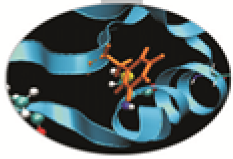
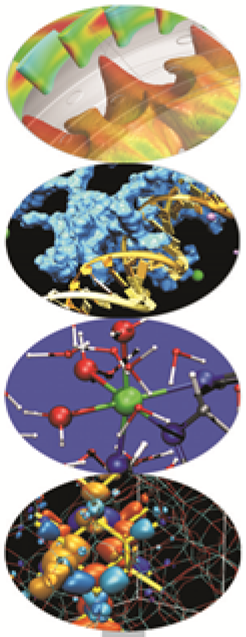
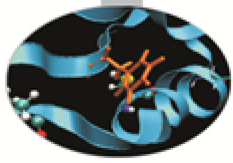


CUDA Efficient Programming



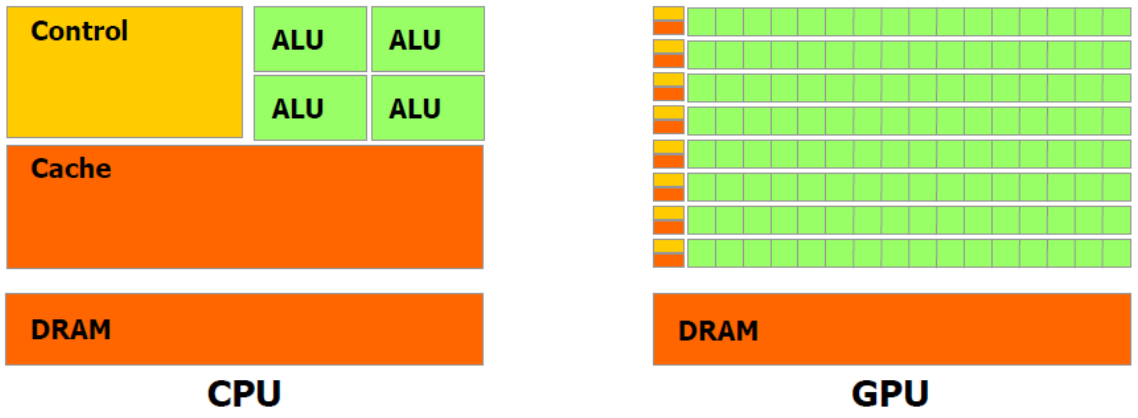


Outline

1. Overview and general concepts
2. Performance Metrics
3. Memory Optimizations
4. Execution Optimization
5. Tools Overview

Different worlds: host and device

	Host	Device
Threading resources	2 threads per core (SMT), 24/32 threads per node. The thread is the atomic execution unit.	e.g.: 1536 (thd x sm) * 14 (sm) = 21504. The Warp (32 thd) is the atomic execution unit.
Threads	«Heavy» entities, context switches and resources management.	Extremely lightweight, managed grouped into warps, fast context switch, no resources management (statically allocated once).
Memory	e.g.: 48 GB / 32 thd = 1.5 GB/thd, 300 cycles lat., 6.4 GB/s band (DDR3), 3 caching levels with lots of speculation logic.	e.g.: 6 GB / 21504 thd = 0.3 MB/thd, 600 cycles lat*, 144 GB/s band (GDDR5)*, fake caches. * coalesced



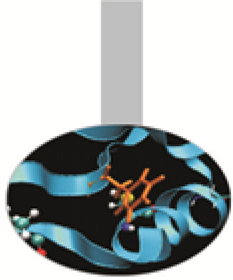
Obtaining maximum performance benefit

- Focus on achieving high occupancy (more on this later, for know you can translate «high occupancy» as «many many… many threads in flight»).
- Focus on how to exploit the SIMT (data parallel) programming model.
- Deeply analyze your algorithm in order to find hotspots and embarassingly parallel-enabled portions.

Furthermore, pay attention to the Amdahl's law:

$$S = \frac{1}{(1-P)+P/N}$$

Hint: avoid the jump-start-to-code approach: porting your serial and/or multithreaded and/or message passing CPU application to GPU is *not* in general an easy task.



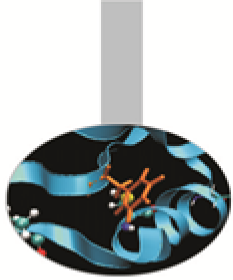
CUDA Enabled GPU: compute capability

The compute capability is a kind of version tag that identifies:

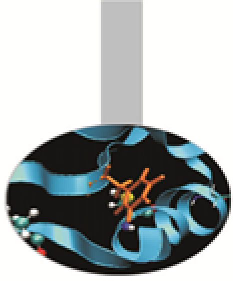
- instructions and features supported by the board;
- coalescing rules;
- the board’s resources constraints;
- throughput of some instructions (hardware implementation).

The compute capability is given as a ***major***.*dot*.***minor*** version number (i.e: 2.0, 2.1, 3.0, 3.5).

Compute capability: resources constraints



Technical Specifications	Compute Capability						
	1.0	1.1	1.2	1.3	2.x	3.0	3.5
Maximum dimensionality of grid of thread blocks	2				3		
Maximum x-dimension of a grid of thread blocks	65535					2^{31}-1	
Maximum y- or z-dimension of a grid of thread blocks	65535						
Maximum dimensionality of thread block	3						
Maximum x- or y-dimension of a block	512				1024		
Maximum z-dimension of a block	64						
Maximum number of threads per block	512				1024		
Warp size	32						
Maximum number of resident blocks per multiprocessor	8					16	
Maximum number of resident warps per multiprocessor	24		32		48	64	
Maximum number of resident threads per multiprocessor	768		1024		1536	2048	
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K	64 K	
Maximum number of 32-bit registers per thread	128				63		255
Maximum amount of shared memory per multiprocessor	16 KB				48 KB		
Number of shared memory banks	16				32		
Amount of local memory per thread	16 KB				512 KB		
Constant memory size	64 KB						
Cache working set per multiprocessor for constant memory	8 KB						
Cache working set per multiprocessor for texture memory	Device dependent, between 6 KB and 8 KB						
Maximum width for a 1D texture reference bound to a CUDA array	8192				65536		



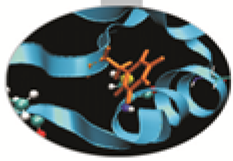
Performance metrics

Performance metrics

- Wall-clock time
  - you always want to keep that one at a minimum
- Theoretical (peak) bandwidth Vs effective bandwidth
  - that allows you to measure performance of a memory-bound kernel
- Theoretical (peak) FLOPS* Vs effective FLOPS**
  - that allows you to measure performance of a compute-bound kernel

*theoretical **FL**oating point **O**peration **P**er **S**econd: different kind of ops have in general different throughput . Ops throughput differs among the compute capabilities.

effective **FLoating point **O**peration **P**er **S**econd: can be difficult to count the effective number of operations that the kernel is doing during execution.



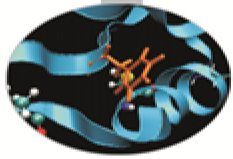
Timing

- You can use the standard timing facilities (host side) in an almost standard way…
- …but remember: CUDA calls can be asynchronous!

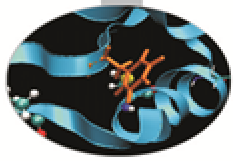
```
start = clock()
my_kernel<<< blocks, threads>>>();
cudaDeviceSynchronize();
end = clock();
```

- CUDA provides the **cudaEvents** facility. They grant you access to the GPU timer.
- Needed to time a single stream without loosing Host/Device concurrency.

```
cdaEvent_t start, stop;
cudaEventCreate(start); cudaEventCreate(stop);
cudaEventRecord(start, 0);
My_kernel<<<block2, threads>>> ();
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float ElapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
cudaEventDestroy(start); cudaEventDestroy(stop);
```



Bandwidth



1. Get GPU main memory's theoretical bandwidth (ECC off):

$$B = freq * busw * nlin = \underset{\uparrow \atop \text{clock freq.}}{(1.107\ GHz)} * \underbrace{\left(\frac{\overset{\text{bus width (bits)}}{512} * \overset{\text{lines per clock}}{2}}{8} B\right)}_{\text{bytes per clock}} = \underset{\nwarrow \text{ GeForce GTX 280 Bandwidth}}{141.6\ GB/s}$$

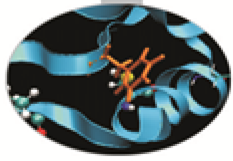
2. Get kernel's effective bandwidth:

```
// slice of a totally memory bound kernel ahead: memcpy D2D;
// dim(mat_a)=dim(mat_b)=2048x2048
int xIdx = blockIdx.x*blockDim.x+threadIdx.x;
int yIdx = blockIdx.y*blockDim.y+threadIdx.y;
if (xIdx < 2048 && yIdx < 2048)
   mat_a[xIdx][yIdx]=mat_b[xIdx][yIdx];
```

$$\mathbf{B} = \frac{\mathbf{D^r + D^w}}{\mathbf{t}} = \frac{\mathbf{2048^2 * 4 * 2}}{\mathbf{t}}$$

3. Compute the effective to theoretical bandwidth ratio. Then ask:
   - Is it around 70-75% of the peak? Good job*.
   - Is it much lower than 70% of the peak? Plenty of room for memory access optimization and performance improvement*.

*once again: the bandwidth metric is valid for memory bound kernel



Memory Optimizations

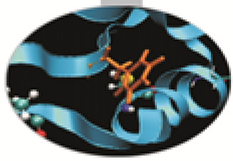
Data Transfers

- Host and Device have their own address space
- GPU boards are connected to host via PCIe bus
- Low bandwidth, extremely low latency

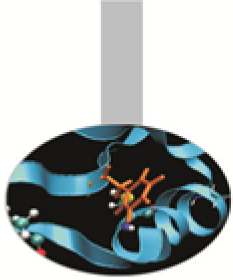
Technology	Peak Bandwidth
PCIex GEN2 (16x, full duplex)	8 GB/s (peak)
PCIex GEN3 (16x, full duplex)	16 GB/s (peak)
DDR3 (full duplex)	26 GB/s (single channel)

- Focus on how to minimize transfers and copybacks*.

* Try to find a good trade off!



Page-locked memory



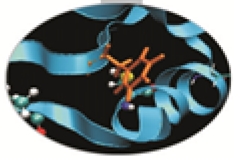
- Pinned (or page-locked memory) is a main memory area that is not pageable by the operating system;
- Ensures faster transfers (the DMA engine can work without CPU intervention);
- The only way to get closer to PCI peak bandwidth;
- Allows CUDA asynchronous operations (including *Zero Copy*) to work correctly.

```
// allocate page-locked memory
cudaMallocHost(&area, sizeof(double) * N);
// free page-locked memory
cudaFreeHost(area);
```

```
// allocate regular memory
area = (double*) malloc( sizeof(double) * N );
// lock area pages (CUDA >= 4.0)
cudaHostRegister( area, sizeof(double) * N, cudaHostRegisterPortable );
// unlock area pages (CUDA >= 4.0)
cudaHostUnregister(area);
// free regular memory
cudaFreeHost(area);
```

Warning: page-locked memory is a scarce resource.
Use with caution: allocating too much page-locked memory can reduce overall system performance
Breath relief: nVidia guys allocate up to 95% of a Linux compute node memory as 'pinned' memory in real world applications «without much problems» they say…

Zero Copy



- CUDA allows to map a page-locked host memory area to the device’s address space;

```
// allocate page-locked and mapped memory
cudaHostAlloc(&area, sizeof(double) * N, cudaHostAllocMapped);
// invoke retrieving device pointer for mapped area
cudaHostGetDevicePointer( &dev_area, area, 0 );
my_kernel<<< g, b >>>( dev_area );
// free page-locked and mapped memory
cudaFreeHost(area);
```

- The only way to provide on-the-fly a kernel data that doesn’t fit into the device’s global memory.
- Very convenient for large data with sparse access pattern.

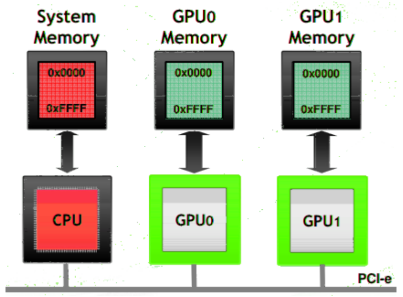
Unified Virtual Addressing

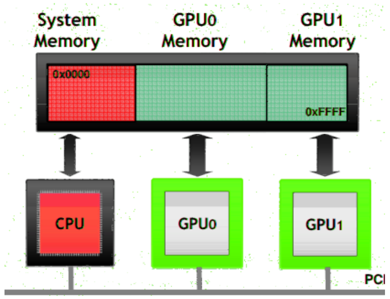
CUDA 4.0 introduced one (virtual) address space for all CPU and GPUs memory:

- automatically detects physical memory location from pointer value
- enables libraries to simplify their interfaces (e.g. `cudaMemcpy`)

Pre-UVA	UVA
Each source-destination permutation has its own option	Same interface

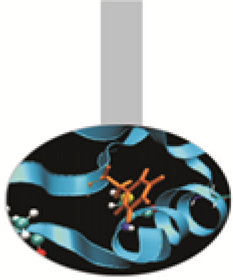
`cudaMemcpyHostToHost`
`cudaMemcpyHostToDevice`
`cudaMemcpyDeviceToHost`
`cudaMemcpyDeviceToDevice`
	`cudaMemcpyDefault`





Pointers returned by cudaHostAlloc() can be used directly from within kernels running on UVA enabled devices (i.e. there is no need to obtain a device pointer via cudaHostGetDevicePointer())

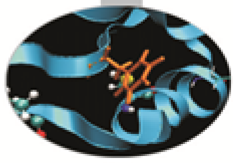
Multi-GPUs: P2P



```
cudaDeviceCanAccessPeer(&can_access_peer_0_1, gpuid_0, gpuid_1);
cudaDeviceCanAccessPeer(&can_access_peer_1_0, gpuid_1, gpuid_0);

cudaSetDevice(gpuid_0);
cudaDeviceEnablePeerAccess(gpuid_1, 0);

cudaSetDevice(gpuid_1);
cudaDeviceEnablePeerAccess(gpuid_0, 0);

cudaMemcpy(gpu0_buf, gpu1_buf, buf_size, cudaMemcpyDefault);
```

- `cudaMemcpy()` knows that our buffers are on different devices (UVA), will do a P2P copy now
- Note that this will *transparently* fall back to a normal copy through the host if P2P is not available

Multi-GPUs: direct access

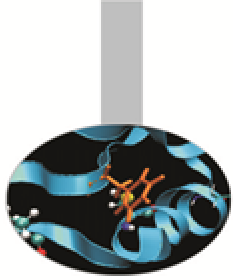


```
__global__ void SimpleKernel(float *src, float *dst) {
  const int idx = blockIdx.x * blockDim.x + threadIdx.x;
  dst[idx] = src[idx];
}
```

```
cudaDeviceCanAccessPeer(&can_access_peer_0_1, gpuid_0, gpuid_1);
cudaDeviceCanAccessPeer(&can_access_peer_1_0, gpuid_1, gpuid_0);

cudaSetDevice(gpuid_0);
cudaDeviceEnablePeerAccess(gpuid_1, 0);
cudaSetDevice(gpuid_1);
cudaDeviceEnablePeerAccess(gpuid_0, 0);

cudaSetDevice(gpuid_0);
SimpleKernel<<<blocks, threads>>> (gpu0_buf, gpu1_buf);
SimpleKernel<<<blocks, threads>>> (gpu1_buf, gpu0_buf);
cudaSetDevice(gpuid_1);
SimpleKernel<<<blocks, threads>>> (gpu0_buf, gpu1_buf);
SimpleKernel<<<blocks, threads>>> (gpu1_buf, gpu0_buf);
```

- After P2P initialization, this kernel can now read and write data in the memory of multiple GPUs (just *dereferencing pointers!*)
- UVA ensures that the kernel knows whether its argument is from local memory, another GPU or zero-copy from the host

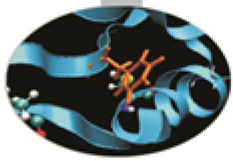
Asynchronous CPU/GPU operations



- Asynchronous operations: control is returned to the host thread before the device has completed the requested task
  - Kernel calls are asynchronous by default
  - Memory copies from host to device of a memory block of 64 KB or less
  - Memory set function calls
  - The **`cudaMemcpy()`** has an asynchronous version (**`cudaMemcpyAsync`**)
- Remember: standard memory transfers and copybacks *are* ***blocking***

```
// First transfer
cudaMemcpyAsync(d_A, h_A, size, cudaMemcpyHostToDevice, 0);
// First invocation
MyKernel<<<100, 512, 0, 0>>> (d_A, size);
// Second transfer
cudaMemcpyAsync(d_B, h_B, size, cudaMemcpyHostToDevice, 0);
// Second invocation
MyKernel2<<<100, 512, 0, 0>>> (d_B, size);
// Wrapup
cudaMemcpyAsync(h_A, d_A, size, cudaMemcpyDeviceToHost, 0);
cudaMemcpyAsync(h_B, d_B, size, cudaMemcpyDeviceToHost, 0);
cudaThreadSyncronize();
```

Asynchronous GPU Operations: CUDA Stream



- A **stream** is a FIFO command queue;
- **Default stream (aka stream '0')**: Kernel launches and memory copies that do not specify any stream (or set the stream to zero) are issued to the default stream.
- A stream is independent to every other active stream;
- Streams are the main way to exploit concurrent execution and I/O operations
- Explicit Synchronization:
  - cudaDeviceSynchronize()
    - blocks host until all issued CUDA calls are complete
  - cudaStreamSynchronize(streamId)
    - blocks host until all CUDA calls in streamid are complete
  - cudaStreamWaitEvent(streamId, event)
    - all commands added to the stream delay their execution until the event has completed
- Implicit Synchronization:
  - any CUDA command to the default stream,
  - a page-locked host memory allocation,
  - a device memory set or allocation,
  - …

CUDA streams enable concurrency

Concurrency: the ability to perform multiple CUDA operations simultaneously.

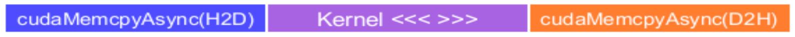
Fermi architecture can simultaneously support:

- Up to 16 CUDA kernels on GPU
- 2 cudaMemcpyAsyncs (in opposite directions)
- Computation on the CPU

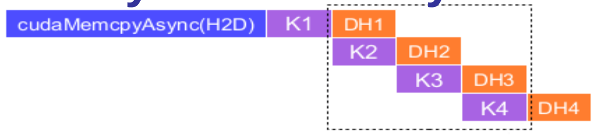
Requirements for Concurrency:

- CUDA operations must be in different, non-0, streams
- cudaMemcpyAsync with host from 'pinned' memory
- Sufficient resources must be available
  - cudaMemcpyAsyncs in different directions
  - Device resources (SMEM, registers, blocks, etc.)

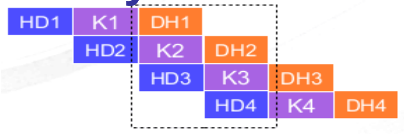
Serial :



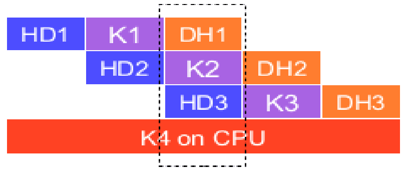
2-way concurrency :



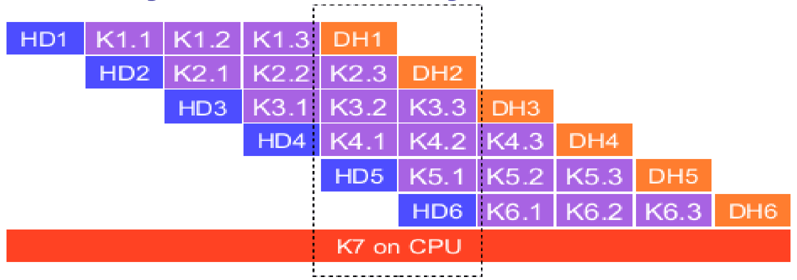
3-way concurrency :



4-way concurrency :

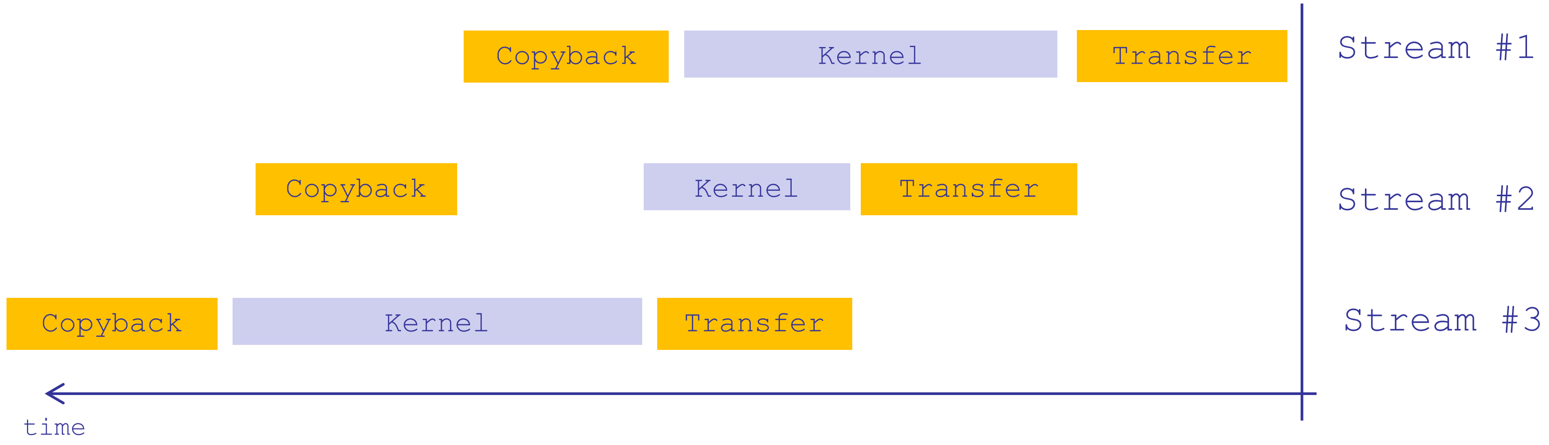


+4-way concurrency :

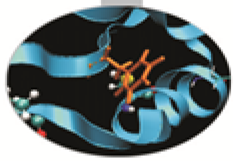


CUDA streams enable concurrency

```
cudaStream_t stream[3];
for (int i=0; i<3; ++i) cudaStreamCreate(&stream[i]);

float* hPtr; cudaMallocHost((void**)&hPtr, 3 * size);

for (int i=0; i<3; ++i) {
  cudaMemcpyAsync(d_inp + i*size, hPtr + i*size,
                        size, cudaMemcpyHostToDevice, stream[i]);

  MyKernel<<<100, 512, 0, stream[i]>>>(d_out+i*size, d_inp+i*size, size);

  cudaMemcpyAsync(hPtr + i*size, d_out + i*size,
                        size, cudaMemcpyDeviceToHost, stream[i]);
}
cudaDeviceSynchronize();

for (int i=0; i<3; ++i) cudaStreamDestroy(&stream[i]);
```



CUDA Streams: overlapping kernels execution



- Starting from capability 2.0 the board has the ability to overlap computations from multiple kernels.
  - CUDA kernels are in different streams,
  - no operations occur on the default stream,
  - the active streams are less than 16.
  - no synchronization happens between command stages,
- Threadblocks for a given kernel are scheduled if all threadblocks for preceding kernels have already been scheduled and there are SM resources available
- Concurrent execution can be limited by implicit dependencies due to hardware limitations: command issue order matters!

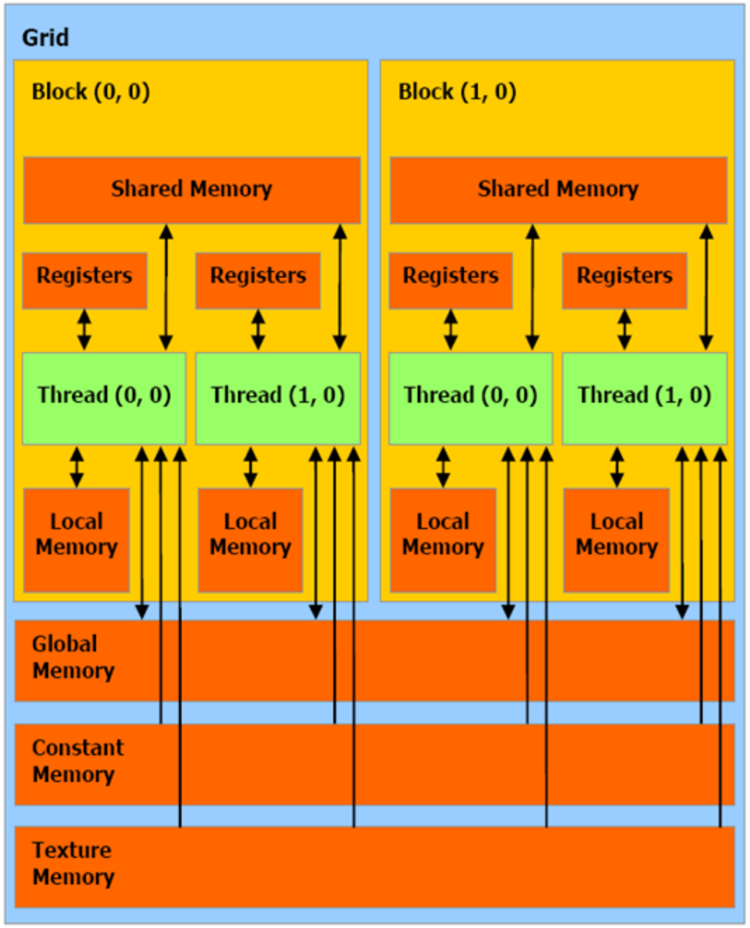
```
// Depth-first commands submission.
Beware: PSEUDO CODE ahead:
for each StreamId:
  do H2D data tile transfer
  launch kernel on data tile
  do D2H result data tile transfer
```

$\neq$

```
// Breadth-first commands submission.
Beware: PSEUDO CODE ahead:
for each StreamId:
  do H2D data tile transfer
for each StreamId:
  launch kernel on data tile
for each StreamId:
  do D2H result data tile transfer
```

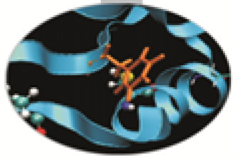
hint: depth-first commands submission is usually better on Fermi. It's a no-issue for Kepler K20 with HyperQ technology

CUDA Memory Hierarchy



Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	†	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

Global Memory

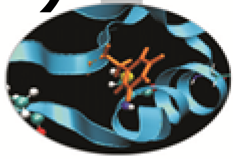


- It is a memory area with the same purpose of the host's main memory;
- High(er) bandwidth, high(er) latency;
- In order to exploit its bandwidth at best, all accesses must be *coalesced*, i.e. memory accesses from different threads need to be grouped toghether and serviced in one memory transaction.
- beware: some threads memory access patterns can be coalesced, some others cannot (coalescence rules depends on GPU compute capability)
- **FERMI** architecture introduces caching mechanisms for GMEM accesses (constant and texture are cached since 1.0)
- L1: private to thread, virtual cache implemented into shared memory
- L2: 768KB, grid-coherent, 25% better latency than DRAM

```
// L1 = 48 KB
// SH = 16 KB
cudaFuncSetCacheConfig( kernel, cudaFuncCachePreferL1);
// L1 = 16 KB
// SH = 48 KB
cudaFuncSetCacheConfig( kernel, cudaFuncCachePreferShared );
```

Kepler architecture introduced some improvements:
32 KB + 32 KB partition option

Global Memory (pre-Fermi)



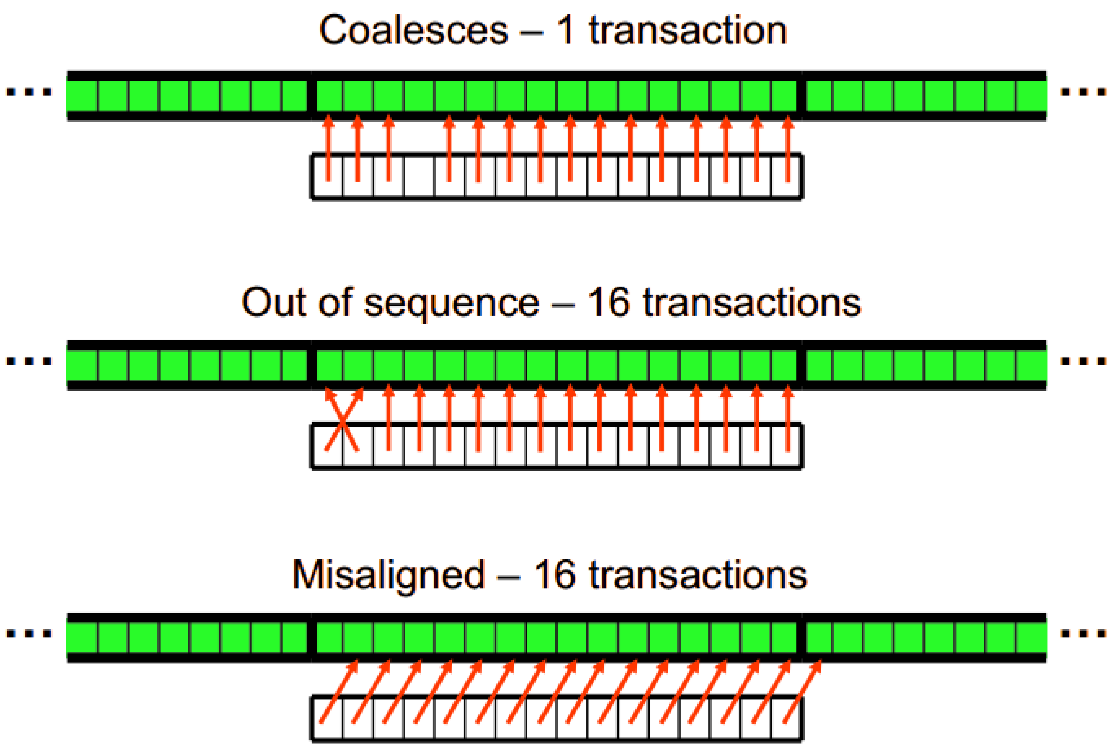
Compute capability 1.0 and 1.1

- A global memory request for a warp is split into two memory requests, one for each half-warp, that are issued independently.
- In order to exploit its bandwidth at best, all accesses must be *coalesced* (*half-warp* accesses contiguous region of device memory).
- Threads must access the words in a strictly increasing sequence: *the nth thread in the half-warp must access the nth word.*
- All 16 words must lie in the same aligned segment
- A coalesced memory access results in:
  - in one 64-byte memory transaction, for 4-byte words
  - in one 128-byte memory transaction, for 8-byte words
  - in two 128-byte memory transactions, for 16-byte words

Coalescing (pre-Fermi)

Compute capability 1.0 and 1.1

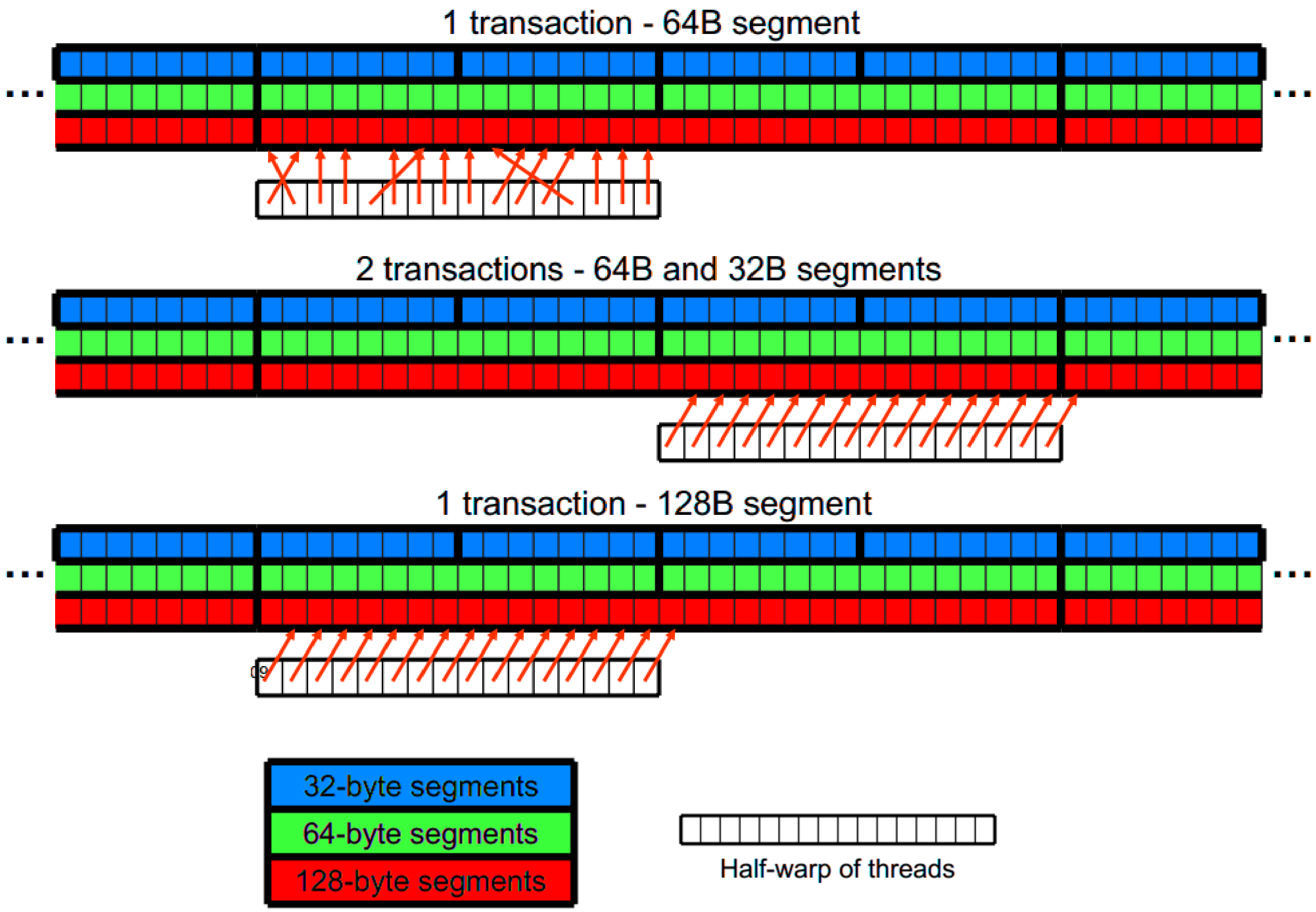
- stricter access requirements
- memory accesses serviced on a half-warp (16 threads) basis
- not all threads need to participate but their memory accesses must be aligned and in order:
  - k-th thread must access k-th word in the segment


Coalesces – 1 transaction
Out of sequence – 16 transactions
Misaligned – 16 transactions

Coalescing (pre-Fermi)

Compute capability 1.2 and 1.3

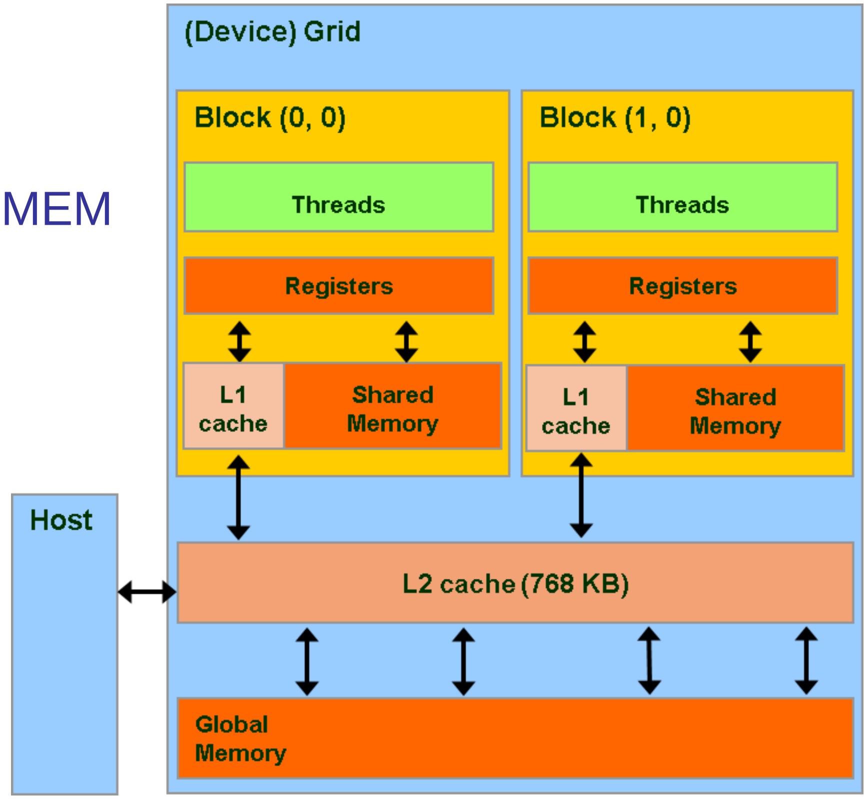
- The memory controller is much improved


1 transaction - 64B segment
2 transactions - 64B and 32B segments
1 transaction - 128B segment
32-byte segments
64-byte segments
128-byte segments
Half-warp of threads

Global Memory (Fermi)

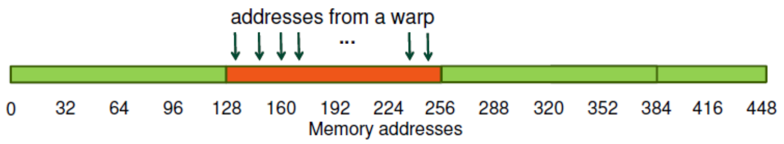
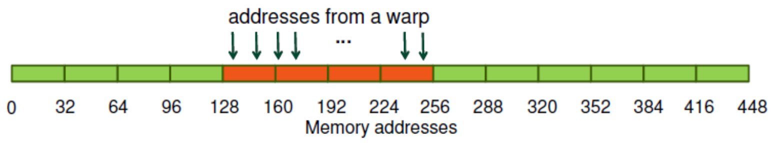
FERMI (Compute Capability 2.x) GMEM Operations

- 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`
    - attempts to hit in L2, then GMEM does not hit in L1.
    - load granularity is **32-bytes**
- Stores:
  - Invalidate L1, write-back for L2



Global Memory Load Operation (Fermi)

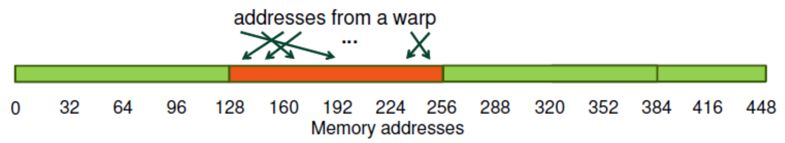
- Memory operations are issued per warp (32 threads)
  - just like all other instructions
- Operation:
  - Threads in a warp provide memory addresses
  - Determine which lines/segments are needed
  - Request the needed lines/segments

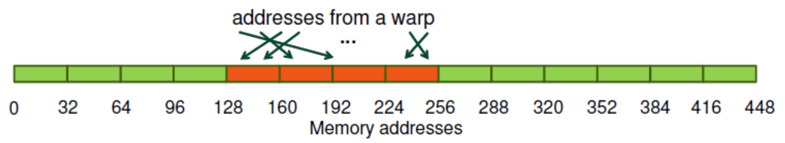
Warp requests 32 aligned, consecutive 4-byte words (128 bytes)	
Caching Load	**Non-caching Load**
Addresses fall within 1 cache-line	Addresses fall within 4 segments
128 bytes move across the bus	128 bytes move across the bus
Bus utilization: **100%**	Bus utilization: **100%**
 addresses from a warp ... 0 32 64 96 128 160 192 224 256 288 320 352 384 416 448 Memory addresses	 addresses from a warp ... 0 32 64 96 128 160 192 224 256 288 320 352 384 416 448 Memory addresses

Global Memory Load Operation (Fermi)

Warp requests 32 aligned, permuted 4-byte words (128 bytes)

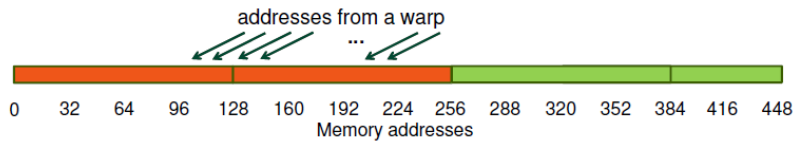
Caching Load	Non-caching Load
Addresses fall within 1 cache-line	Addresses fall within 4 segments
128 bytes move across the bus	128 bytes move across the bus
Bus utilization: **100%**	Bus utilization: **100%**

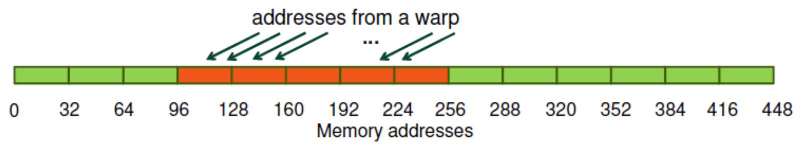




Warp requests 32 misaligned, consecutive 4-byte words (128 bytes)

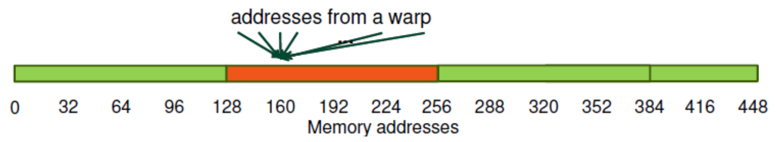
Caching Load	Non-caching Load
Addresses fall within 2 cache-lines	Addresses fall within at most 5 segments
256 bytes move across the bus	160 bytes move across the bus
Bus utilization: **50%**	Bus utilization: **at least 80%**

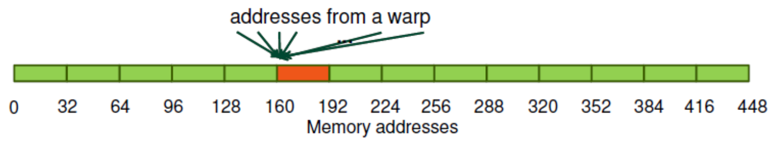




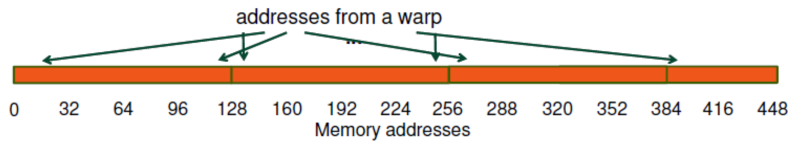
Global Memory Load Operation (Fermi)

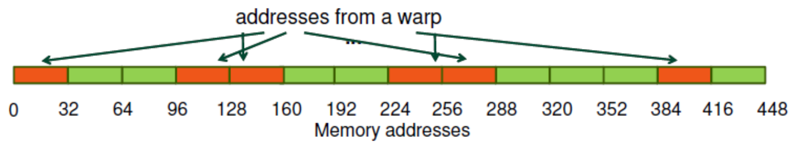
All threads in a warp request the same 4-byte word (4 bytes)	
Caching Load	**Non-caching Load**
Addresses fall within 1 cache-line	Addresses fall within 1 segments
128 bytes move across the bus	32 bytes move across the bus
Bus utilization: **3.125%**	Bus utilization: **12.5%**



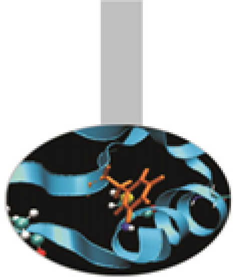


Warp requests 32 scattered 4-byte words (128 bytes)	
Caching Load	**Non-caching Load**
Addresses fall within N cache-lines	Addresses fall within N segments
N*128 bytes move across the bus	N*32 bytes move across the bus
Bus utilization: **128 / (N*128)**	Bus utilization: **128 / (N*32)**





Shared memory



- A sort of *explicit* cache (i.e. under programmer control)
- Resides on the chip so it is *much* faster than the on-board memory
- Divided into equally-sized memory modules (banks) which can be accessed simultaneously (32 banks can be accessed simultaneously by the same warp)
- 48KB on Fermi by default*

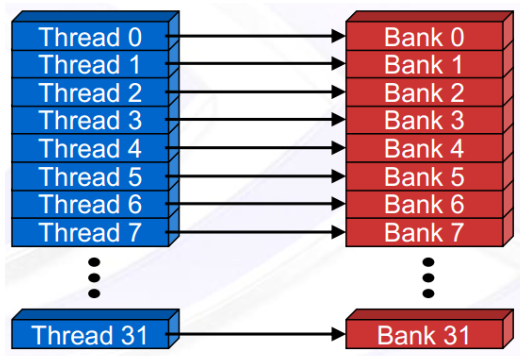
*__Kepler__ architecture introduced some improvements:
- ability to switch from 4B to 8B banks
- (2x bandwidth for double precision codes)

- **Uses**:
  - Inter-thread communication within a block
  - Cache data to reduce redundant global memory accesses
  - To improve global memory access patterns
- **Organization**:
  - 32 banks, 4-byte wide banks
  - Successive 4-byte words belong to different banks
  - Each bank has 32-bit per cycle bandwidth.

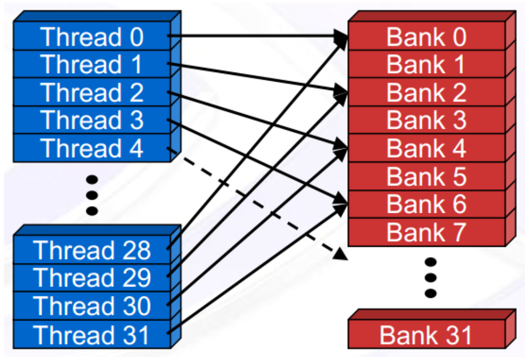
Shared Memory Bank Conflicts

- If at least two threads belonging to the same half-warp (whole warp for capability 1.0) access the same shared memory bank, there is a ***bank conflict*** and the accesses are serialized (groups transactions in conflict-free accesses);
- If all the threads access the same address, a *broadcast* is performed;
- If part of the half-warp accesses the same address, a *multicast* is performed (capability >= 2.0);

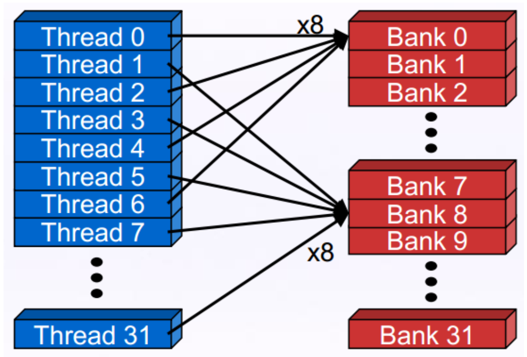
No Bank Conflict



2-way Bank Conflicts



8-way Bank Conflicts



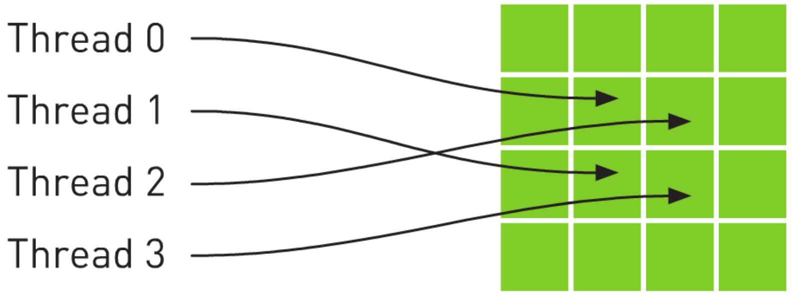
Lunch break

The second part will start at 14:30.
Please, try to be on time ☺

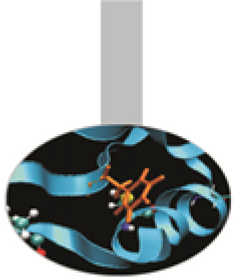
Texture Memory

- **Read only**, must be set by the host;
- Load requests are cached (dedicated cache);
- specifically, texture memories and caches are designed for graphics applications where memory access patterns exhibit a great deal of spatial locality;
- Dedicated texture cache hardware provides:
  - Out-of-bounds index handling (clamp or wrap-around)
  - Optional interpolation (on-the-fly interpolation)
  - Optional format conversion
- could bring benefits if the threads within the same block access memory using regular 2D patterns, but you need appropriate binding;

For typical linear patterns, global memory (if coalesced) is faster.

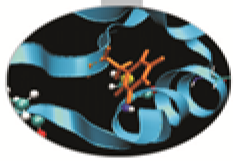


Texture Memory



```
// allocate array and copy image data
cudaChannelFormatDesc channelDesc =
                    cudaCreateChannelDesc(32, 0, 0, 0, cudaChannelFormatKindFloat);
cudaArray* cu_array;
cudaMallocArray( &cu_array, &channelDesc, width, height );
cudaMemcpyToArray( cu_array, 0, 0, h_data, size, cudaMemcpyHostToDevice);
// set texture parameters
tex.addressMode[0] = cudaAddressModeWrap;
tex.addressMode[1] = cudaAddressModeWrap;
tex.filterMode = cudaFilterModeLinear;
tex.normalized = true;    // access with normalized texture coordinates
// Bind the array to the texture
cudaBindTextureToArray( tex, cu_array, channelDesc);
```

```
// declare texture reference for 2D float texture
texture<float, 2, cudaReadModeElementType> tex;

__global__ void transformKernel( float* g_odata, int width, int height, float theta)
{
  // calculate normalized texture coordinates
  unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
  unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;
  float u = x / (float) width;
  float v = y / (float) height;
  // transform coordinates
  u -= 0.5f;
  v -= 0.5f;
  float tu = u*cosf(theta) - v*sinf(theta) + 0.5f;
  float tv = v*cosf(theta) + u*sinf(theta) + 0.5f;
  // read from texture and write to global memory
  g_odata[y*width + x] = tex2D(tex, tu, tv);
}
```

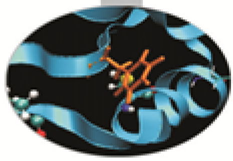


Kepler global loads through texture

The compiler (LLVM) can detect texture-compliant loads and map them to the new «*global load through texture*» `PTX` instruction:

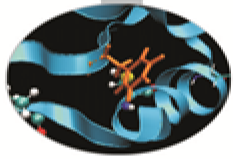
- global loads are going to pass through texture pipeline;
- dedicated cache (no L1 pressure) and memory pipe, relaxed coalescing;
- automatically generated by compiler (no texture map needed) for accesses through compliant pointers (*constant* and *restricted*);
- useful for bandwidth-limited kernels
  - global memory bandwidth and texture memory bandwidth stack up.

Constant Memory



- Extremely fast on-board memory area
- **Read only**, must be set by the host
- 64 KB, cached reads in a dedicated L1 (register space)
- Coalesced access if all threads of a warp read the same address (serialized otherwise)
- `__constant__` qualifier in declarations
- Useful:
  - To off-load long argument lists from shared memory (compute capability 1.x)
  - For coefficients and other data that is read uniformly by warps

```
__device__ __constant__ parameters_t args;

__host__ void copy_params(const parameters_t* const host_args) {

    cudaMemcpyToSymbol(“args", host_args, sizeof(parameters_t));

}
```



Registers

- Just like CPU registers, access has no latency;
- used for scalar data local to a thread;
- taken by the compiler from the Streaming Multiprocessor (SM) pool and statically allocated to each thread;
    - each SM of a Fermi GPU has a 32KB register file, 64KB for a Kepler GPU
- *register pressure one of the most dangerous occupancy limiting factors.*

Registers

Some tips to reduce register pressure:

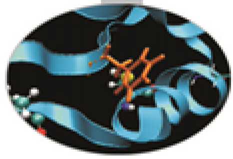
- try to offload data to shared memory;
- use launch bounds to force the number of resident blocks;

```
#define MAX_THREADS_PER_BLOCK 256
#define MIN_BLOCKS_PER_MP     2

__global__ void
__launch_bounds__( MAX_THREADS_PER_BLOCK,
MIN_BLOCKS_PER_MP )
my_kernel( int* inArr, int* outArr ) { … }
```

- limit register usage via compiler option.

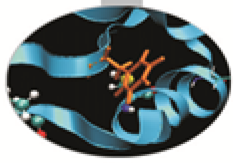
```
# nvcc -Xptas -v mykernel.cu
ptxas info    : Compiling entry function '_Z12my_kernelP9domain_t_' for 'sm_20'
ptxas info    : Used 13 registers, 8+16 bytes smem
```

```
# nvcc --maxrregcount 10 -Xptas -v mykernel.cu
ptxas info    : Compiling entry function '_Z12my_kernelP9domain_t_' for 'sm_20'
ptxas info    : Used 10 registers, 12+0 bytes lmem, 8+16 bytes smem
```



Local memory

- “Local” because it’s private on a per-thread basis;
- it’s actually a global memory area used to spill out data when the SM runs out of register resources;
- arrays declared inside a kernel go to LMEM;
- local memory accesses are cached (just like global memory).
- DISCLAIMER: local memory is not a GPU resource you want to use: It used by the compiler as needed. Its use can hardly hit your kernel performance too: variables that you think are in registers are instead stored in the device global memory.

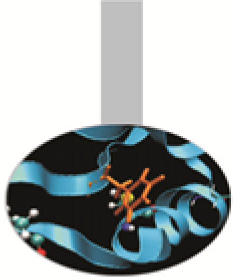


Execution Optimization

Occupancy

The board's occupancy is the ratio of active warps to the maximum number of warps supported on a multiprocessor.

Keeping the hardware busy helps the warp scheduler to hide latencies.

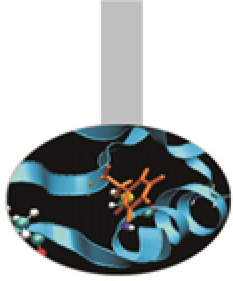


Occupancy: constraints

Every board’s resource can become an occupancy limiting factor:

- shared memory allocated per block,
- registers allocated per thread,
- block size
  - (max threads (warp) per SM/max blocks per SM)

Given an actual kernel configuration, is possible to predict the maximum *theoretical occupancy* allowed.



Occupancy: block sizing tips

Some experimentation is required.

However there are some heuristic rules:

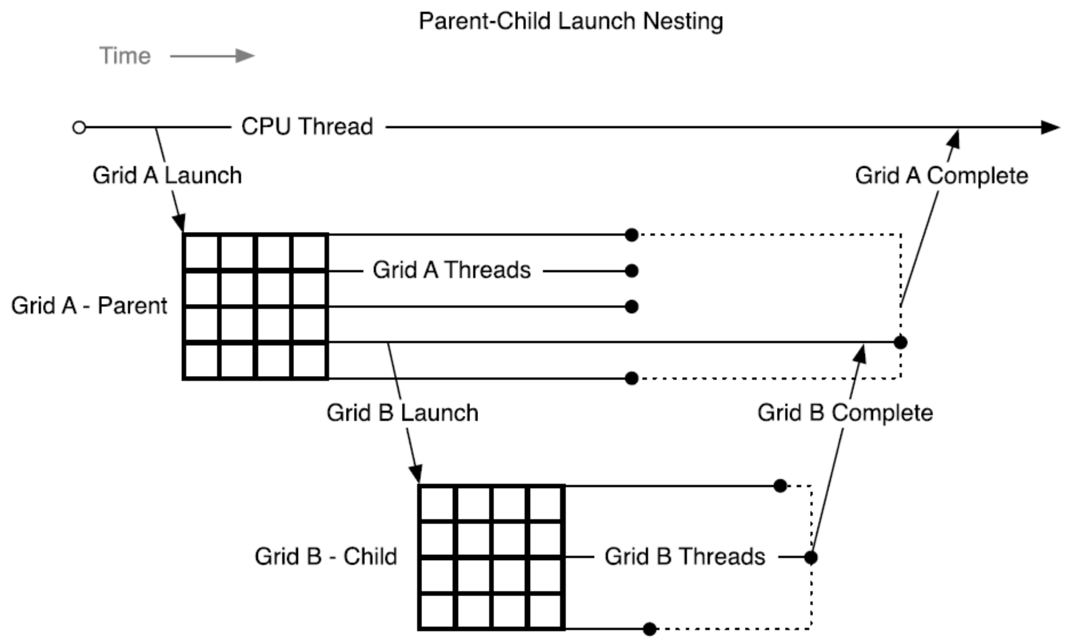
- threads per block should be a **multiple of warp size**;
- a minimum of **64 threads per block** should be used;
- **128-256 threads per block** is universally known to be a good starting point for further experimentation;
- prefer to split **very large** blocks into **smaller blocks**.

Kepler: dynamic parallelism

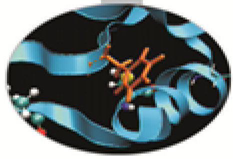
- One of the biggest CUDA limitations is the need to fit a single grid configuration for the whole kernel.

If you need to reshape the grid, you have to resync back to host and split your code.

- Kepler K20 (in addition to CUDA 5.x) introduced *Dynamic Parallelism*
- It enables a global kernel to be called from within another kernel
- The child grid can be *dynamically sized* and *optionally synchronized*



```
__global__ ChildKernel(void* data){
  //Operate on data
}

__global__ ParentKernel(void *data){
  ChildKernel<<<16, 1>>>(data);
}

// In Host Code:
ParentKernel<<<256, 64>>>(data);
```

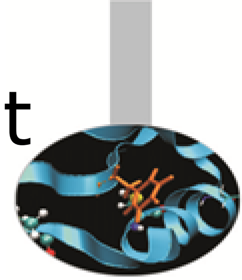


Instructions throughput

Arithmetic ops:

- prefer integer shift operators instead of division and modulo (would be less useful with LLVM);
- beware of (implicit) casts (very expensive);
- use intrinsics for trascendental functions where possible;
- try the fast math implementation.

Capability: instruction throughput



	Compute Capability					
	1.0 1.1 1.2	1.3	2.0	2.1	3.0	3.5
32-bit floating-point add, multiply, multiply-add	8	8	32	48	192	192
64-bit floating-point add, multiply, multiply-add	1	1	16(*)	4	8	64
32-bit integer add	10	10	32	48	160	160
32-bit integer compare	10	10	32	48	160	160
32-bit integer shift	8	8	16	16	32	64
Logical operations	8	8	32	48	160	160
32-bit integer multiply, multiply-add, sum of absolute difference	Multiple instructions	Multiple instructions	16	16	32	32
24-bit integer multiply (`__[u]mul24`)	8	8	Multiple instructions	Multiple instructions	Multiple instructions	Multiple instructions
32-bit floating-point reciprocal, reciprocal square root, base-2 logarithm (`__log2f`), base 2 exponential (`exp2f`), sine (`__sinf`), cosine (`__cosf`)	2	2	4	8	32	32
Type conversions from 8-bit and 16-bit integer to 32-bit types	8	8	16	16	128	128
Type conversions from and to 64-bit types	Multiple instructions	1	16(*)	4	8	32
All other type conversions	8	8	16	16	32	32
(*) Throughput is lower for GeForce GPUs.						

instructions x cycle x SM

Control Flow

Different execution paths inside the same warp are managed by the predication mechanism and lead to thread divergence.

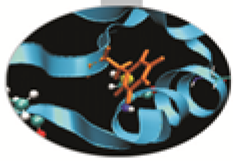
```
if ( threadIdx.x == 0 ) {…}
```

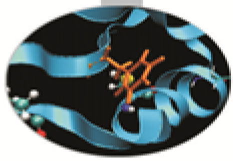
```
if ( threadIdx.x == 0 ) {…}
else {…}
```

```
if ( threadIdx.x == 0 ) {…}
else  if (threadIdx.x == 1) {…}
```

```
if ( vec[ threadIdx.x ] > 1.0f ) {…}
```

- Minimize/<u>avoid</u> the number of execution branches inside <u>a threads warp</u>;
- make the compiler's life easier by <u>unrolling</u> loops (hand-coded, pragma or option);
- use signed counters for loops (relaxed semantic in respect to the unsigned int: it allows more aggressive loop optimizations);





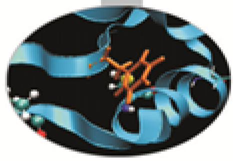
Exploiting Multi-GPUs

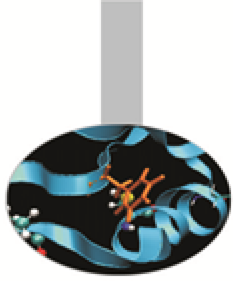
CUDA >= 4.0 introduced the N-to-N bound feature:

1. Every **host** thread can be bound to any board
2. Every board can be bound to an arbitrary number of **host** threads
3. Multi-GPU can be exploited through your favourite multi-threading paradigm (OpenMP, pthreads, etc…)

```
#pragma omp parallel
#pragma omp sections
{
#pragma omp section
  {
     cutilSafeCall(cudaSetDevice(0));
     cudaMemcpy(device_data_1, host_data_1, size, cudaMemcpyHostToDevice);
     my_kernel<<< grid, block >>>(device_data_1);
     // ...
  }
#pragma omp section
  {
     cutilSafeCall(cudaSetDevice(1));
     cudaMemcpy(device_data_2, host_data_2, size, cudaMemcpyHostToDevice);
     my_kernel<<< grid, block >>>(device_data_2);
     // ...
  }
}
```

Tools Overview

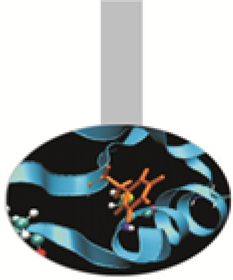




Development tools

- Common
  - Memory Checker
  - Built-in profiler
  - Visual Profiler
- Linux
  - CUDA GDB
  - Parallel Nsight for Eclipse
- Windows
  - Parallel Nsight for VisualStudio

Profiling tools: built-in



The CUDA runtime provides a useful profiling facility without the need of external tools.

```
export CUDA_PROFILE=1
export CUDA_PROFILE_CONFIG=$HOME/.config
```

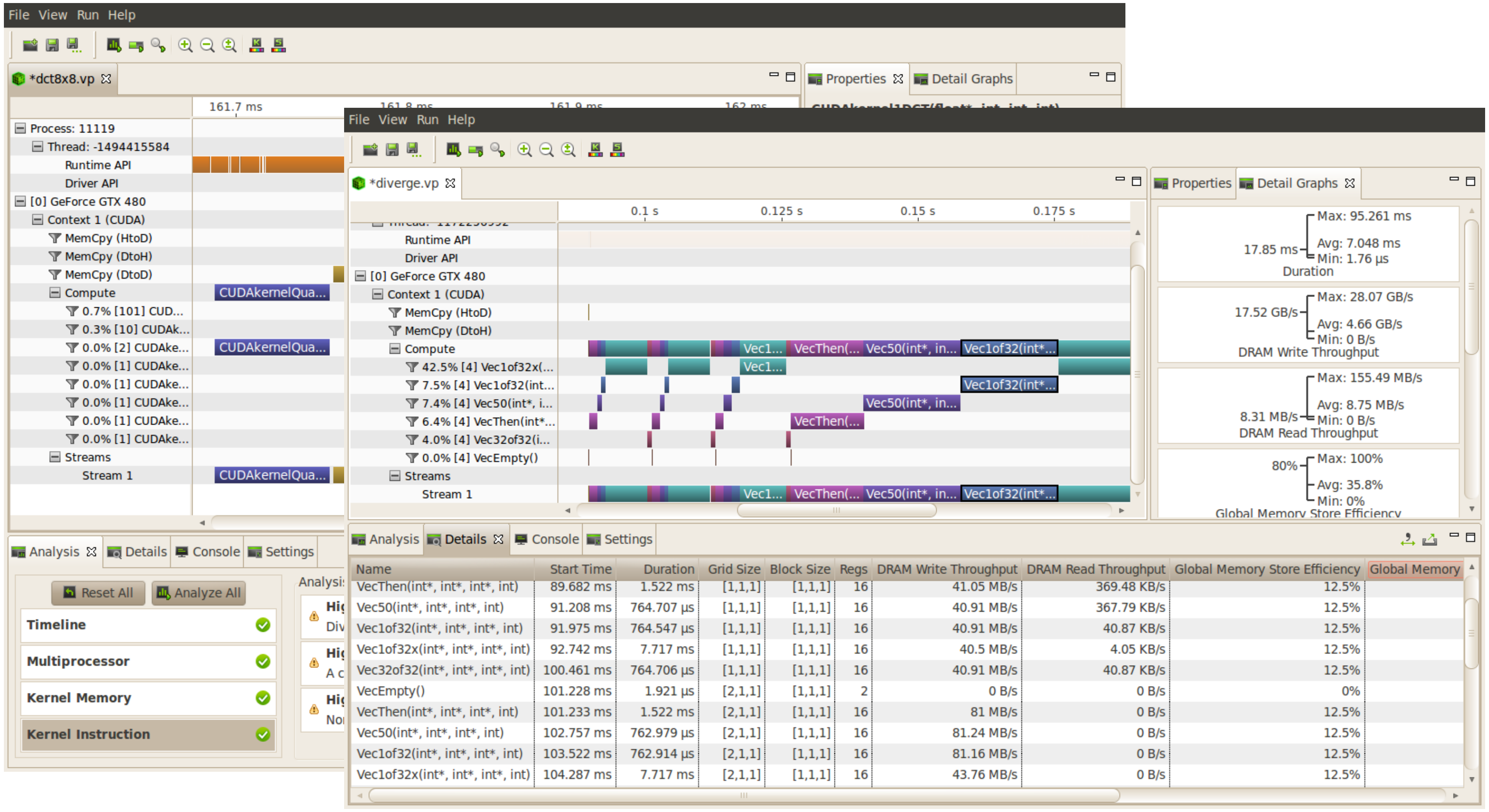
```
// Contents of config
gld_coherent
gld_incoherent
gst_coherent
gst_incoherent
```

```
gld_incoherent: Number of non-coalesced global memory loads
gld_coherent: Number of coalesced global memory loads
gst_incoherent: Number of non-coalesced global memory stores
gst_coherent: Number of coalesced global memory stores
local_load: Number of local memory loads
local_store: Number of local memory stores
branch: Number of branch events taken by threads
divergent_branch: Number of divergent branches within a warp
instructions: instruction count
warp_serialize: Number of threads in a warp that serialize
based on address conflicts to shared or constant memory
cta_launched: executed thread blocks
```

```
method,gputime,cputime,occupancy,gld_incoherent,gld_coherent,gst_incoherent,gst_coherent
method=[ memcopy ] gputime=[ 438.432 ]
method=[ _Z17reverseArrayBlockPiS_ ] gputime=[ 267.520 ] cputime=[ 297.000 ] occupancy=[ 1.000 ]
gld_incoherent=[ 0 ] gld_coherent=[ 1952 ] gst_incoherent=[ 62464 ] gst_coherent=[ 0 ]
method=[ memcopy ] gputime=[ 349.344 ]
```

Profiling: Visual Profiler

- Traces execution at host, driver and kernel levels (unified timeline)
- Supports automated analysis (hardware counters)



Duration: 17.85 ms — Max: 95.261 ms, Avg: 7.048 ms, Min: 1.76 µs

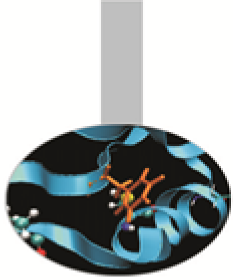
DRAM Write Throughput: 17.52 GB/s — Max: 28.07 GB/s, Avg: 4.66 GB/s, Min: 0 B/s

DRAM Read Throughput: 8.31 MB/s — Max: 155.49 MB/s, Avg: 8.75 MB/s, Min: 0 B/s

Global Memory Store Efficiency: 80% — Max: 100%, Avg: 35.8%, Min: 0%

Name	Start Time	Duration	Grid Size	Block Size	Regs	DRAM Write Throughput	DRAM Read Throughput	Global Memory Store Efficiency	Global Memory
VecThen(int*, int*, int*, int)	89.682 ms	1.522 ms	[1,1,1]	[1,1,1]	16	41.05 MB/s	369.48 KB/s	12.5%	
Vec50(int*, int*, int*, int)	91.208 ms	764.707 µs	[1,1,1]	[1,1,1]	16	40.91 MB/s	367.79 KB/s	12.5%	
Vec1of32(int*, int*, int*, int)	91.975 ms	764.547 µs	[1,1,1]	[1,1,1]	16	40.91 MB/s	40.87 KB/s	12.5%	
Vec1of32x(int*, int*, int*, int)	92.742 ms	7.717 ms	[1,1,1]	[1,1,1]	16	40.5 MB/s	4.05 KB/s	12.5%	
Vec32of32(int*, int*, int*, int)	100.461 ms	764.706 µs	[1,1,1]	[1,1,1]	16	40.91 MB/s	40.87 KB/s	12.5%	
VecEmpty()	101.228 ms	1.921 µs	[2,1,1]	[1,1,1]	2	0 B/s	0 B/s	0%	
VecThen(int*, int*, int*, int)	101.233 ms	1.522 ms	[2,1,1]	[1,1,1]	16	81 MB/s	0 B/s	12.5%	
Vec50(int*, int*, int*, int)	102.757 ms	762.979 µs	[2,1,1]	[1,1,1]	16	81.24 MB/s	0 B/s	12.5%	
Vec1of32(int*, int*, int*, int)	103.522 ms	762.914 µs	[2,1,1]	[1,1,1]	16	81.16 MB/s	0 B/s	12.5%	
Vec1of32x(int*, int*, int*, int)	104.287 ms	7.717 ms	[2,1,1]	[1,1,1]	16	43.76 MB/s	0 B/s	12.5%	

Debugging: CUDA-GDB



- Well-known tool enhanced with CUDA extensions
- Works well on single-gpu systems (OS graphics disabled)
- Can be run under GDB-targeted tools and GUIs (multi-gpu systems)

```
(cuda-gdb) info cuda threads
BlockIdx ThreadIdx To BlockIdx ThreadIdx Count Virtual PC Filename Line
Kernel 0* (0,0,0) (0,0,0) (0,0,0) (255,0,0) 256 0x0000000000866400 bitreverse.cu 9
(cuda-gdb) thread
[Current thread is 1 (process 16738)]
(cuda-gdb) thread 1
[Switching to thread 1 (process 16738)]
#0 0x000019d5 in main () at bitreverse.cu:34
34 bitreverse<<<1, N, N*sizeof(int)>>>(d);
(cuda-gdb) backtrace
#0 0x000019d5 in main () at bitreverse.cu:34
(cuda-gdb) info cuda kernels
Kernel Dev Grid SMs Mask GridDim BlockDim Name Args
0 0 1 0x00000001 (1,1,1) (256,1,1) bitreverse data=0x110000
```

Debugging: CUDA-MEMCHECK

- It’s able to detect buffer overflows, misaligned global memory accesses and leaks
- Device-side allocations are supported
- Standalone or fully integrated in CUDA-GDB

```
$ cuda-memcheck --continue ./memcheck_demo
========= CUDA-MEMCHECK
Mallocing memory
Running unaligned_kernel
Ran unaligned_kernel: no error
Sync: no error
Running out_of_bounds_kernel
Ran out_of_bounds_kernel: no error
Sync: no error
========= Invalid __global__ write of size 4
========= at 0x00000038 in memcheck_demo.cu:5:unaligned_kernel
========= by thread (0,0,0) in block (0,0,0)
========= Address 0x200200001 is misaligned
=========
========= Invalid __global__ write of size 4
========= at 0x00000030 in memcheck_demo.cu:10:out_of_bounds_kernel
========= by thread (0,0,0) in block (0,0,0)
========= Address 0x87654320 is out of bounds
=========
=========
========= ERROR SUMMARY: 2 errors
```

Parallel NSight

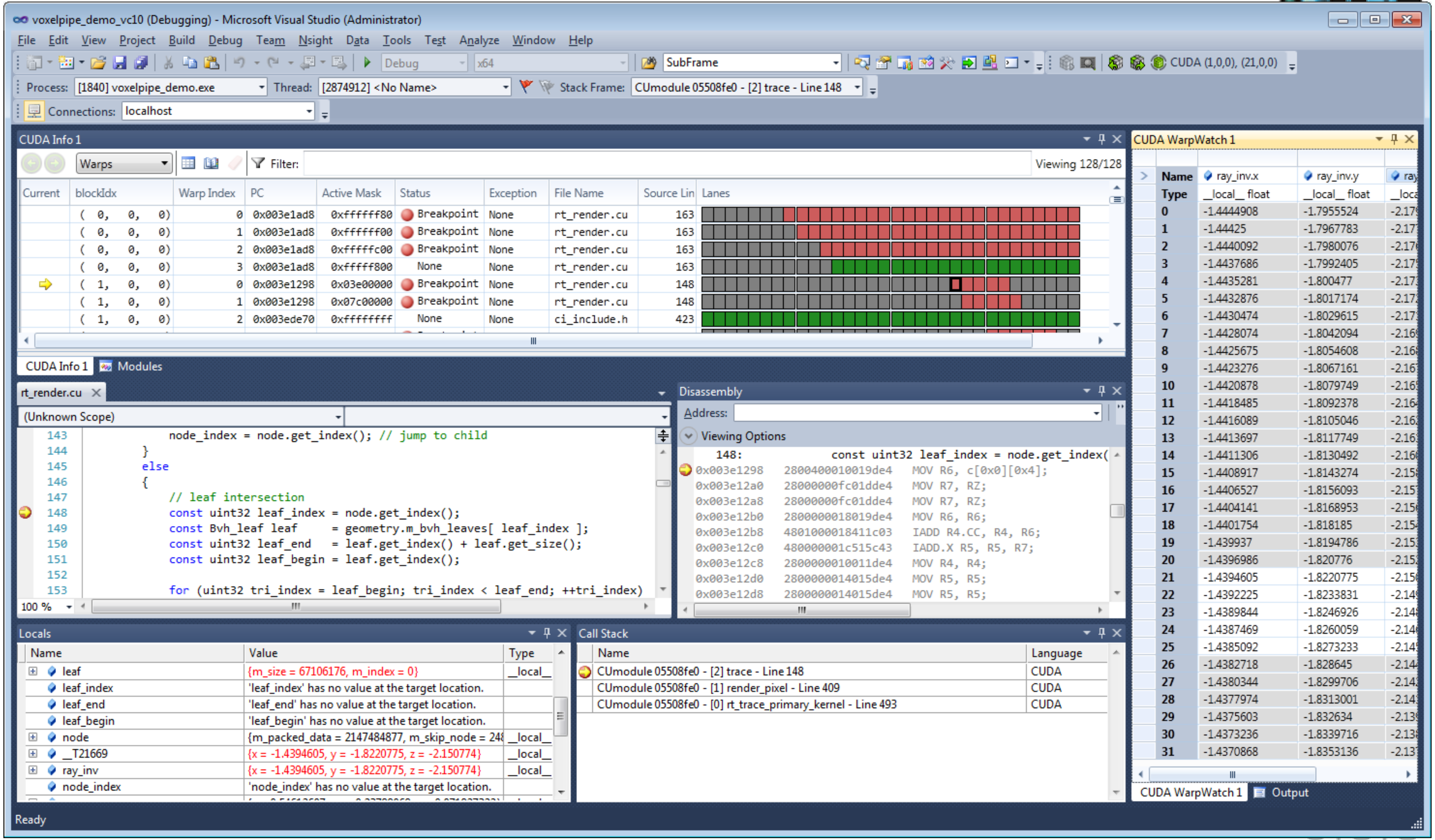
- Plug-in for major IDEs (Eclipse and VisualStudio)
- Aggregates all external functionalities:
  - Debugger (fully integrated)
  - Visual Profiler
  - Memory correctness checker
- As a plug-in, it extends all the convenience of IDEs to CUDA

On Windows systems:

- Now works on a single GPU
- Supports remote debugging and profiling
- Latest version (2.2) introduced live PTX assembly view, warp inspector and expression lamination

CINECA SCAI SuperComputing Applications and Innovation

Parallel NSight


voxelpipe_demo_vc10 (Debugging) - Microsoft Visual Studio (Administrator)

File Edit View Project Build Debug Team Nsight Data Tools Test Analyze Window Help

Debug x64 SubFrame CUDA (1,0,0), (21,0,0)

Process: [1840] voxelpipe_demo.exe Thread: [2874912] <No Name> Stack Frame: CUmodule 05508fe0 - [2] trace - Line 148

Connections: localhost

CUDA Info 1

Warps Filter: Viewing 128/128

Current	blockIdx	Warp Index	PC	Active Mask	Status	Exception	File Name	Source Lin	Lanes
	(0, 0, 0)	0	0x003e1ad8	0xffffff80	Breakpoint	None	rt_render.cu	163	
	(0, 0, 0)	1	0x003e1ad8	0xffffff00	Breakpoint	None	rt_render.cu	163	
	(0, 0, 0)	2	0x003e1ad8	0xfffffc00	Breakpoint	None	rt_render.cu	163	
	(0, 0, 0)	3	0x003e1ad8	0xfffff800	None	None	rt_render.cu	163	
⇨	(1, 0, 0)	0	0x003e1298	0x03e00000	Breakpoint	None	rt_render.cu	148	
	(1, 0, 0)	1	0x003e1298	0x07c00000	Breakpoint	None	rt_render.cu	148	
	(1, 0, 0)	2	0x003ede70	0xffffffff	None	None	ci_include.h	423	

CUDA Info 1 Modules

rt_render.cu

(Unknown Scope)

```
            node_index = node.get_index(); // jump to child
        }
        else
        {
            // leaf intersection
            const uint32 leaf_index = node.get_index();
            const Bvh_leaf leaf     = geometry.m_bvh_leaves[ leaf_index ];
            const uint32 leaf_end   = leaf.get_index() + leaf.get_size();
            const uint32 leaf_begin = leaf.get_index();

            for (uint32 tri_index = leaf_begin; tri_index < leaf_end; ++tri_index)
```

100 %

Disassembly

Address:

Viewing Options

```
    148:              const uint32 leaf_index = node.get_index(
0x003e1298    2800400010019de4    MOV R6, c[0x0][0x4];
0x003e12a0    28000000fc01dde4    MOV R7, RZ;
0x003e12a8    28000000fc01dde4    MOV R7, RZ;
0x003e12b0    2800000018019de4    MOV R6, R6;
0x003e12b8    4801000018411c03    IADD R4.CC, R4, R6;
0x003e12c0    480000001c515c43    IADD.X R5, R5, R7;
0x003e12c8    2800000010011de4    MOV R4, R4;
0x003e12d0    2800000014015de4    MOV R5, R5;
0x003e12d8    2800000014015de4    MOV R5, R5;
```

Locals

Name	Value	Type
leaf	{m_size = 67106176, m_index = 0}	__local__
leaf_index	'leaf_index' has no value at the target location.	
leaf_end	'leaf_end' has no value at the target location.	
leaf_begin	'leaf_begin' has no value at the target location.	
node	{m_packed_data = 2147484877, m_skip_node = 24	__local__
__T21669	{x = -1.4394605, y = -1.8220775, z = -2.150774}	__local__
ray_inv	{x = -1.4394605, y = -1.8220775, z = -2.150774}	__local__
node_index	'node_index' has no value at the target location.	

Call Stack

Name	Language
CUmodule 05508fe0 - [2] trace - Line 148	CUDA
CUmodule 05508fe0 - [1] render_pixel - Line 409	CUDA
CUmodule 05508fe0 - [0] rt_trace_primary_kernel - Line 493	CUDA

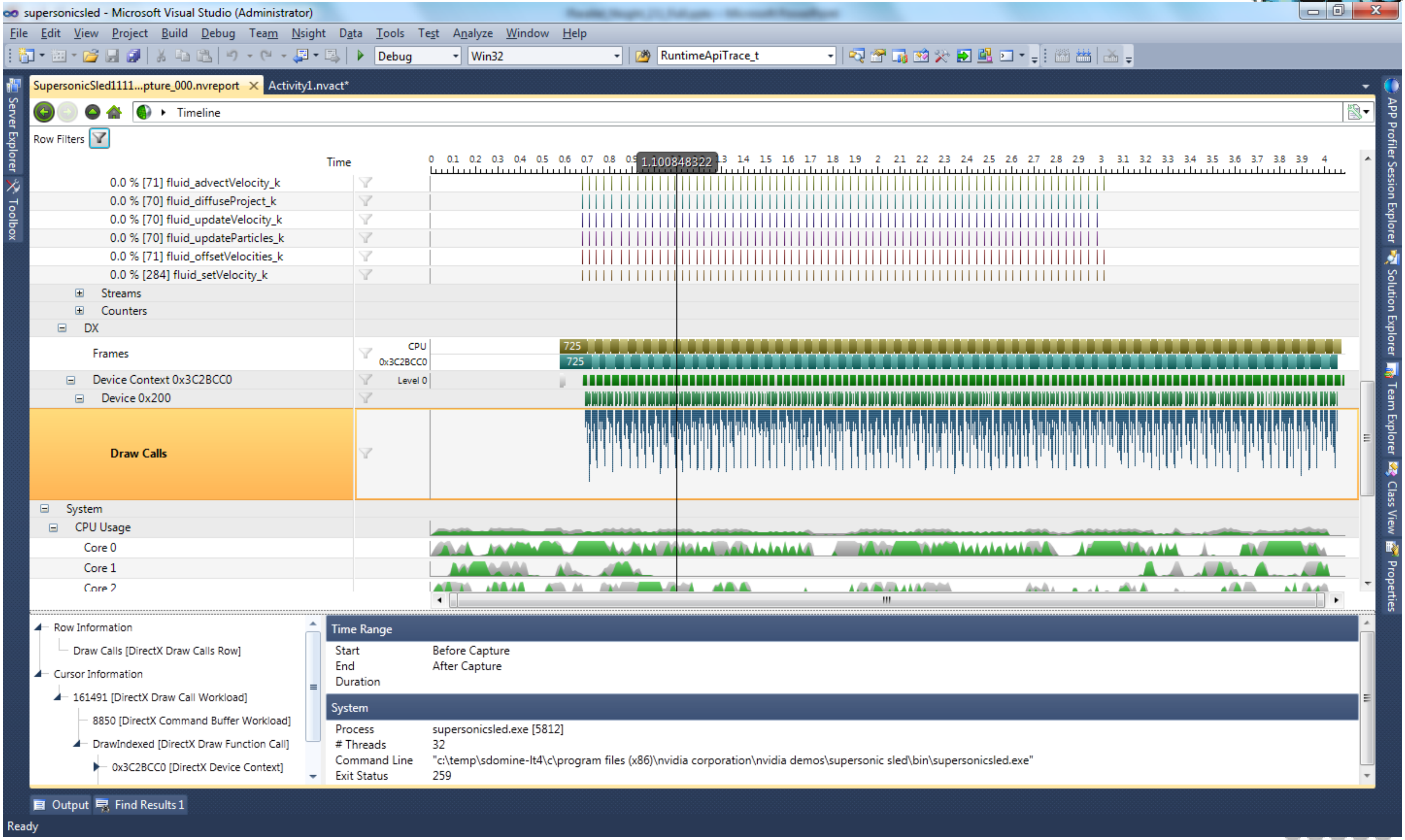
CUDA WarpWatch 1

Name	ray_inv.x	ray_inv.y	ray
Type	__local__ float	__local__ float	__loca
0	-1.4444908	-1.7955524	-2.17
1	-1.44425	-1.7967783	-2.17
2	-1.4440092	-1.7980076	-2.17
3	-1.4437686	-1.7992405	-2.17
4	-1.4435281	-1.800477	-2.17
5	-1.4432876	-1.8017174	-2.17
6	-1.4430474	-1.8029615	-2.17
7	-1.4428074	-1.8042094	-2.16
8	-1.4425675	-1.8054608	-2.16
9	-1.4423276	-1.8067161	-2.16
10	-1.4420878	-1.8079749	-2.16
11	-1.4418485	-1.8092378	-2.16
12	-1.4416089	-1.8105046	-2.16
13	-1.4413697	-1.8117749	-2.16
14	-1.4411306	-1.8130492	-2.16
15	-1.4408917	-1.8143274	-2.15
16	-1.4406527	-1.8156093	-2.15
17	-1.4404141	-1.8168953	-2.15
18	-1.4401754	-1.818185	-2.15
19	-1.439937	-1.8194786	-2.15
20	-1.4396986	-1.820776	-2.15
21	-1.4394605	-1.8220775	-2.15
22	-1.4392225	-1.8233831	-2.14
23	-1.4389844	-1.8246926	-2.14
24	-1.4387469	-1.8260059	-2.14
25	-1.4385092	-1.8273233	-2.14
26	-1.4382718	-1.828645	-2.14
27	-1.4380344	-1.8299706	-2.14
28	-1.4377974	-1.8313001	-2.14
29	-1.4375603	-1.832634	-2.13
30	-1.4373236	-1.8339716	-2.13
31	-1.4370868	-1.8353136	-2.13

CUDA WarpWatch 1 Output

Ready

Parallel NSight


supersonicsled - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Team Nsight Data Tools Test Analyze Window Help
Debug
Win32
RuntimeApiTrace_t
SupersonicSled1111...pture_000.nvreport
Activity1.nvact*
Timeline
Row Filters
Server Explorer
Toolbox
APP Profiler Session Explorer
Solution Explorer
Team Explorer
Class View
Properties
Output
Find Results 1
Ready

Time 1.100848322

0.0 % [71] fluid_advectVelocity_k
0.0 % [70] fluid_diffuseProject_k
0.0 % [70] fluid_updateVelocity_k
0.0 % [70] fluid_updateParticles_k
0.0 % [71] fluid_offsetVelocities_k
0.0 % [284] fluid_setVelocity_k
Streams
Counters
DX
Frames — CPU 725; 0x3C2BCC0 725
Device Context 0x3C2BCC0 — Level 0
Device 0x200
Draw Calls
System
CPU Usage
Core 0
Core 1
Core 2

Row Information
- Draw Calls [DirectX Draw Calls Row]

Cursor Information
- 161491 [DirectX Draw Call Workload]
  - 8850 [DirectX Command Buffer Workload]
  - DrawIndexed [DirectX Draw Function Call]
    - 0x3C2BCC0 [DirectX Device Context]

Time Range

Start	Before Capture
End	After Capture
Duration	

System

Process	supersonicsled.exe [5812]
# Threads	32
Command Line	"c:\temp\sdomine-lt4\c\program files (x86)\nvidia corporation\nvidia demos\supersonic sled\bin\supersonicsled.exe"
Exit Status	259