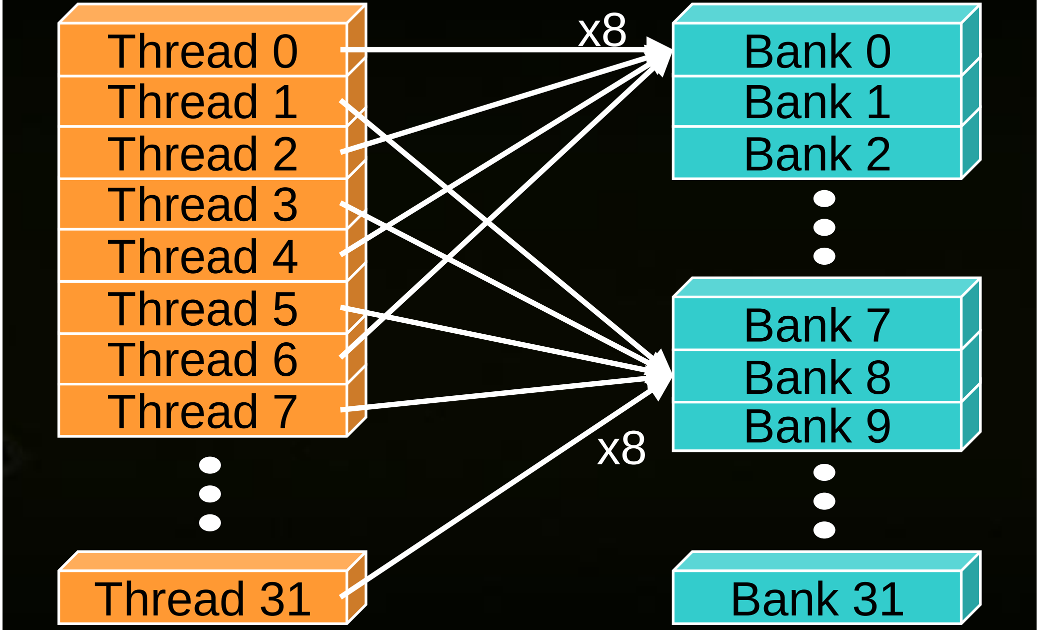


Bank Addressing Examples

- 2-way Bank Conflicts

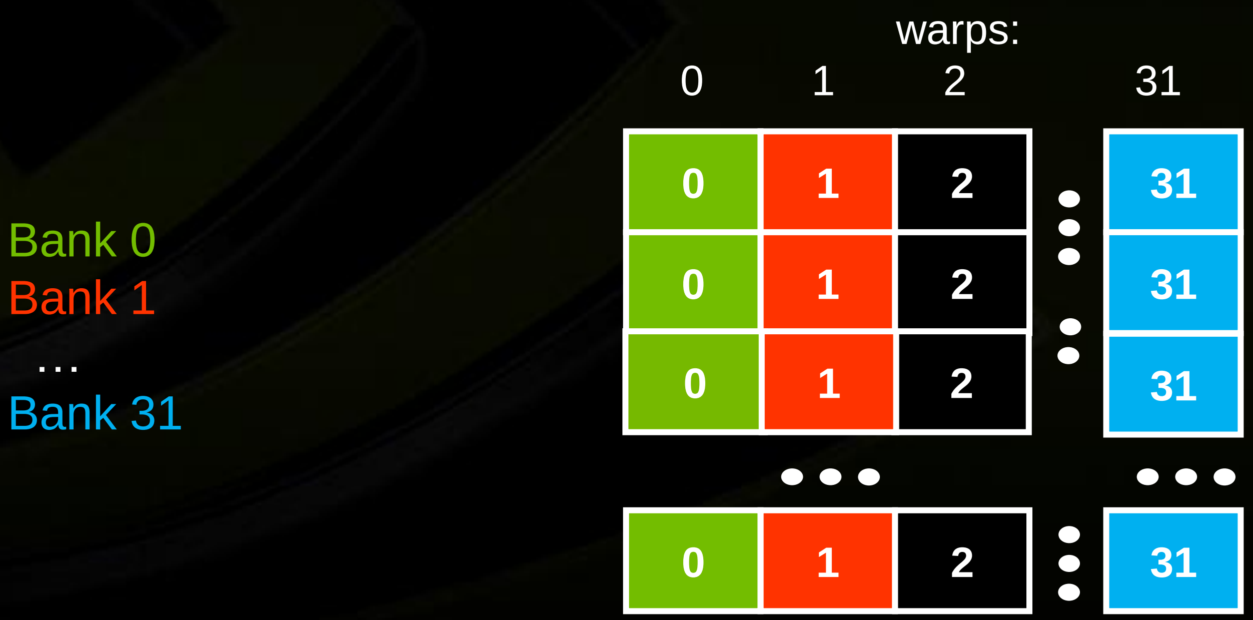


- 8-way Bank Conflicts



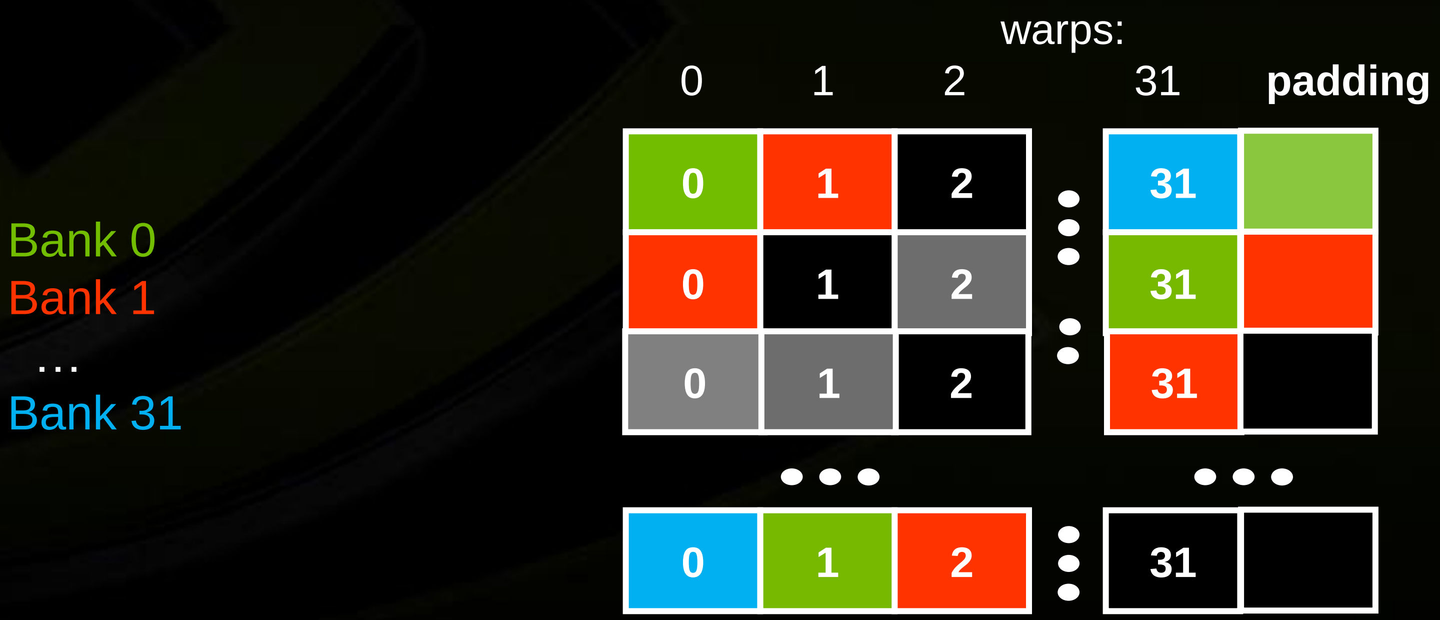
Shared Memory: Avoiding Bank Conflicts

- 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



Additional “memories”

- ***Texture*** **and** ***constant***
- **Read-only**
- **Data resides in global memory**
- **Read through different caches**

Constant Memory

- Ideal for coefficients and other data that is read uniformly by warps
- Data is stored in global memory, read through a constant-cache
  - __constant__ qualifier in declarations
  - Can only be read by GPU kernels
  - Limited to 64KB
- Fermi adds uniform accesses:
  - Kernel pointer argument qualified with *const*
  - Compiler must determine that all threads in a threadblock will dereference the same address
  - No limit on array size, can use any global memory pointer
- Constant cache throughput:
  - 32 bits per warp per 2 clocks per multiprocessor
  - To be used when all threads in a warp read the same address
    - Serializes otherwise

Constant Memory

- Ideal for coefficients and other data that is read uniformly by warps
- Data is stored in global me
  - __constant__ qualifier in
  - Can only be read by GPU
  - Limited to 64KB
- Fermi adds uniform acce
  - Kernel pointer argument
  - Compiler must determin
    the same address
  - No limit on array size, ca
- Constant cache throughp
  - 32 bits per warp per 2 clo
  - To be used when all threads in a warp read the same address
    - Serializes otherwise

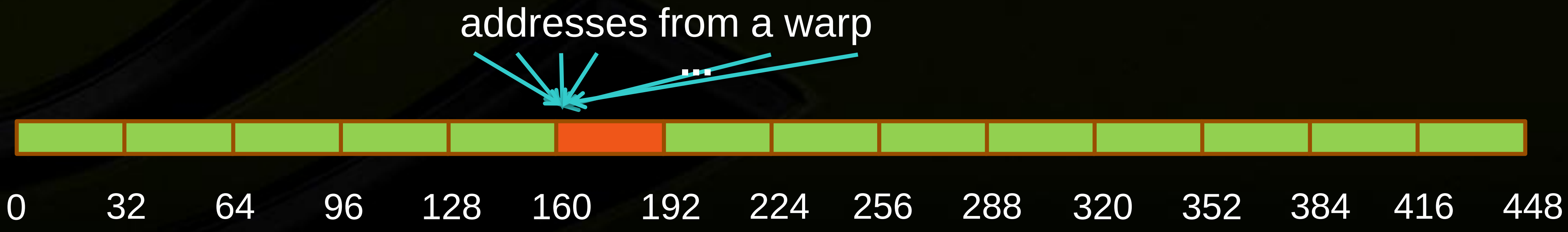
```
__global__ void kernel( const float *g_a )
{
    ...
    float x = g_a[15];                    // uniform
    float y = g_a[blockIdx.x+5];    // uniform
    float z = g_a[threadIdx.x];      // non-uniform
    ...
}
```

Constant Memory

- Ideal for coefficients and other data that is read uniformly by warps
- Data is stored in global memory, read through a constant-cache
  - __constant__ qualifier in declarations
  - Can only be read by GPU kernels
  - Limited to 64KB
- Fermi adds uniform accesses:
  - Kernel pointer argument qualified with *const*
  - Compiler must determine that all threads in a threadblock will dereference the same address
  - No limit on array size, can use any global memory pointer
- Constant cache throughput:
  - 32 bits per warp per 2 clocks per multiprocessor
  - To be used when all threads in a warp read the same address
    - Serializes otherwise

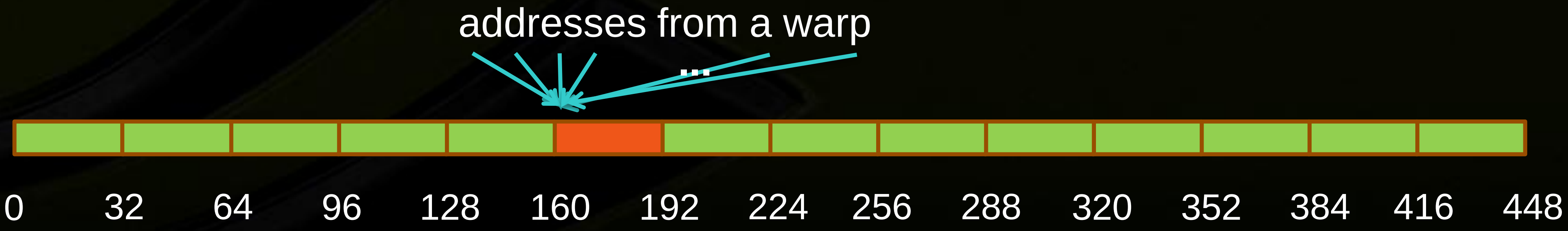
Constant Memory

- Kernel executes 10K threads (320 warps)per SM during its lifetime
- All threads access the same 4B word
- Using GMEM:
  - Each warp fetches 32B -> 10KB of bus traffic
  - Caching loads potentially worse – 128B line, very likely to be evicted multiple times



Constant Memory

- Kernel executes 10K threads (320 warps)per SM during its lifetime
- All threads access the same 4B word
- Using constant/uniform access:
  - First warp fetches 32 bytes
  - All others hit in constant cache -> 32 bytes of bus traffic
    - Unlikely to be evicted over kernel lifetime – other loads do not go through this cache



Texture

- Separate cache
- Dedicated texture cache hardware provides:
  - Out-of-bounds index handling
    - clamp or wrap-around
  - Optional interpolation
    - Think: using fp indices for arrays
    - Linear, bilinear, trilinear
      - Interpolation weights are 9-bit
  - Optional format conversion
    - {char, short, int} -> float
  - All of these are “free”

Instruction Throughput / Control Flow

Runtime Math Library and Intrinsics

- Two types of runtime math library functions
  - __func(): many map directly to hardware ISA
    - Fast but lower accuracy (see CUDA Programming Guide for full details)
    - Examples: __sinf(x), __expf(x), __powf(x, y)
  - func(): compile to multiple instructions
    - Slower but higher accuracy (5 ulp or less)
    - Examples: sin(x), exp(x), pow(x, y)

- A number of additional intrinsics:
  - __sincosf(), __frcp_rz(), ...
  - Explicit IEEE rounding modes (rz,rn,ru,rd)

Control Flow

- Instructions are issued per 32 threads (warp)
- Divergent branches:
  - Threads within a single warp take different paths
    - `if-else`, ...
  - Different execution paths within a warp are serialized
- Different warps can execute different code with no impact on performance
- Avoid diverging within a warp
  - Example with divergence:
    - `if (threadIdx.x > 2) {...} else {...}`
    - Branch granularity < warp size
  - Example without divergence:
    - `if (threadIdx.x / WARP_SIZE > 2) {...} else {...}`
    - Branch granularity is a whole multiple of warp size

Heterogeneous Computing

CPU-GPU INTERACTION

Pinned (non-pageable) memory

- Pinned memory enables:
  - faster PCIe copies (~6 GB/s with PCIe 2.0)
  - memcopies asynchronous with CPU
  - memcopies asynchronous with GPU
- Usage
  - cudaHostAlloc / cudaFreeHost
    - instead of malloc / free
- Implication:
  - pinned memory is essentially removed from host virtual memory

Streams and Async API

- Default API:
  - Kernel launches are asynchronous with CPU
  - Memcopies (D2H, H2D) block CPU thread
  - CUDA calls are serialized by the driver
- Streams and async functions provide:
  - Memcopies (D2H, H2D) asynchronous with CPU
  - Ability to concurrently execute a kernel and a memcopy
- Stream = sequence of operations that execute in issue-order on GPU
  - Operations from different streams can be interleaved
  - A kernel and memcopy from different streams can be overlapped

Overlap kernel and memory copy

- **Requirements:**
  - **D2H or H2D memcopy from pinned memory**
  - **Device with compute capability ≥ 1.1 (G84 and later)**
  - **Kernel and memcopy in different, non-0 streams**
- **Code:**

```
cudaStream_t   stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);

cudaMemcpyAsync( dst, src, size, dir, stream1 );
kernel<<<grid, block, 0, stream2>>>(…);
```

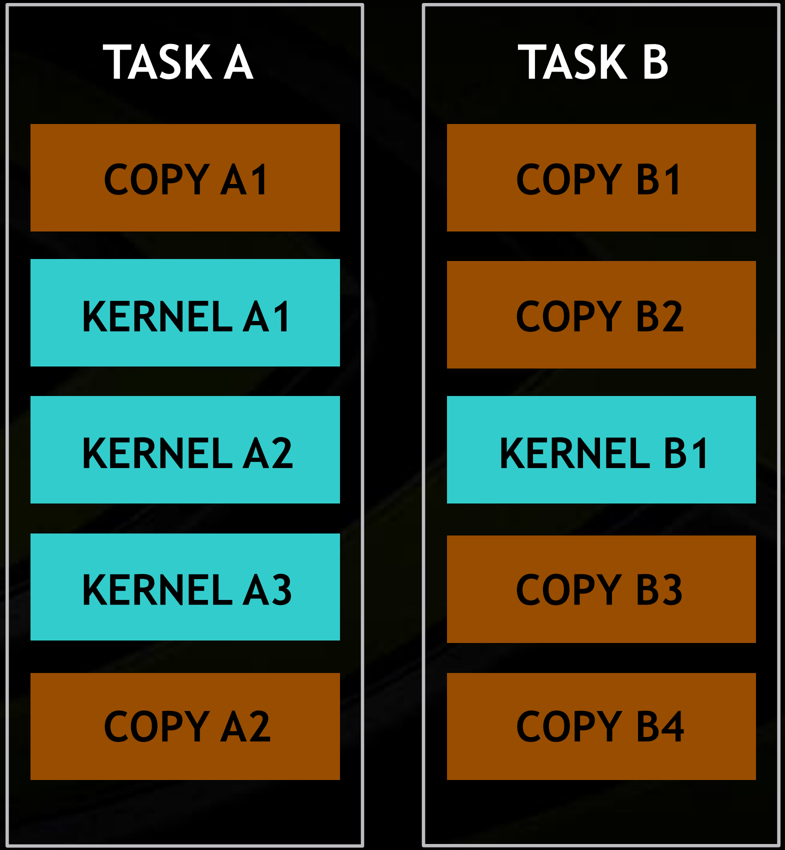
potentially overlapped

Call Sequencing for Optimal Overlap

- CUDA calls are dispatched to the hw in the sequence they were issued
- One kernel and one memcopy can be executed concurrently
- A call is dispatched if both are true:
  - Resources are available
  - Preceding calls in the same stream have completed
- Note that if a call blocks, it blocks all other calls of the same type behind it, even in other streams

CUDA Streams

Independent Tasks



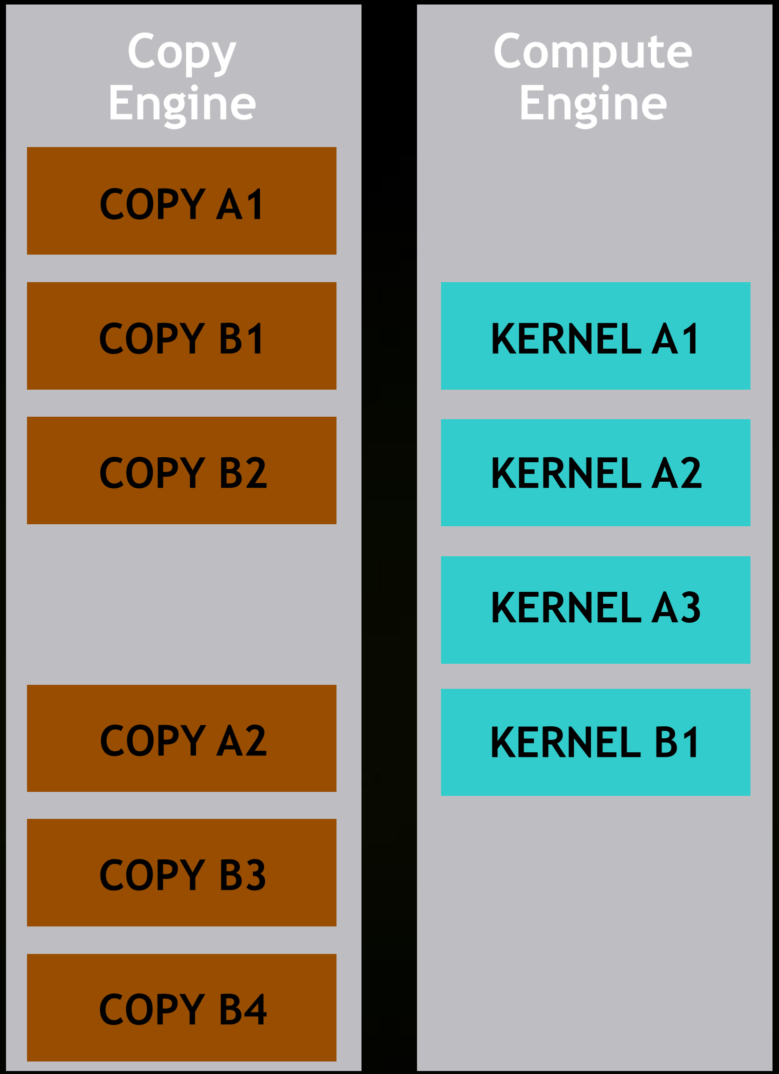
TASK A

- COPY A1
- KERNEL A1
- KERNEL A2
- KERNEL A3
- COPY A2

TASK B

- COPY B1
- COPY B2
- KERNEL B1
- COPY B3
- COPY B4

Scheduling on GPU



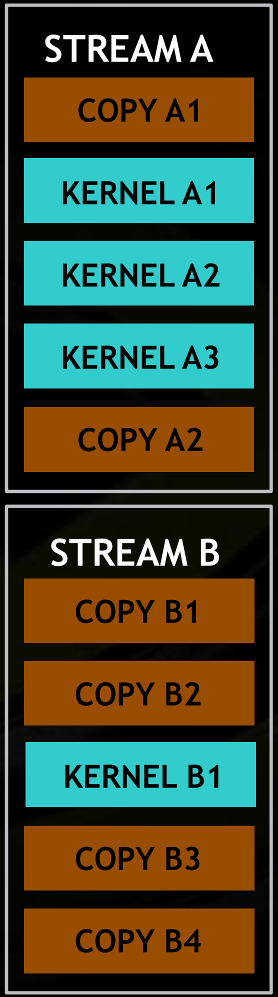
Copy Engine

- COPY A1
- COPY B1
- COPY B2
- COPY A2
- COPY B3
- COPY B4

Compute Engine

- KERNEL A1
- KERNEL A2
- KERNEL A3
- KERNEL B1

Avoid Serialization!



STREAM A

COPY A1

KERNEL A1

KERNEL A2

KERNEL A3

COPY A2

STREAM B

COPY B1

COPY B2

KERNEL B1

COPY B3

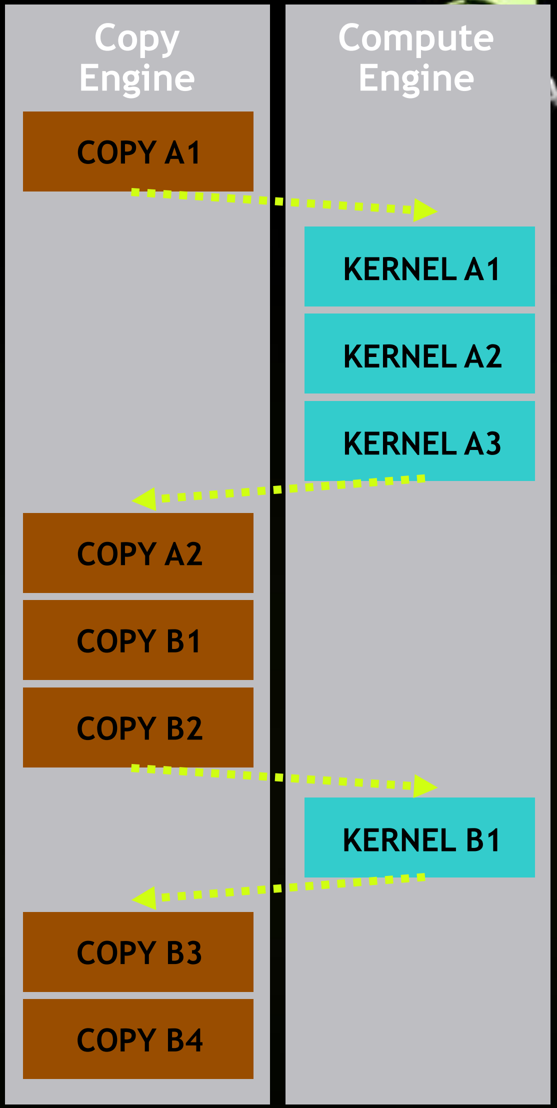
COPY B4

WRONG WAY!

```
CudaMemcpyAsync(A1…,StreamA);
KernelA1<<<…,StreamA>>>();
KernelA2<<<…,StreamA>>>();
KernelA3<<<…,StreamA>>>();
CudaMemcpyAsync(A2…,StreamA);


CudaMemcpyAsync(B1…,StreamB);
CudaMemcpyAsync(B2…,StreamB);
KernelB1<<<…,StreamB>>>();
CudaMemcpyAsync(B2…,StreamB);
CudaMemcpyAsync(B2…,StreamB);
```

- **Engine queues are filled in the order code is executed**



Copy Engine

COPY A1

COPY A2

COPY B1

COPY B2

COPY B3

COPY B4

Compute Engine

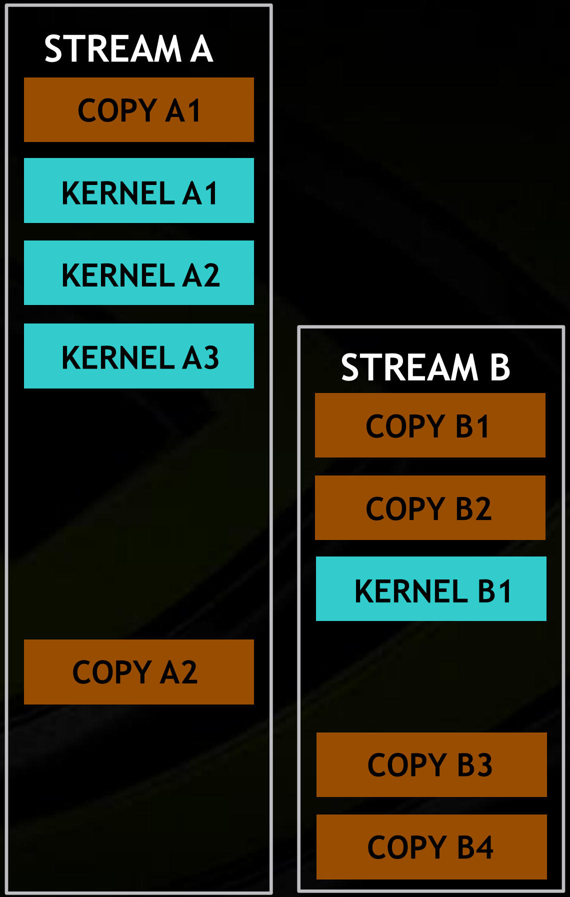
KERNEL A1

KERNEL A2

KERNEL A3

KERNEL B1

Stream Code Order


STREAM A

COPY A1

KERNEL A1

KERNEL A2

KERNEL A3

COPY A2

STREAM B

COPY B1

COPY B2

KERNEL B1

COPY B3

COPY B4

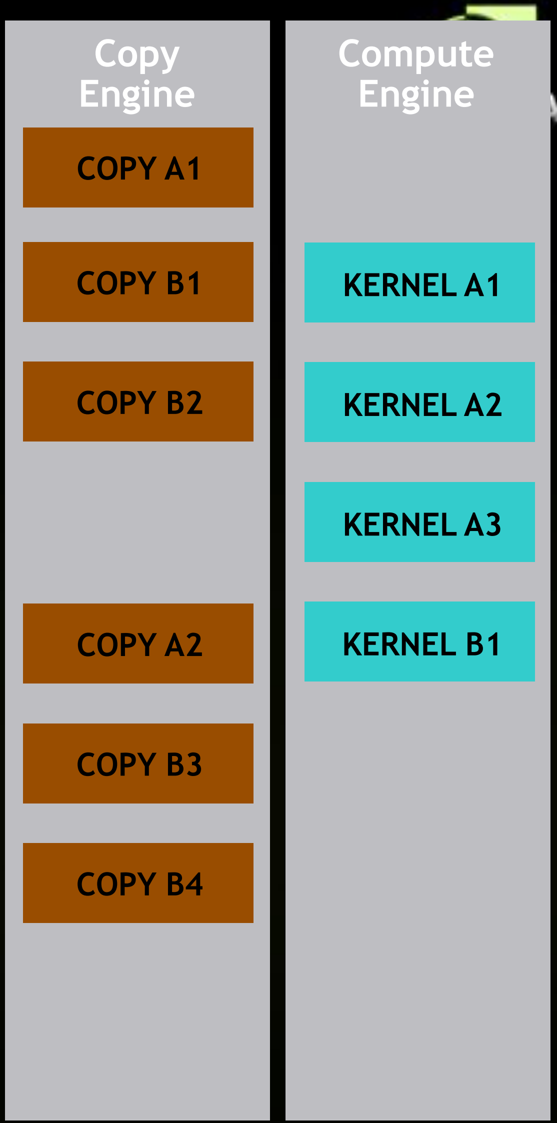
CORRECT WAY!

```
CudaMemcpyAsync(A1…,StreamA);
KernelA1<<<…,StreamA>>>();
KernelA2<<<…,StreamaA>>>();
KernelA3<<<…,StreamA>>>();


CudaMemcpyAsync(B1…,StreamB);
CudaMemcpyAsync(B2…,StreamB);
KernelB1<<<…,StreamB>>>();

CudaMemcpyAsync(A2…,StreamA);

CudaMemcpyAsync(B2…,StreamB);
CudaMemcpyAsync(B2…,StreamB);
```


Copy Engine

COPY A1

COPY B1

COPY B2

COPY A2

COPY B3

COPY B4

Compute Engine

KERNEL A1

KERNEL A2

KERNEL A3

KERNEL B1

More on Fermi Concurrent Kernels

- Kernels may be executed concurrently if they are issued into different streams
- Scheduling:
  - Kernels are executed in the order in which they were issued
  - Threadblocks for a given kernel are scheduled if all threadblocks for preceding kernels have been scheduled and there still are SM resources available

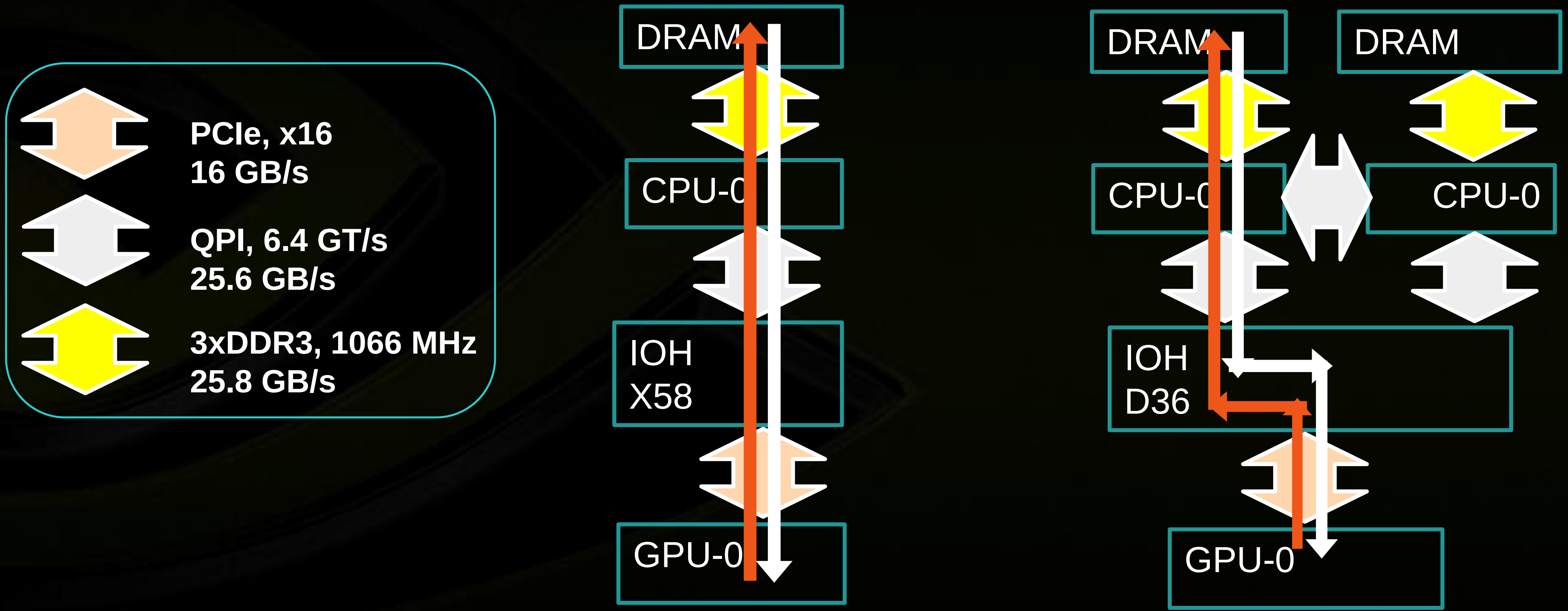
More on Fermi Dual Copy

- Fermi is capable of duplex communication with the host
  - PCIe bus is duplex
  - The two memcopies must be in different streams, different directions
- Not all current host systems can saturate duplex PCIe bandwidth:
  - Likely issues with IOH chips
  - If this is important to you, test your host system

Duplex Copy: Experimental Results

10.8 GB/s

7.5 GB/s



PCIe, x16
16 GB/s

QPI, 6.4 GT/s
25.6 GB/s

3xDDR3, 1066 MHz
25.8 GB/s

DRAM

CPU-0

IOH
X58

GPU-0

DRAM

DRAM

CPU-0

CPU-0

IOH
D36

GPU-0

Duplex Copy: Experimental Results

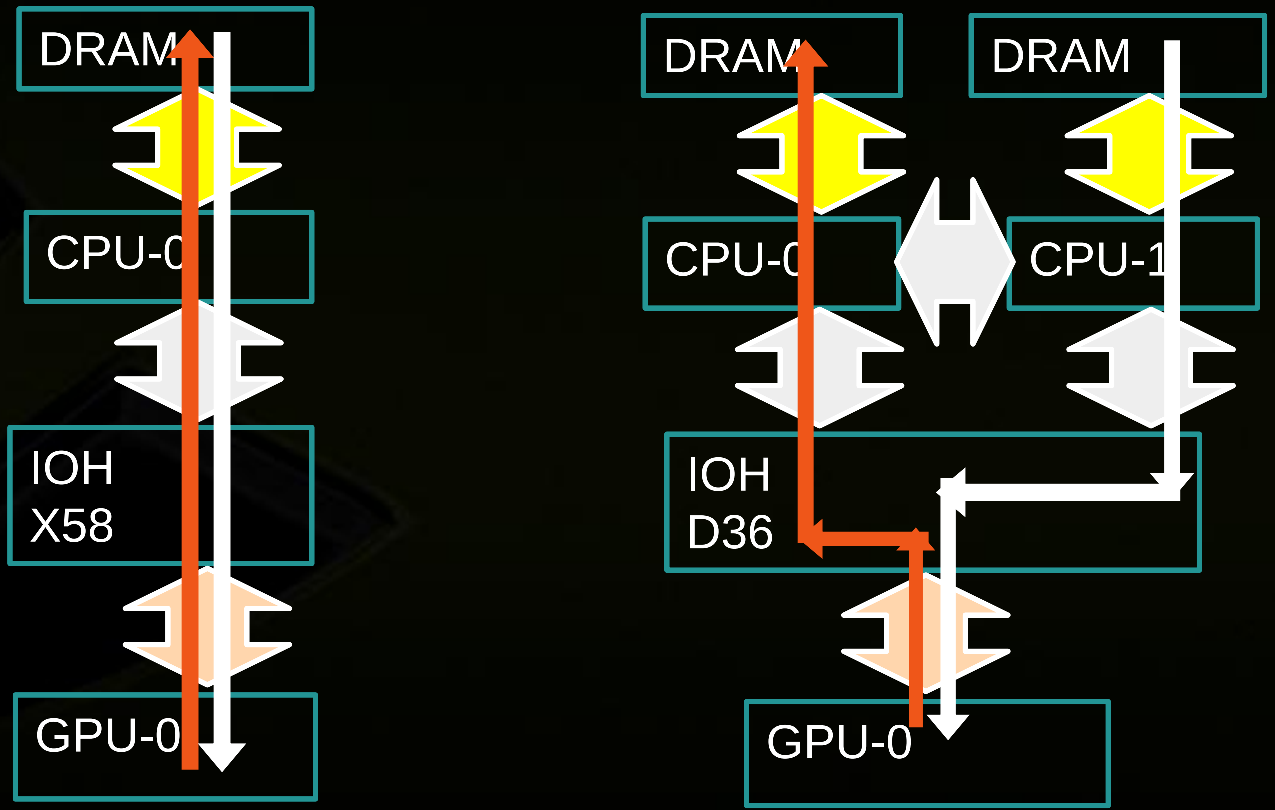
PCIe, x16
16 GB/s

QPI, 6.4 GT/s
25.6 GB/s

3xDDR3, 1066 MHz
25.8 GB/s

10.8 GB/s

11 GB/s



Copy-Compute Overlap with Zero-Copy

- Pinned host memory can be mapped into GPU memory space
  - Read/Write via GPU translates to PCIe traffic
  - Increased latency can still be hidden if enough threads available
  - Also cached on Fermi
- Finer grained overlap
  - Overlap on the warp level
  - Coalesced memory access is even more critical!
- Good for small amounts of memory or read-once bulk data

Zero-Copy Example

- Use-case: iterative solver on the GPU
  - CPU launches kernel to advance the solution
  - CPU launches second kernel to check convergence
    - GPU writes result to mapped host memory directly
  - CPU has to sync with GPU before checking the criterion and possibly triggering next iteration

Iteration Overhead	Linux	Vista (WDDM)
No sync (Reference)	1.8 us	2.3 us
Sync'd back to back	27.0 us	83.8 us
Sync'd pinned back to back	19.3 us	82.5 us
Async copy + event sync	23.3 us	85.4 us
Zero copy + event sync	16.9 us	53.3 us

CUDA 3.2

LATEST FEATURES

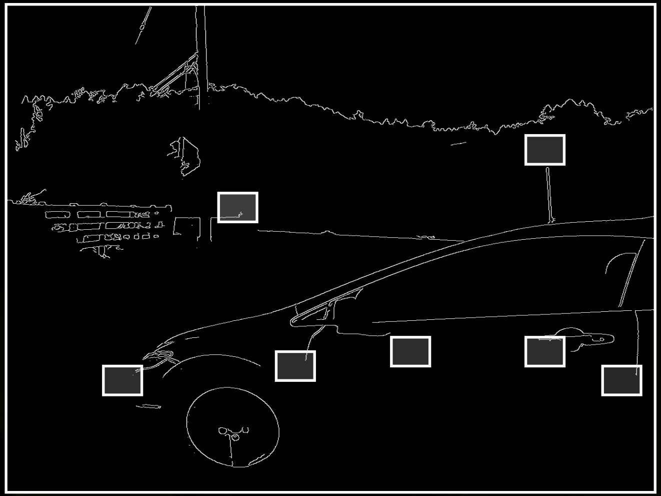
Fermi Host-Side printf

- **Printf – maybe most used debugging “tool”**

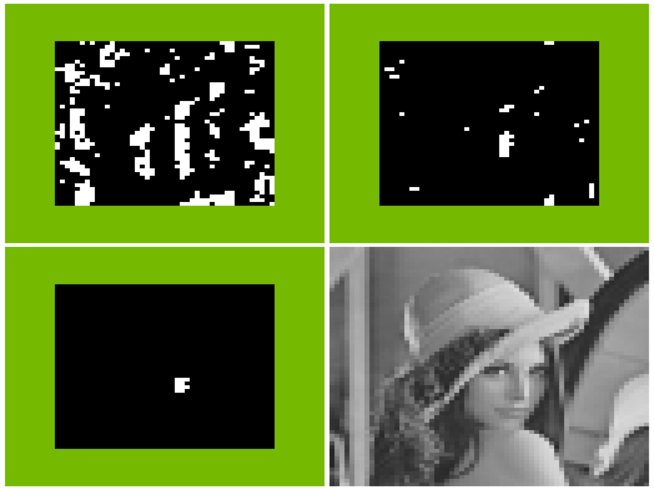
```
#include <stdio.h>   // Standard C printf
__global__
void helloWorld()
{
   int tid = blockIdx.x * blockDim.x + threadIdx.x;
   printf(“Thread %d: Hello, World!\n”, tid);  // GPU calls CPU function
}
helloWorld<<<1,3>>>();

Output:
Thread 0: Hello, World!
Thread 1: Hello, World!
Thread 2: Hello, World!
```

Varying workloads



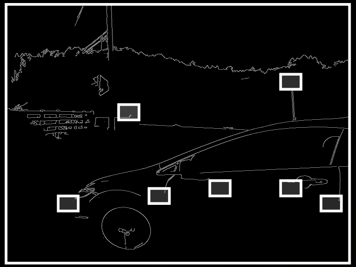
Canny Edge Detection



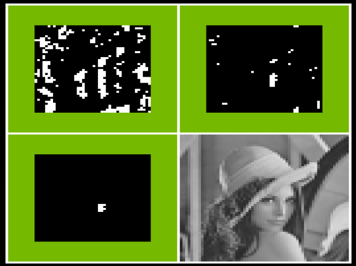
Haar Classfiers
(Face Detection)

- Many computer vision techniques have varying workloads as the algorithm progresses
  - Canny: Hysteresis only necessary in some regions
  - Haar Classifiers: Later stage classifiers only necessary in some regions

Varying workloads



Canny Edge Detection



Haar Classfiers
(Face Detection)

- How to determine which blocks remain active between kernels, how many active pixels?
  - Shared memory tricks / atomics
  - CUDA 3.1:
    - int __syncthreads_{X}(int predicate);
      - **int __syncthreads_count(int predicate);**
      - **int __syncthreads_and(int predicate);**
      - **int __syncthreads_or(int predicate);**

Code Example

```
__global__ void shmem_sync_Kernel(float* data, int*g_out)
{
   __shared__ int flag;
   if( threadIdx.x == 0 && threadIdx.y == 0) flag = 0 ;
   __syncthreads();

   if( data[someLocation] != 0 ) {
         flag = 1;
   }
   __syncthreads();
   if( flag == 1 && threadIdx.x == 0 && threadIdx.y == 0 ) {
         g_out[blockIdx.x + blockDim.x*blockIdx.y] = 1;
   }

}
```

```
__global__ void sync_predicate_Kernel(int width, int height,
int *g_out)
{

   if( __syncthreads_or( data[someLocation] != 0 )  )
   {
      if( threadIdx.x == 0 && threadIdx.y == 0 ) {
         g_out[blockIdx.x + blockIdx.y * gridDim.x ] = 1;
      }
   }


}
```

Device Side malloc/free

```
__global__ void mallocTest()
{
        char* ptr = (char*)malloc(123);
        printf(“Thread %d got pointer: %p\n”, threadIdx.x, ptr);
        free(ptr);}
        void main()
{
```

will output:

```
Thread 0 got pointer: 00057020
Thread 1 got pointer: 0005708c
Thread 2 got pointer: 000570f8
Thread 3 got pointer: 00057164
Thread 4 got pointer: 000571d0
```

Resources

TOOLS & LIBRARIES

NVIDIA Parallel Developer Program

All GPGPU developers should become NVIDIA Registered Developers

Benefits include:

- **Early Access to Pre-Release Software**
  - **Beta software and libraries**
- **Submit & Track Issues and Bugs**
- **Announcing new benefits**
  - **Exclusive Q&A Webinars with NVIDIA Engineering**
  - **Exclusive deep dive CUDA training webinars**
  - **In depth engineering presentations on beta software**

Sign up Now: www.nvidia.com/ParallelDeveloper

GPU Technology Conference

- Visit www.nvidia.com/gtc
- Recordings and downloadable files
  - Keynotes
  - All 280 Talks
  - Posters
  - Program guide
- Next GTC: October 2011

I was truly amazed by the quality of the speakers and the presentations - and the strong focus on applications. The information gathered at this event was enlightening and useful. Absolutely one of the best - and most important conferences in the technology and advanced computing sector.

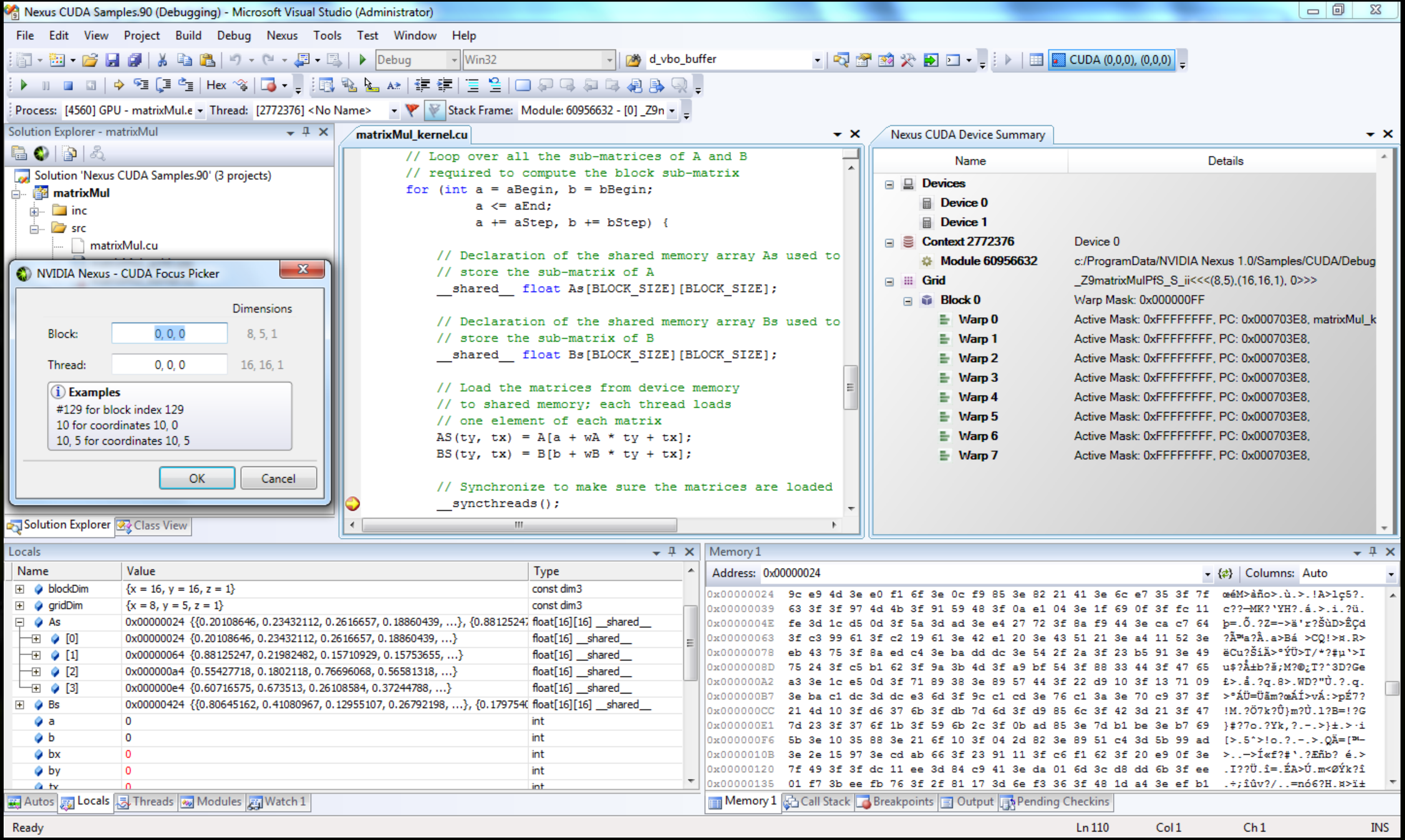
-Mike Bernhardt
-CEO & Sr. Investigative Journalist
The Exascale Report

NVIDIA Parallel Nsight 1.5

- Complete Debugger & Profiler
- Windows Vista and 7
- Remote and local debugging
- Integrated in Visual Studio
- Debugging:
  - CUDA C/C++, Direct Compute, HLSL
- Memchecker:
  - CUDA C/C++
- Profiling (Professional Version):
  - CUDA C/C++, OpenCL, Direct Compute

Microsoft® Visual Studio® PARTNER

NVIDIA Parallel Nsight - Debugging



Nexus CUDA Samples.90 (Debugging) - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Nexus Tools Test Window Help
Debug
Win32
d_vbo_buffer
CUDA (0,0,0), (0,0,0)
Hex
Solution Explorer - matrixMul
Solution 'Nexus CUDA Samples.90' (3 projects)
matrixMul
inc
src
matrixMul.cu
Solution Explorer
Class View
matrixMul_kernel.cu
Nexus CUDA Device Summary
Locals
Autos
Locals
Threads
Modules
Watch 1
Memory 1
Call Stack
Breakpoints
Output
Pending Checkins
Ready
Ln 110
Col 1
Ch 1
INS

Process: [4560] GPU - matrixMul.e Thread: [2772376] <No Name> Stack Frame: Module: 60956632 - [0] _Z9n

NVIDIA Nexus - CUDA Focus Picker

		Dimensions
Block:	0, 0, 0	8, 5, 1
Thread:	0, 0, 0	16, 16, 1

Examples

#129 for block index 129
10 for coordinates 10, 0
10, 5 for coordinates 10, 5

OK | Cancel

```
        // Loop over all the sub-matrices of A and B
        // required to compute the block sub-matrix
        for (int a = aBegin, b = bBegin;
                 a <= aEnd;
                 a += aStep, b += bStep) {

            // Declaration of the shared memory array As used to
            // store the sub-matrix of A
            __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

            // Declaration of the shared memory array Bs used to
            // store the sub-matrix of B
            __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

            // Load the matrices from device memory
            // to shared memory; each thread loads
            // one element of each matrix
            AS(ty, tx) = A[a + wA * ty + tx];
            BS(ty, tx) = B[b + wB * ty + tx];

            // Synchronize to make sure the matrices are loaded
            __syncthreads();
```

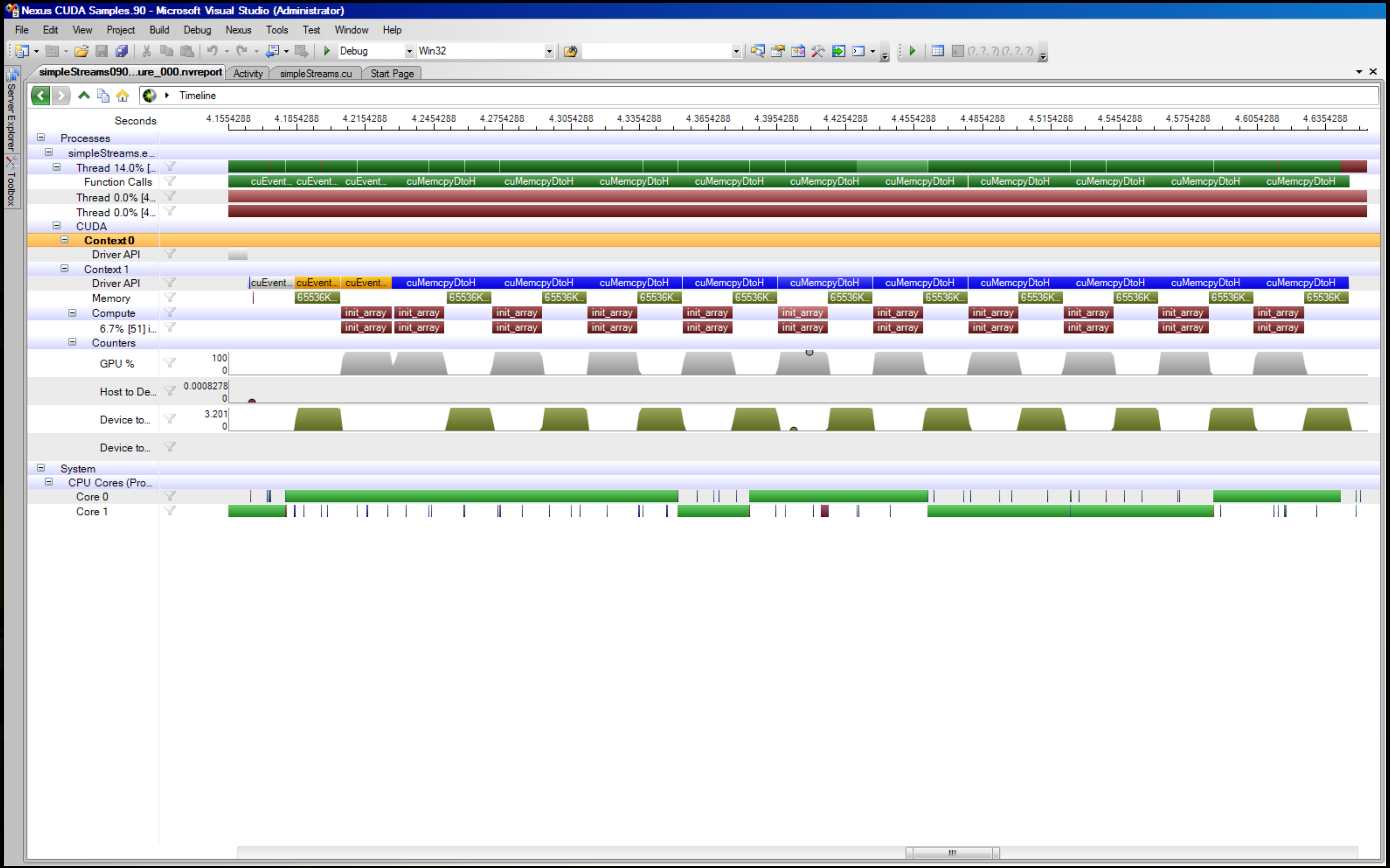
Name	Details
Devices	
Device 0	
Device 1	
Context 2772376	Device 0
Module 60956632	c:/ProgramData/NVIDIA Nexus 1.0/Samples/CUDA/Debug
Grid	_Z9matrixMulPfS_S_ii<<<(8,5),(16,16,1), 0>>>
Block 0	Warp Mask: 0x000000FF
Warp 0	Active Mask: 0xFFFFFFFF, PC: 0x000703E8, matrixMul_k
Warp 1	Active Mask: 0xFFFFFFFF, PC: 0x000703E8,
Warp 2	Active Mask: 0xFFFFFFFF, PC: 0x000703E8,
Warp 3	Active Mask: 0xFFFFFFFF, PC: 0x000703E8,
Warp 4	Active Mask: 0xFFFFFFFF, PC: 0x000703E8,
Warp 5	Active Mask: 0xFFFFFFFF, PC: 0x000703E8,
Warp 6	Active Mask: 0xFFFFFFFF, PC: 0x000703E8,
Warp 7	Active Mask: 0xFFFFFFFF, PC: 0x000703E8,

Name	Value	Type
blockDim	{x = 16, y = 16, z = 1}	const dim3
gridDim	{x = 8, y = 5, z = 1}	const dim3
As	0x00000024 {{0.20108646, 0.23432112, 0.2616657, 0.18860439, ...}, {0.8812524	float[16][16] __shared__
[0]	0x00000024 {0.20108646, 0.23432112, 0.2616657, 0.18860439, ...}	float[16] __shared__
[1]	0x00000064 {0.88125247, 0.21982482, 0.15710929, 0.15753655, ...}	float[16] __shared__
[2]	0x000000a4 {0.55427718, 0.1802118, 0.76696068, 0.56581318, ...}	float[16] __shared__
[3]	0x000000e4 {0.60716575, 0.673513, 0.26108584, 0.37244788, ...}	float[16] __shared__
Bs	0x00000424 {{0.80645162, 0.41080967, 0.12955107, 0.26792198, ...}, {0.179754	float[16][16] __shared__
a	0	int
b	0	int
bx	0	int
by	0	int
tx	0	int

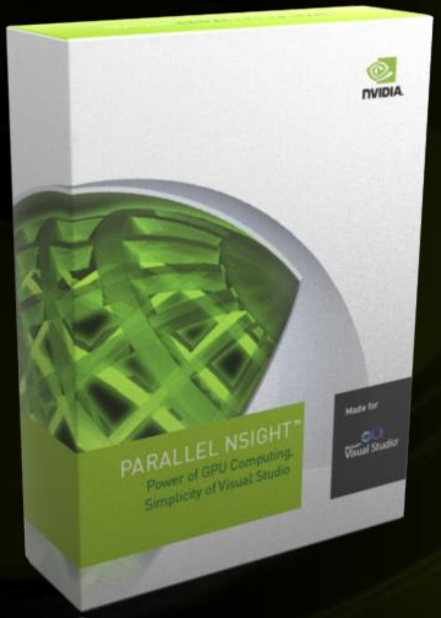
Address: 0x00000024 Columns: Auto

```
0x00000024  9c e9 4d 3e e0 f1 6f 3e 0c f9 85 3e 82 21 41 3e 6c e7 35 3f 7f  œéM>àño>.ù.>.!A>lç5?.
0x00000039  63 3f 3f 97 4d 4b 3f 91 59 48 3f 0a e1 04 3e 1f 69 0f 3f fc 11  c??—MK?'YH?.á.>.i.?ü.
0x0000004E  fe 3d 1c d5 0d 3f 5a 3d ad 3e e4 27 72 3f 8a f9 44 3e ca c7 64  þ=.Õ.?Z=->ä'r?ŠùD>ÊÇd
0x00000063  3f c3 99 61 3f c2 19 61 3e 42 e1 20 3e 43 51 21 3e a4 11 52 3e  ?Ã™a?Â.a>Bá >CQ!>¤.R>
0x00000078  eb 43 75 3f 8a ed c4 3e ba dd dc 3e 54 2f 2a 3f 23 b5 91 3e 49  ëCu?ŠíÄ>ºÝÜ>T/*?#µ'>I
0x0000008D  75 24 3f c5 b1 62 3f 9a 3b 4d 3f a9 bf 54 3f 88 33 44 3f 47 65  u$?Å±b?š;M?©¿T?ˆ3D?Ge
0x000000A2  a3 3e 1c e5 0d 3f 71 89 38 3e 89 57 44 3f 22 d9 10 3f 13 71 09  £>.å.?q‰8>‰WD?"Ù.?.q.
0x000000B7  3e ba c1 dc 3d dc e3 6d 3f 9c c1 cd 3e 76 c1 3a 3e 70 c9 37 3f  >ºÁÜ=Üãm?œÁÍ>vÁ:>pÉ7?
0x000000CC  21 4d 10 3f d6 37 6b 3f db 7d 6d 3f d9 85 6c 3f 42 3d 21 3f 47  !M.?Ö7k?Û}m?Ù…l?B=!?G
0x000000E1  7d 23 3f 37 6f 1b 3f 59 6b 2c 3f 0b ad 85 3e 7d b1 be 3e b7 69  }#?7o.?Yk,?.-…>}±¾>·i
0x000000F6  5b 3e 10 35 88 3e 21 6f 10 3f 04 2d 82 3e 89 51 c4 3d 5b 99 ad  [>.5ˆ>!o.?.-‚>‰QÄ=[™-
0x0000010B  3e 2e 15 97 3e cd ab 66 3f 23 91 11 3f c6 f1 62 3f 20 e9 0f 3e  >..—>Í«f?#'.?Æñb? é.>
0x00000120  7f 49 3f 3f dc 11 ee 3d 84 c9 41 3e da 01 6d 3c d8 dd 6b 3f ee  .I??Ü.î=„ÉA>Ú.m<ØÝk?î
0x00000135  01 f7 3b ee fb 76 3f 2f 81 17 3d 6e f3 36 3f 48 1d a4 3e ef b1  .÷;îûv?/..=nó6?H.¤>ï±
```

NVIDIA Parallel Nsight - Profiling

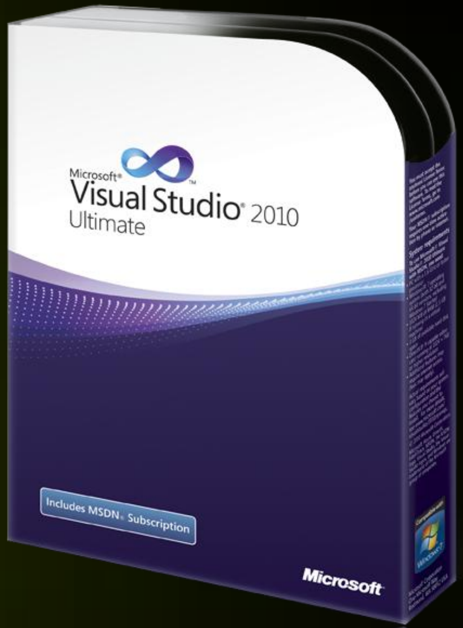


NVIDIA Parallel Nsight™ 1.51



- All Professional features now available free of charge!

- Key Features
  - Microsoft Visual Studio 2010
  - Single System Debugging
  - Tesla Compute Cluster
  - CUDA Toolkit 3.2



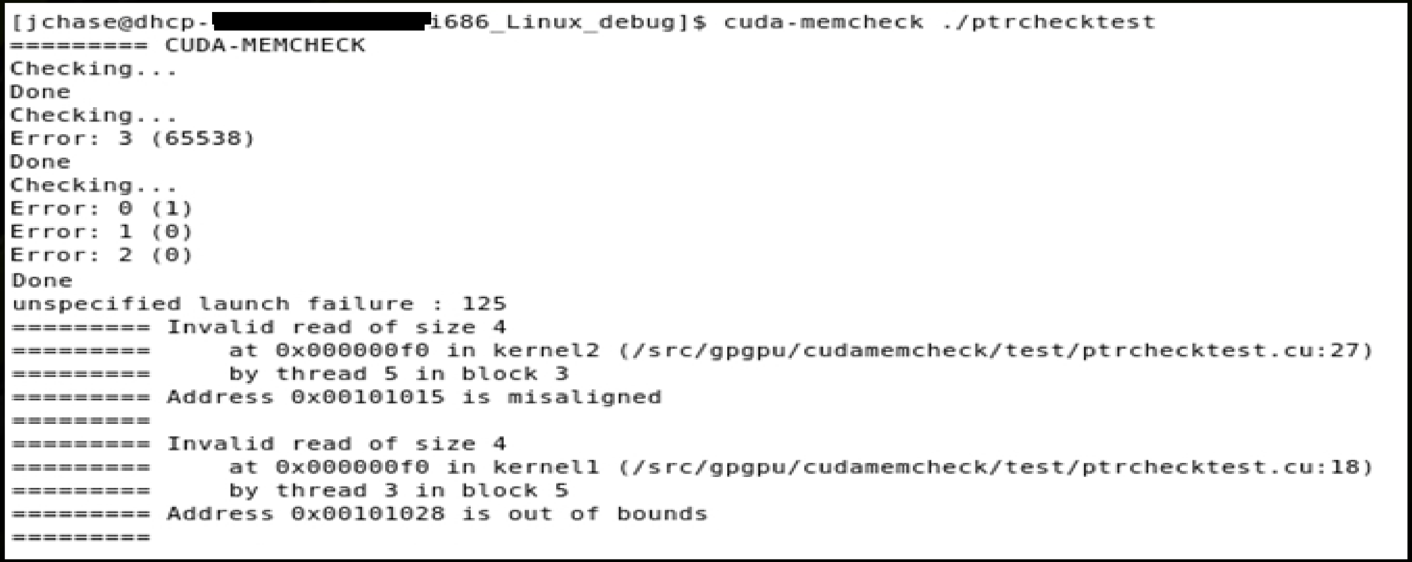
http://www.nvidia.com/GetParallelNsight

CUDA-GDB

- Extended gdb – useable via ddd
- Debug GPUs not in use by X
  - Compute Capability >= 1.1
- Features
  - Debug Host und Device code
  - Breakpoints at functions or line#
  - Single-step on the warp level
  - Change CUDA Threads or Blocks
  - Display Device Memory

CUDA-MemCheck

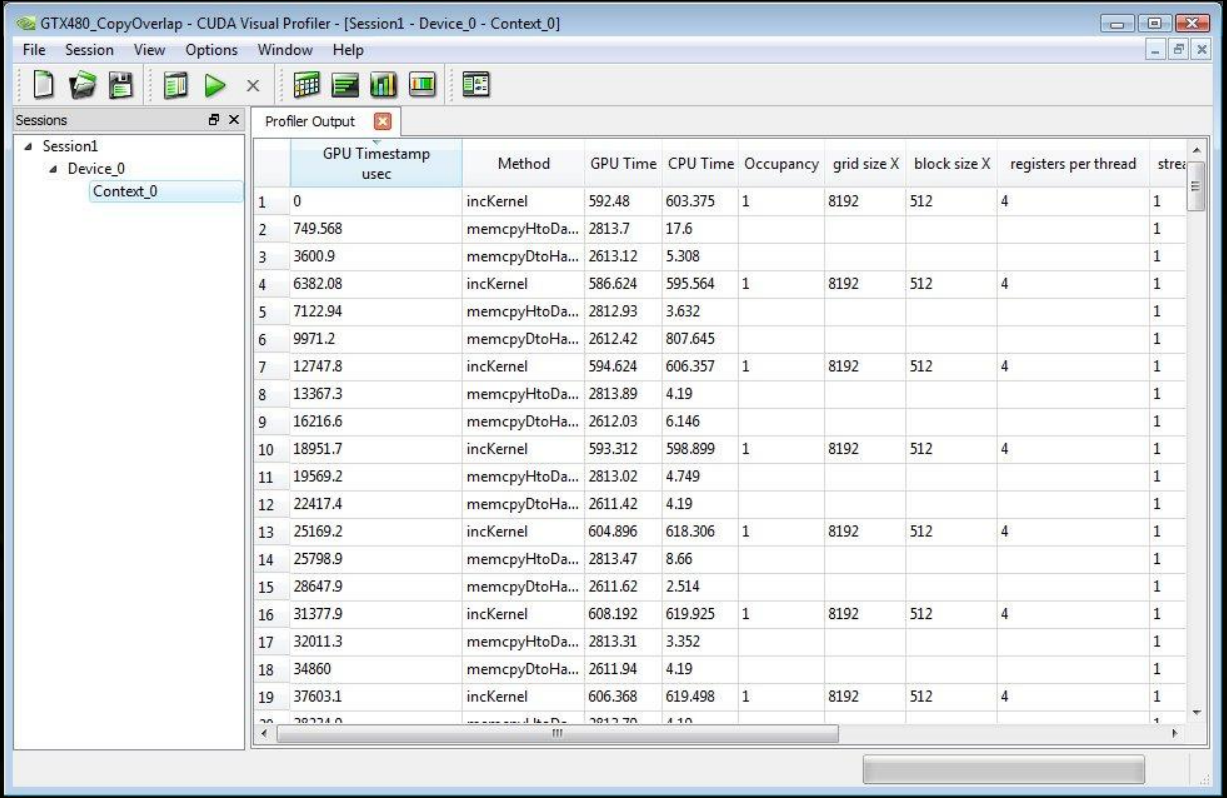
- Discovers memory access errors
  - Out of bounds access
  - Misaligned access
- Integrated in CUDA-GDB and Parallel Nsight
- Standalone available for Linux and Windows XP



```
[jchase@dhcp-[redacted]i686_Linux_debug]$ cuda-memcheck ./ptrchecktest
========= CUDA-MEMCHECK
Checking...
Done
Checking...
Error: 3 (65538)
Done
Checking...
Error: 0 (1)
Error: 1 (0)
Error: 2 (0)
Done
unspecified launch failure : 125
========= Invalid read of size 4
=========     at 0x000000f0 in kernel2 (/src/gpgpu/cudamemcheck/test/ptrchecktest.cu:27)
=========     by thread 5 in block 3
========= Address 0x00101015 is misaligned
=========
========= Invalid read of size 4
=========     at 0x000000f0 in kernel1 (/src/gpgpu/cudamemcheck/test/ptrchecktest.cu:18)
=========     by thread 3 in block 5
========= Address 0x00101028 is out of bounds
=========
```

Visual Profiler

- CUDA & OpenCL
- Performance Analysis with HW-Counters
- Works directly with app
  - No special built necessary



GTX480_CopyOverlap - CUDA Visual Profiler - [Session1 - Device_0 - Context_0]

	GPU Timestamp usec	Method	GPU Time	CPU Time	Occupancy	grid size X	block size X	registers per thread	strea
1	0	incKernel	592.48	603.375	1	8192	512	4	1
2	749.568	memcpyHtoDa...	2813.7	17.6					1
3	3600.9	memcpyDtoHa...	2613.12	5.308					1
4	6382.08	incKernel	586.624	595.564	1	8192	512	4	1
5	7122.94	memcpyHtoDa...	2812.93	3.632					1
6	9971.2	memcpyDtoHa...	2612.42	807.645					1
7	12747.8	incKernel	594.624	606.357	1	8192	512	4	1
8	13367.3	memcpyHtoDa...	2813.89	4.19					1
9	16216.6	memcpyDtoHa...	2612.03	6.146					1
10	18951.7	incKernel	593.312	598.899	1	8192	512	4	1
11	19569.2	memcpyHtoDa...	2813.02	4.749					1
12	22417.4	memcpyDtoHa...	2611.42	4.19					1
13	25169.2	incKernel	604.896	618.306	1	8192	512	4	1
14	25798.9	memcpyHtoDa...	2813.47	8.66					1
15	28647.9	memcpyDtoHa...	2611.62	2.514					1
16	31377.9	incKernel	608.192	619.925	1	8192	512	4	1
17	32011.3	memcpyHtoDa...	2813.31	3.352					1
18	34860	memcpyDtoHa...	2611.94	4.19					1
19	37603.1	incKernel	606.368	619.498	1	8192	512	4	1

Libraries

- NVIDIA

cuBLAS	Dense linear algebra (subset of full BLAS suite)
cuFFT	1D/2D/3D real and complex
cuSparse	Sparse linear algebra
cuRand	RNGs
NPP	Performance primitives

- Open Source from NV employees

Thrust	STL/Boost style template language
cuDPP	Data parallel primitives (e.g. scan, sort and reduction)
opencurrent	PDEs

CUDA Math Libraries

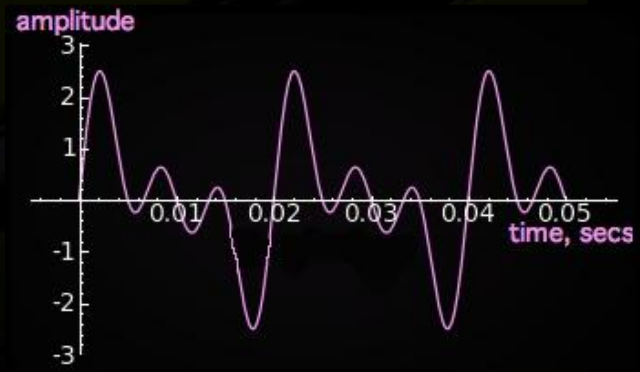
High performance math routines for your applications:

- cuFFT – Fast Fourier Transforms Library
- cuBLAS – Complete BLAS Library
- cuSPARSE – Sparse Matrix Library
- cuRAND – Random Number Generation (RNG) Library

- Included in the CUDA Toolkit (free download)
  - www.nvidia.com/getcuda
- For more information on CUDA libraries:
  - http://www.nvidia.com/object/gtc2010-presentation-archive.html#session2216

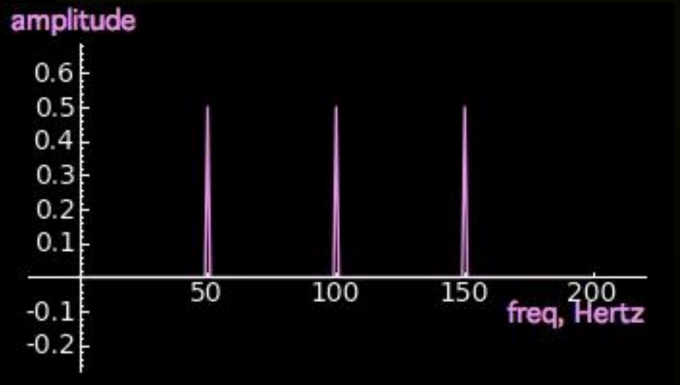
cuFFT

- Multi-dimensional Fast Fourier Transforms
- New in CUDA 3.2:
  - Higher performance of 1D, 2D, 3D transforms with dimensions of powers of 2, 3, 5 or 7
  - Higher performance and accuracy for 1D transform sizes that contain large prime factors



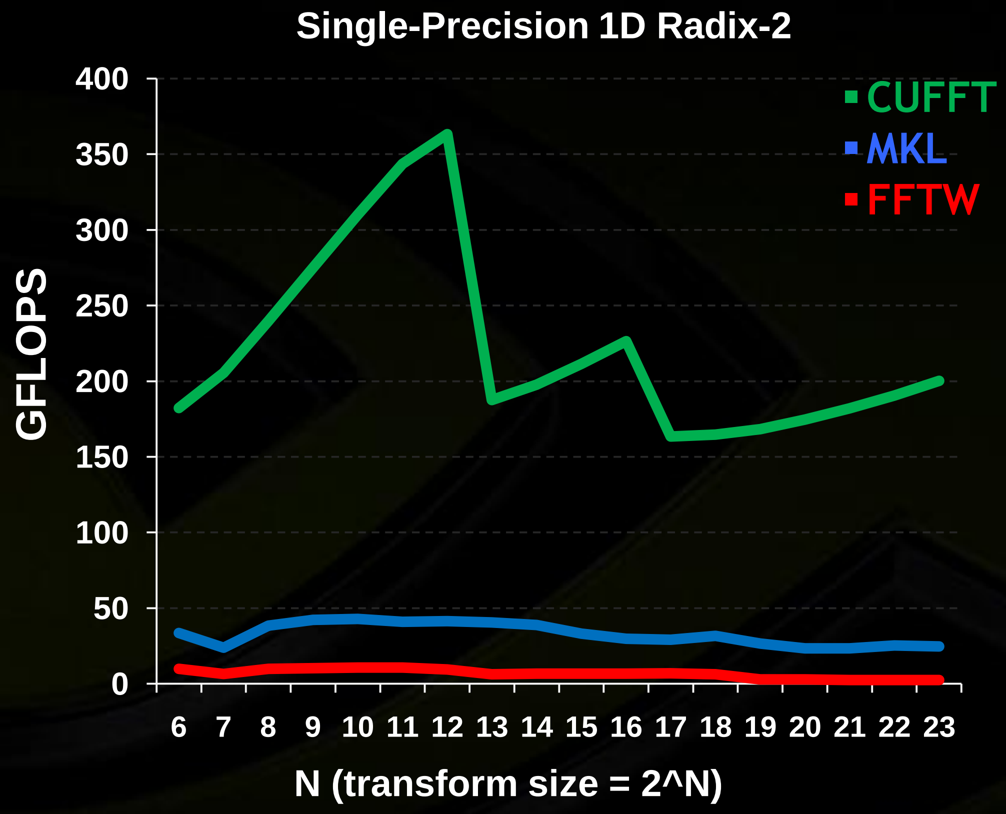
$$F(x) = \sum_{n=0}^{N-1} f(n) e^{-j2\pi(x\frac{n}{N})}$$

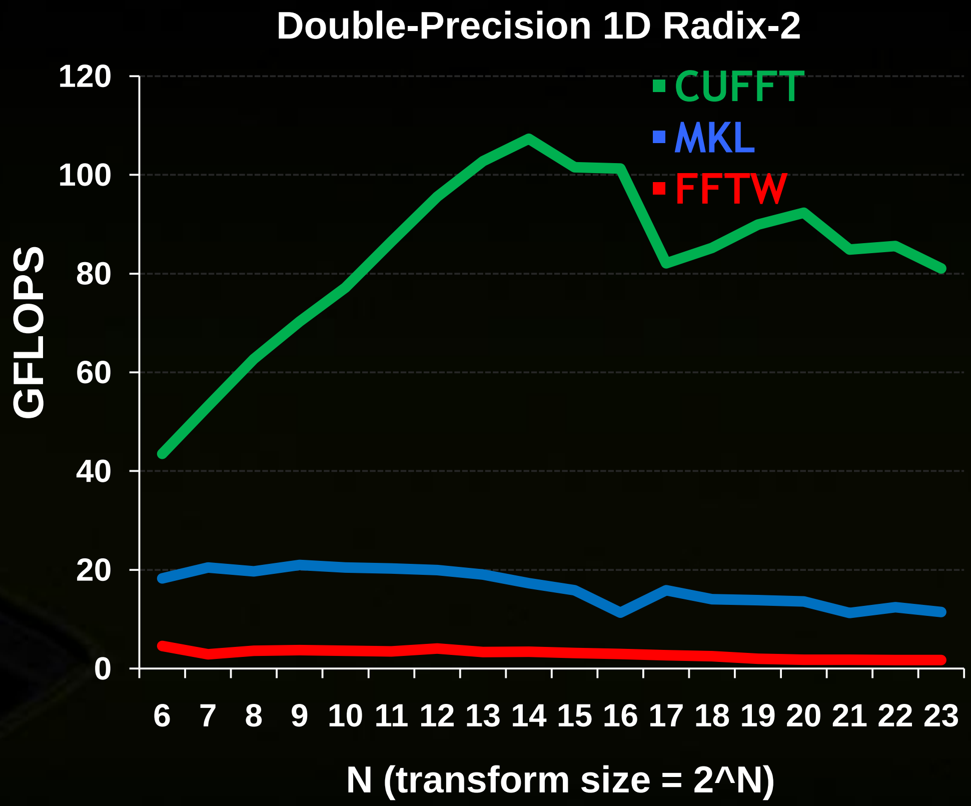
$$f(n) = \frac{1}{N} \sum_{n=0}^{N-1} F(x) e^{j2\pi(x\frac{n}{N})}$$



FFTs up to 8.8x Faster than MKL

1D used in audio processing and as a foundation for 2D and 3D FFTs





* cuFFT 3.2, Tesla C2050 (Fermi) with ECC on

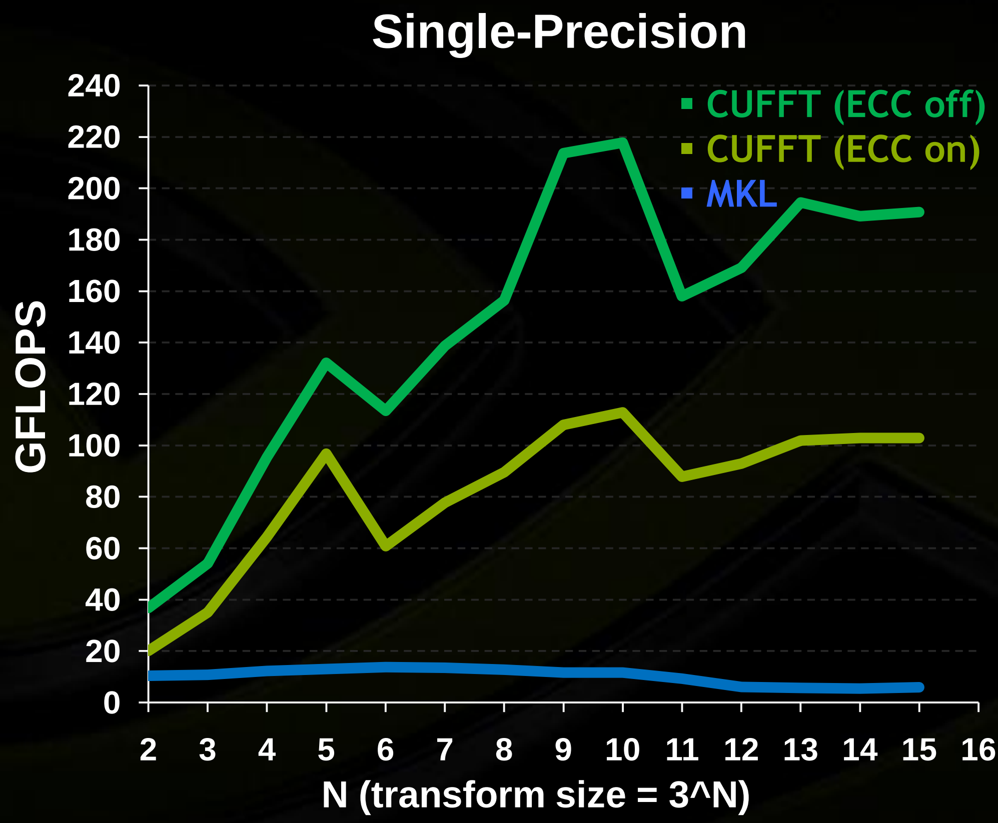
* MKL 10.1r1, 4-core Corei7 Nehalem @ 3.07GHz

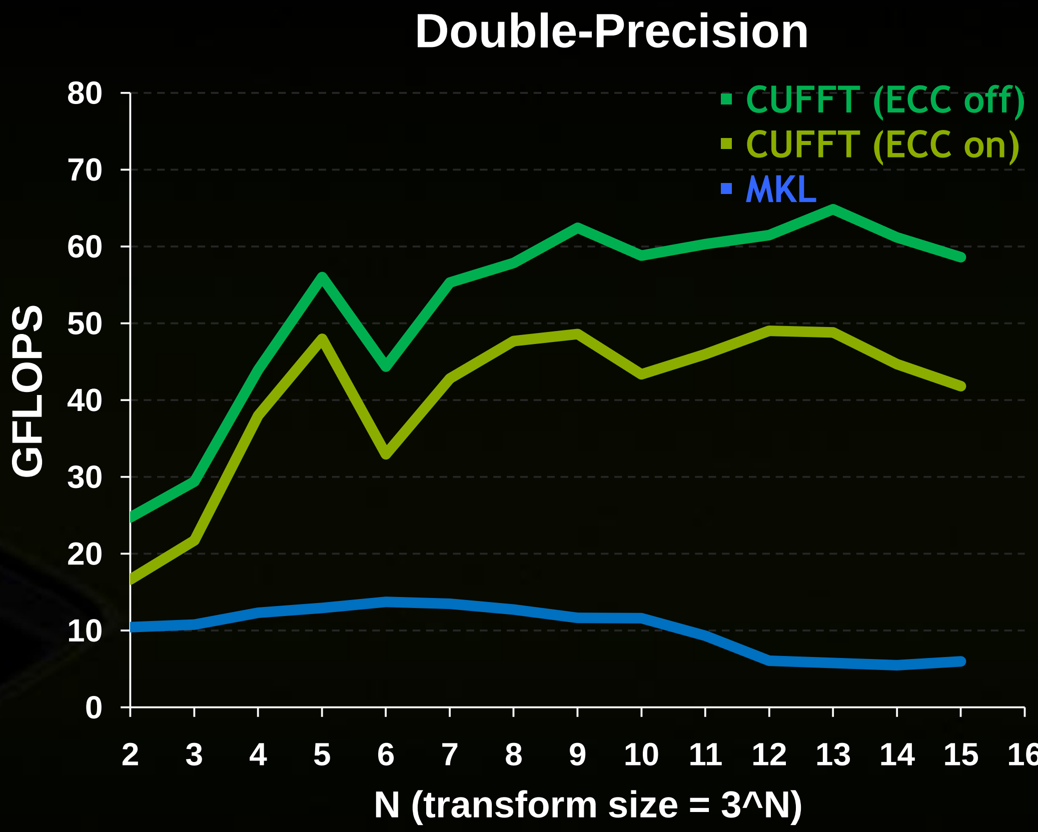
* FFTW single-thread on same CPU

Performance may vary based on OS version and motherboard configuration

cuFFT 1D Radix-3 up to 18x Faster than MKL

18x for single-precision, 15x for double-precision
Similar acceleration for radix-5 and -7





* CUFFT 3.2 on Tesla C2050 (Fermi)

* MKL 10.1r1, 4-core Corei7 Nehalem @ 3.07GHz

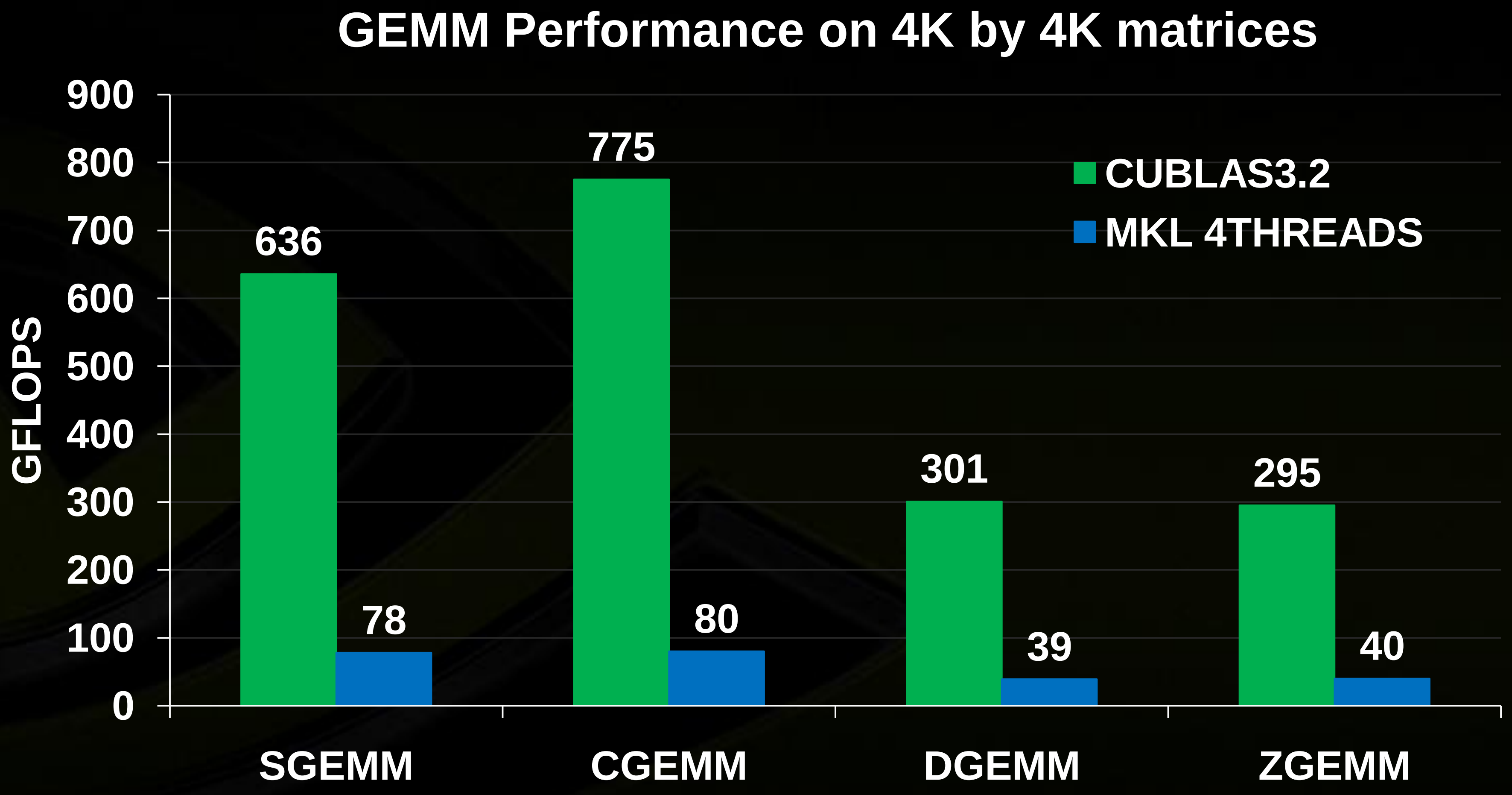
Performance may vary based on OS version and motherboard configuration

cuBLAS: Dense Linear Algebra on GPUs

- Complete BLAS implementation
  - Supports all 152 routines for single, double, complex and double complex

- New in CUDA 3.2
  - 7x Faster GEMM (matrix multiply) on Fermi GPUs
  - Higher performance on SGEMM & DGEMM for all matrix sizes and all transpose cases (NN, TT, TN, NT)

Up to 8x Speedup for all GEMM Types

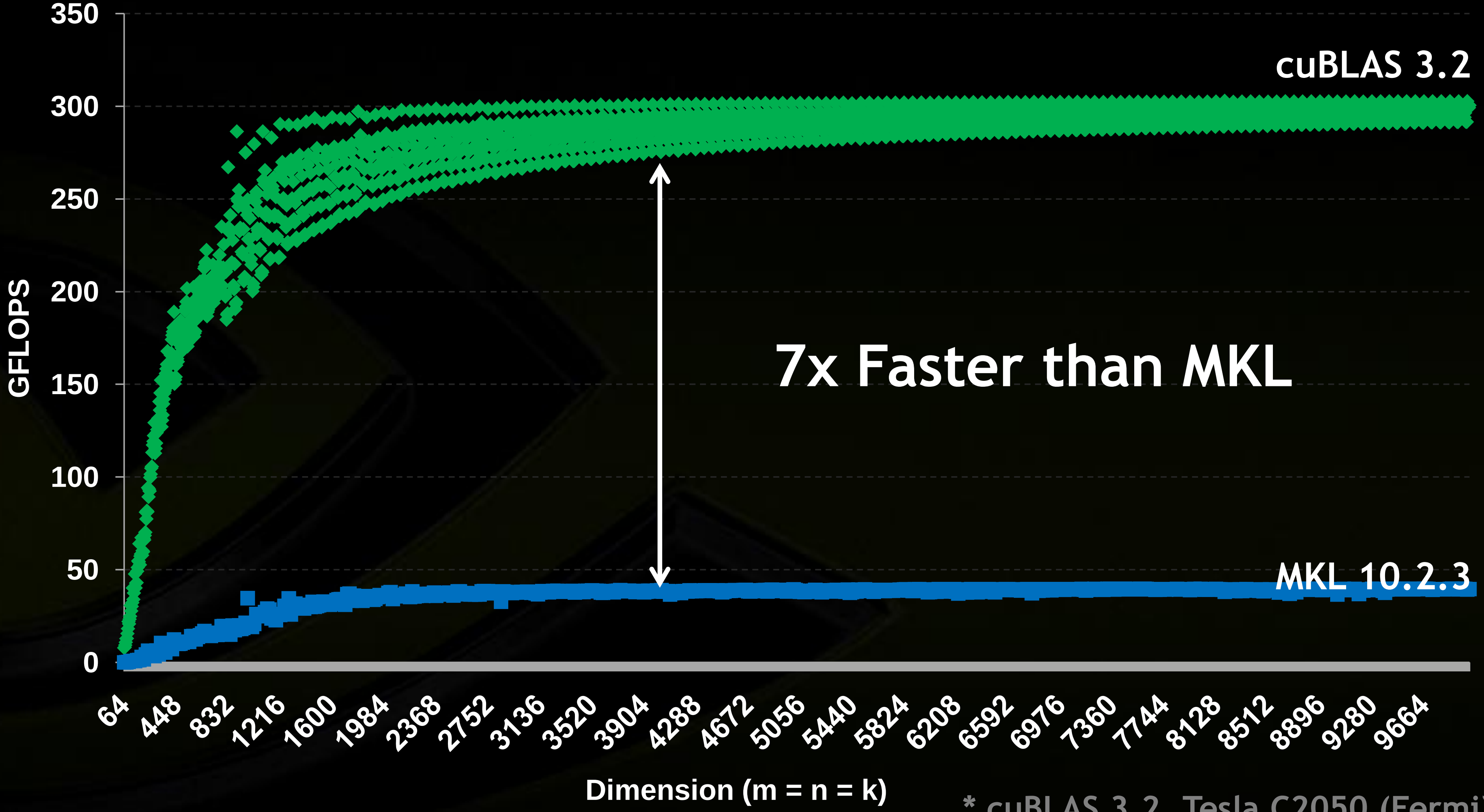


* cuBLAS 3.2, Tesla C2050 (Fermi), ECC on

* MKL 10.2.3, 4-core Corei7 @ 2.66Ghz

Performance may vary based on OS version and motherboard configuration

7x Faster DGEMM Performance



7x Faster than MKL

* cuBLAS 3.2, Tesla C2050 (Fermi), ECC on
* MKL 10.2.3, 4-core Corei7 @ 2.66Ghz

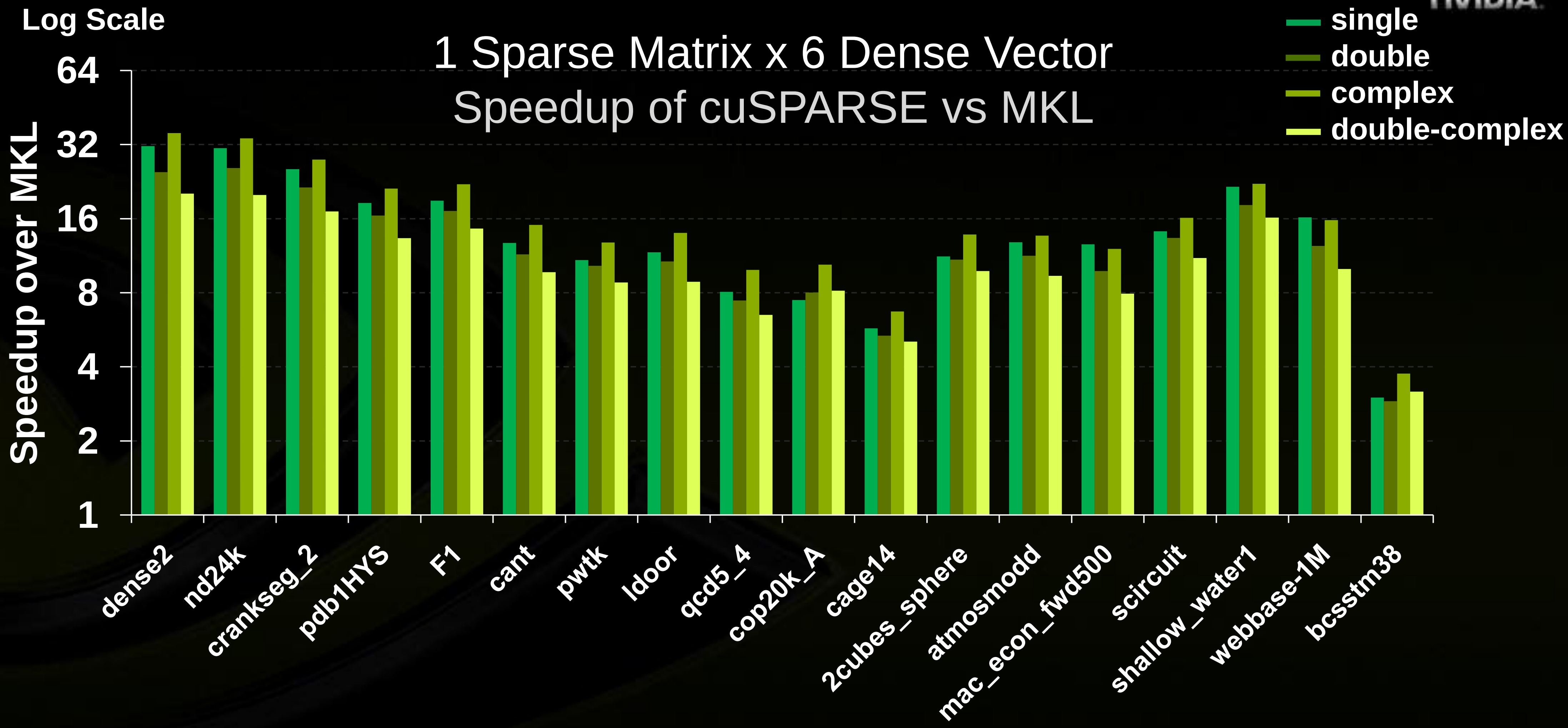
Performance may vary based on OS version and motherboard configuration

cuSPARSE

- New library for sparse linear algebra
- Conversion routines for dense, COO, CSR and CSC formats
- Optimized sparse matrix-vector multiplication

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \backslash\mathrm{alpha} \begin{bmatrix} 1.0 & & & \\ 2.0 & 3.0 & & \\ & & 4.0 & \\ 5.0 & & 6.0 & 7.0 \end{bmatrix} \begin{bmatrix} 1.0 \\ 2.0 \\ 3.0 \\ 4.0 \end{bmatrix} + \backslash\mathrm{beta} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}$$

32x Faster



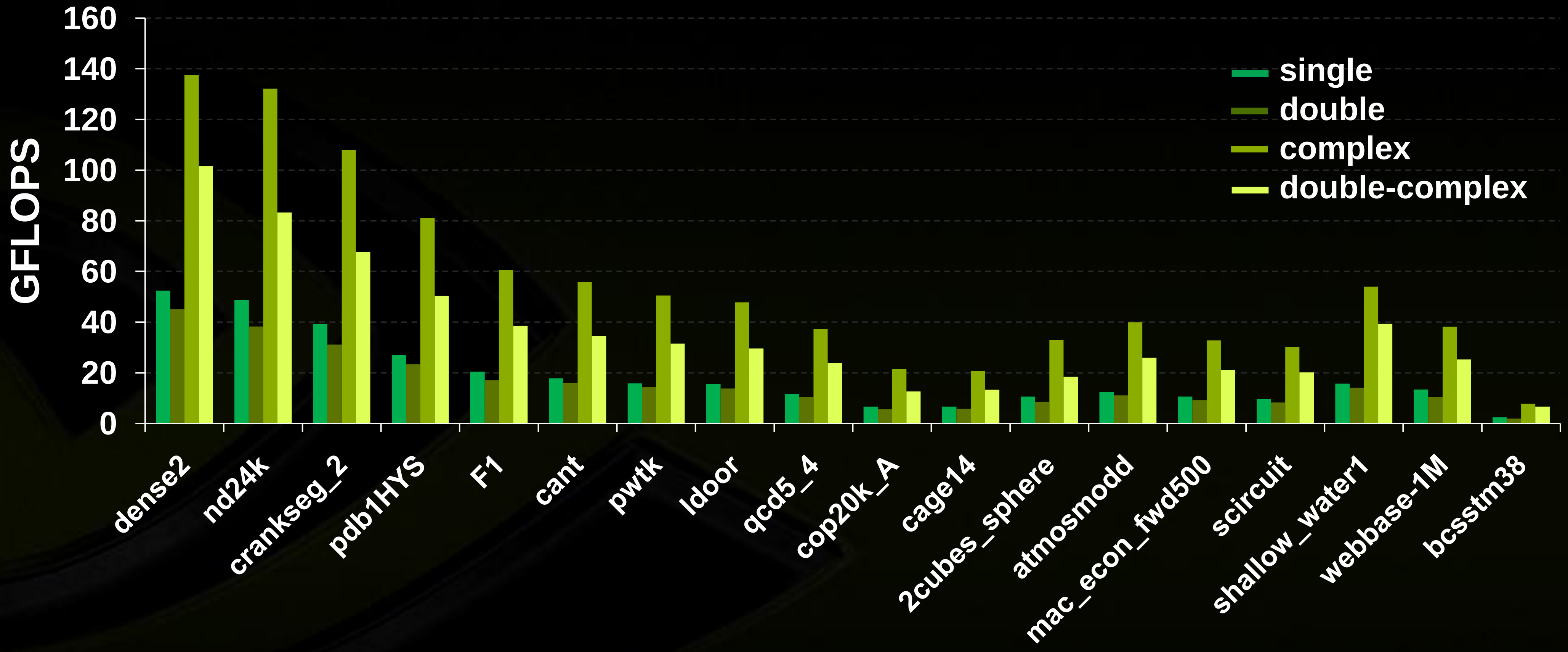
1 Sparse Matrix x 6 Dense Vector

Speedup of cuSPARSE vs MKL

* CUSPARSE 3.2, NVIDIA C2050 (Fermi), ECC on
* MKL 10.2.3, 4-core Corei7 @ 3.07GHz

Performance may vary based on OS version and motherboard configuration

Performance of 1 Sparse Matrix x 6 Dense Vectors



Test cases roughly in order of increasing sparseness

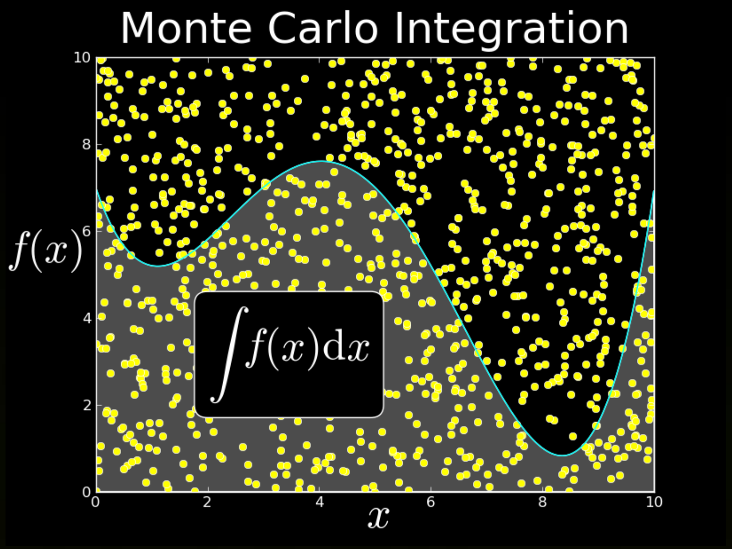
Performance may vary based on OS version and motherboard configuration

* CUSPARSE 3.2, NVIDIA C2050 (Fermi), ECC on

cuRAND

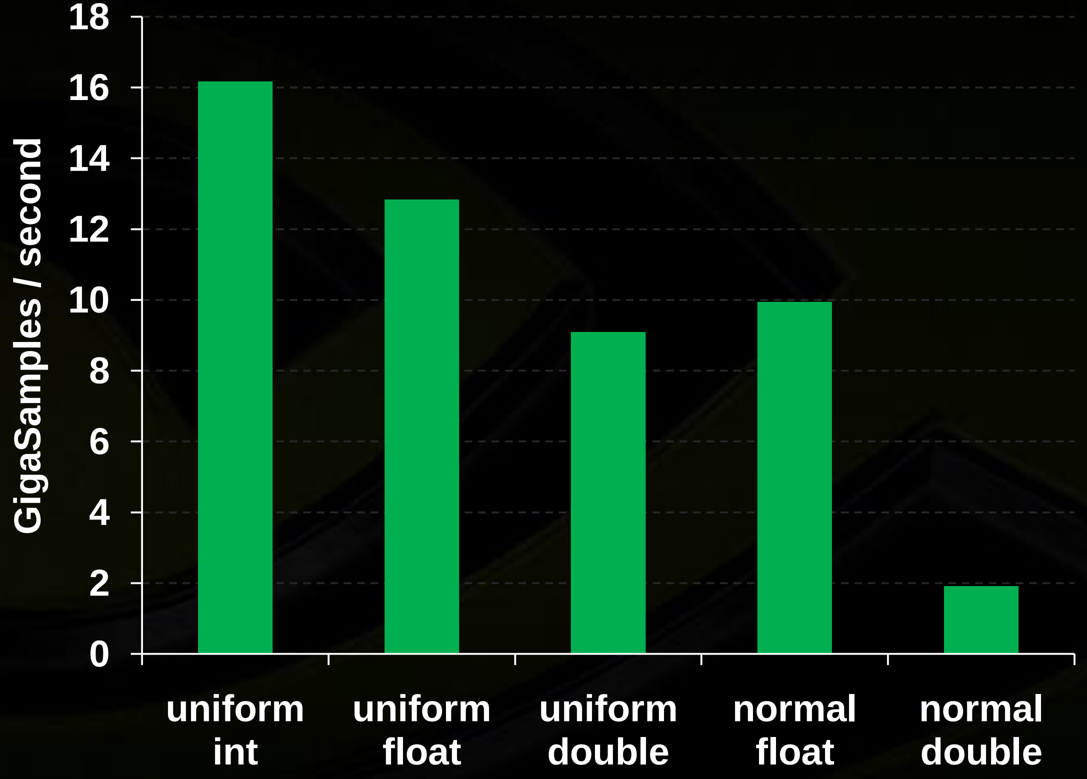
- Library for generating random numbers

- Features supported in CUDA 3.2
  - XORWOW pseudo-random generator
  - Sobol' quasi-random number generators
  - Host API for generating random numbers in bulk
  - Inline implementation allows use inside GPU functions/kernels
  - Single- and double-precision, uniform and normal distributions

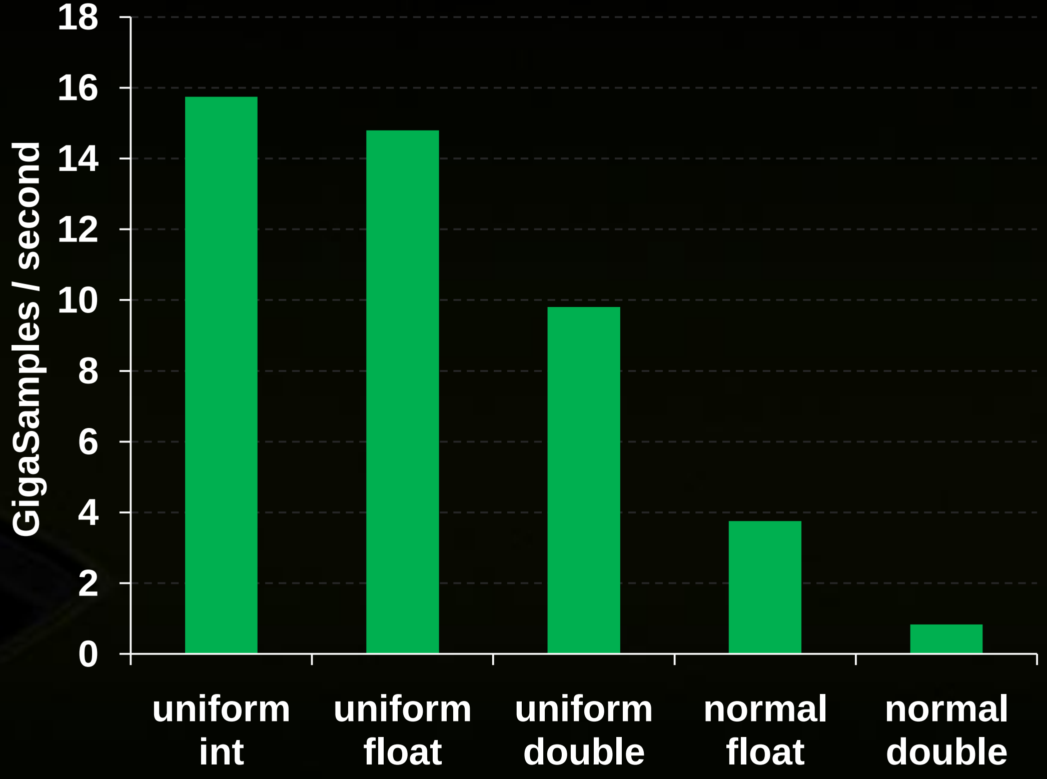


cuRAND Performance

XORWOW Psuedo-RNG



Sobol' Quasi-RNG (1 dimension)



* CURAND 3.2, NVIDIA C2050 (Fermi), ECC on

Performance may vary based on OS version and motherboard configuration