CUDA Efficient Programming

f.ficarelli@cineca.it

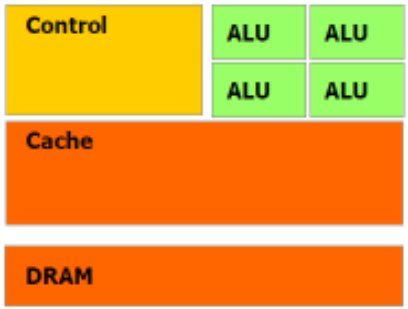
www.cineca.it

Agenda

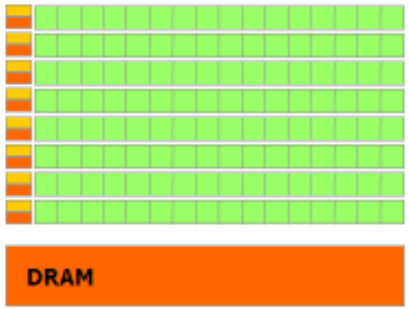
1. Overview and general concepts
2. Performance Metrics
3. Memory Optimizations
4. Execution Optimization
5. Tools Overview

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



CPU



GPU

Maximum performance benefit

- Focus on achieving high occupancy.
- Focus on how to exploit the SIMT model at its best.
- Deeply analyze your algorithm in order to find the hotspots and embarassingly parallel-enabled portions.

  i.e.: pay attention to the Amdahl's law, the porting could be very tough.

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

Capability

The *version tag* that identifies:

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

Capability: atomics

Feature Support(Unlisted features are supported for all compute capabilities)	Compute Capability					
	1.0	1.1	1.2	1.3	2.x, 3.0	3.5
Atomic functions operating on 32-bit integer values in global memory (Atomic Functions)	No	Yes				
atomicExch() operating on 32-bit floating point values in global memory (atomicExch())						
Atomic functions operating on 32-bit integer values in shared memory (Atomic Functions)	No		Yes			
atomicExch() operating on 32-bit floating point values in shared memory (atomicExch())						
Atomic functions operating on 64-bit integer values in global memory (Atomic Functions)						
Warp vote functions (Warp Vote Functions)						
Double-precision floating-point numbers	No			Yes		
Atomic functions operating on 64-bit integer values in shared memory (Atomic Functions)	No				Yes	
Atomic addition operating on 32-bit floating point values in global and shared memory (atomicAdd())						
__ballot() (Warp Vote Functions)						
__threadfence_system() (Memory Fence Functions)						
__syncthreads_count(), __syncthreads_and(), __syncthreads_or() (Synchronization Functions)						
Surface functions (Surface Functions)						
3D grid of thread blocks						
Funnel shift (see reference manual)	No					Yes

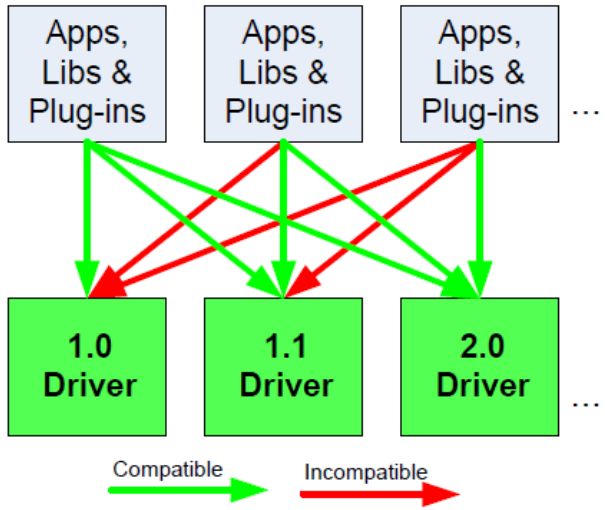
Capability: resources constraints

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		

CUDA APIs

Runtime API	Handles kernel mechanics (loading, parameters setup, invocation, context management, fatbin management). Provides CUDA language extensions.
Driver API	Low level, pure C99 interface (nvcc not needed), explicit management of every resource (context, kernel params, etc…)

The Driver API isn't forward compatible.



Performance metrics

Performance metrics

- Wall time
- Theroetical vs achieved bandwidth
- Achievable vs achieved occupancy
- Memory conflicts

Timing

- It's allowed to use std timing facilities (host side).
- Beware of asynchronous calls!

```
start = clock()
my_kernel<<< g, b, s >>>();
cudaThreadSynchronize();
end = clock();
```

- CUDA provides the *Events* facility.
- Needed to time single streams without loosing concurrency.

```
cudaEventRecord(start, 0);
for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(inputDev + i * size, inputHost + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);
    MyKernel<<<100, 512, 0, stream[i]>>>
              (outputDev + i * size, inputDev + i * size, size);
    cudaMemcpyAsync(outputHost + i * size, outputDev + i * size,
                    size, cudaMemcpyDeviceToHost, stream[i]);
}
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
```

Bandwidth

1. Get board's theoretical bandwidth:

transfer channel width (bits)

DDR

$$B = freq * rate = (1107 * 10^6) * \left(\frac{512 * 2}{8}\right) = 141.6\ GB/s$$

GeForce GTX 280

clock freq. (MHz)

2. Get kernel's effective bandwidth:

```
// __global__ device code, single precision data
if( threadIdx.x < 2048 && threadIdx.y < 2048 ) {
  mat_a[ threadIdx.x ] [ threadIdx.y ] = mat_b[ threadIdx.x ] [ threadIdx.y ];
}
```

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

3. Measure kernel's achieved bandwidth: use profiling tools!

Beware of cudaprof: throughput result is extrapolated and considers wasted transaction data (uncoalesced) as good.

Memory Optimizations

Data Transfers

- Host and Device have their own address space
- GPU boards are connected to host via PCIex 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 raising interrupts);
- 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: locked pages are a limited resource (much smaller than regular pages, `ulimit -l`)

Zero Copy

- CUDA allows to map a page-locked host memory area to 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 larger than 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];
}
```

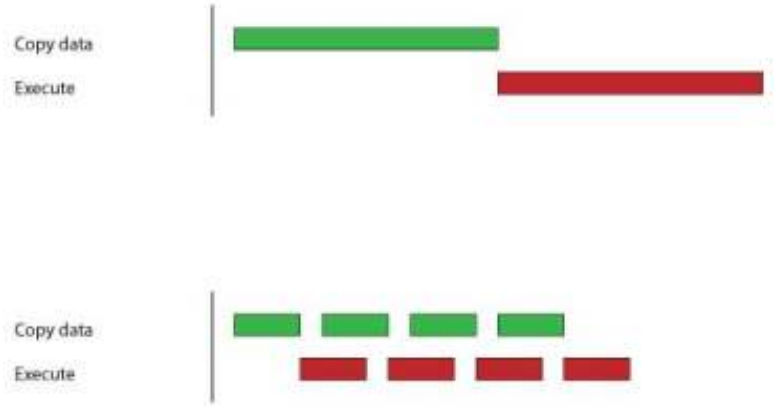
```
cudaDeviceCanAccessPeer(&can_access_peer_0_1, gpuid_0, gpuid_1);
cudaDeviceCanAccessPeer(&can_access_peer_1_0, gpuid_1, gpuid_0);

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 *deferencing pointers!*)
- UVA ensures that the kernel knows whether its argument is from local memory, another GPU or zero-copy from the host

Asynchronous operations

- Kernel calls are asynchronous by default
- Memory transfers and copybacks *are blocking*
- The `cudaMemcpy` has an asynchronous version (`cudaMemcpyAsync`)
- Boards <= 1.3 can overlap *copy-copy (opposite directions)* and *copy-kernel*
- Boards >= 2.0 (Fermi and Kepler) can overlap *kernel-kernel* execution.

```
// First transfer
cudaMemcpyAsync(inputDevPtr, hostPtr, size, cudaMemcpyHostToDevice, 0);
// First invocation
MyKernel<<<100, 512, 0, 0>>> (outputDevPtr, inputDevPtr, size);
// Second transfer
cudaMemcpyAsync(inputDevPtr2, hostPtr2, size, cudaMemcpyHostToDevice, 0);
// Second invocation
MyKernel2<<<100, 512, 0, 0>>> (outputDevPtr2, inputDevPtr2, size);
// Wrapup
cudaMemcpyAsync(hostPtr, outputDevPtr, size, cudaMemcpyDeviceToHost, 0);
cudaMemcpyAsync(hostPtr2, outputDevPtr2, size, cudaMemcpyDeviceToHost, 0);
cudaThreadSyncronize();
```

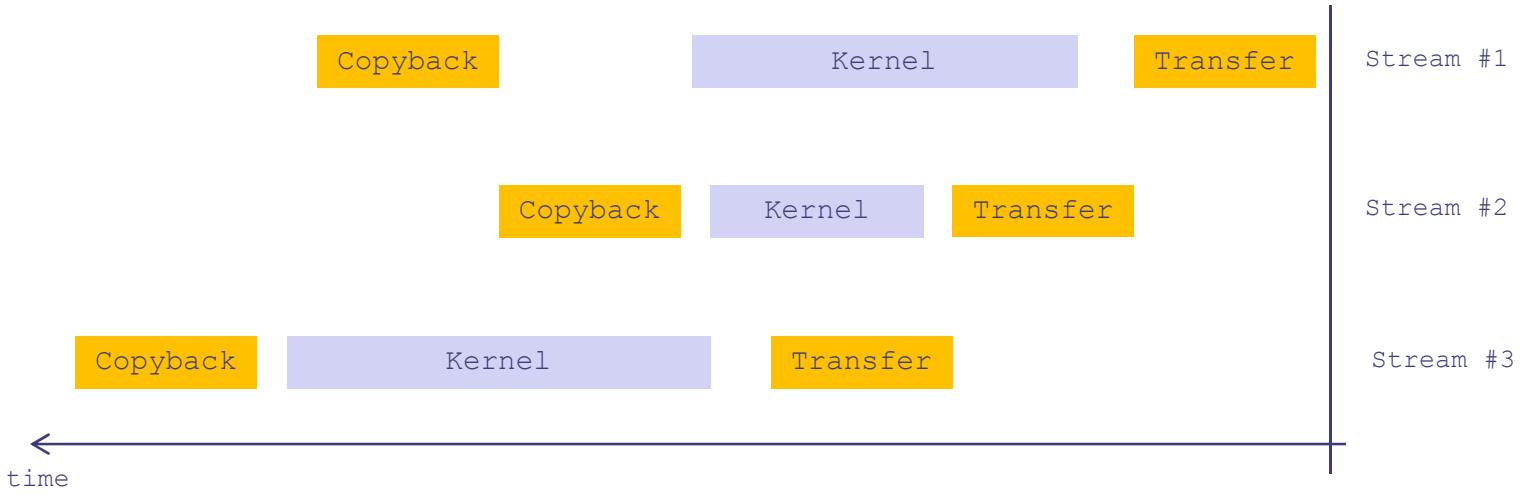


Streams

- A *stream* is a FIFO command queue;
- a stream is independent to every other active stream;
- CUDA streams are the main way to exploit concurrent execution and I/O operations.

CUDA Streams

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

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

for (int i=0; i<3; ++i) cudaMemcpyAsync(inputDevPtr+i*size, hostPtr + i * size, size, cudaMemcpyHostToDevice, stream[i]);

for (int i=0; i<3; ++i) myComputeKernel<<<100, 512, 0, stream[i]>>>(outputDevPtr + i * size, inputDevPtr + i * size, size);

for (int i=0; i<3; ++i) cudaMemcpyAsync(hostPtr + i * size, outputDevPtr+i*size, size, cudaMemcpyDeviceToHost, stream[i]);

cudaThreadSynchronize();

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



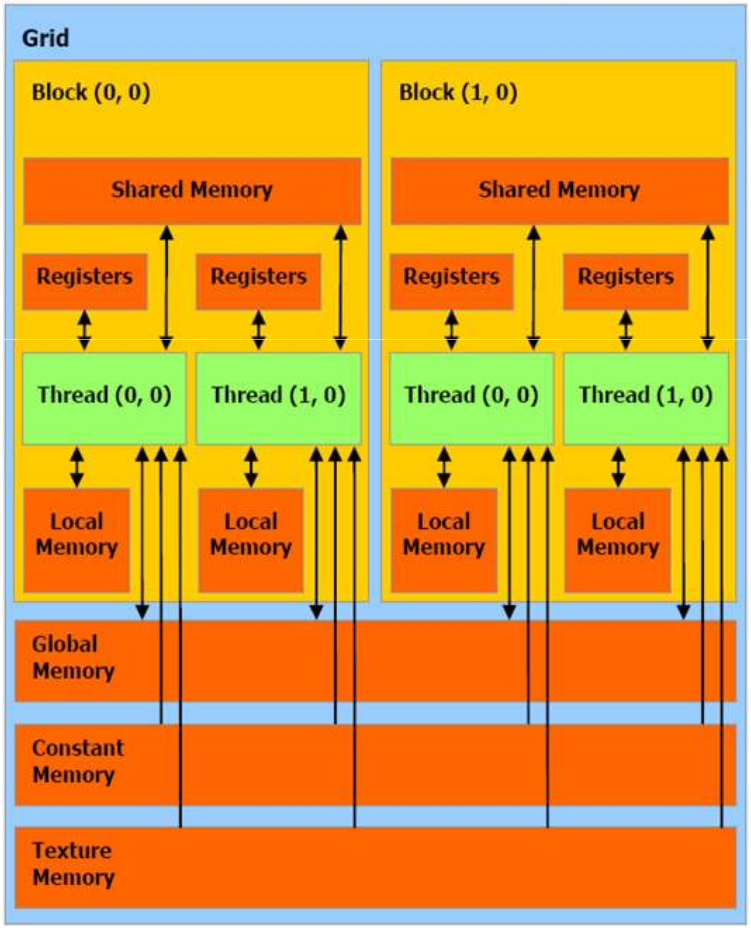
Streams: how to overlap kernels

Starting from capability 2.0 the board has the ability to overlap computations from multiple kernels where:

- submission of commands happens in a breadth-first fashion*;
- no synchronization happens between command stages;
- no operations occur on the default stream;
- the active streams are less than 16*;

*Kepler architecture introduced the *HyperQ* technology:
- No more need for breadth-first command submission
- Supports up to 32 concurrent streams

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

- Memory area with the same purpose as host's main memory;
- High(er) bandwidth, high(er) latency;
- In order to exploit its bandwidth at best, all accesses must be *coalesced*.

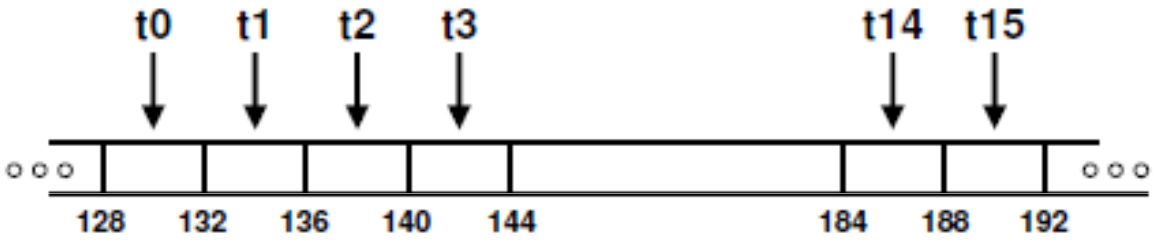
Optimizations : coalescing

The global memory is accessed by 16 threads coalesced if the following three conditions are met:

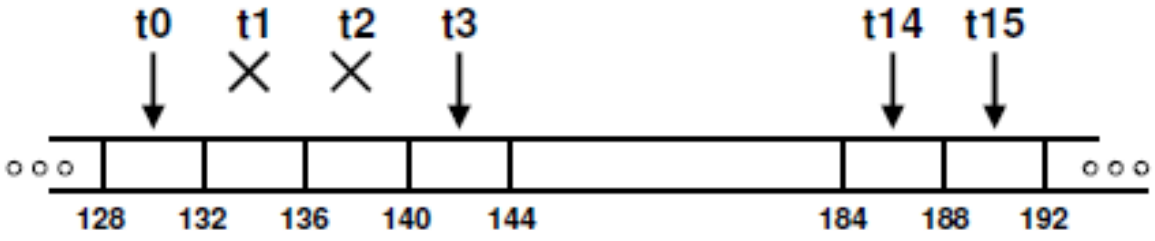
- either 4-byte words, resulting in one 64-byte memory transaction
- Or 8-byte words, resulting in one 128-byte memory transaction
- Or 16-byte words, resulting in two 128-byte memory transactions
- All 16 words must lie in the same aligned segment

- Threads must access the words in a strictly increasing sequence: *the n^{th} thread in the half-warp must access the n^{th} word.*

Optimizations: coalescing, examples: OK

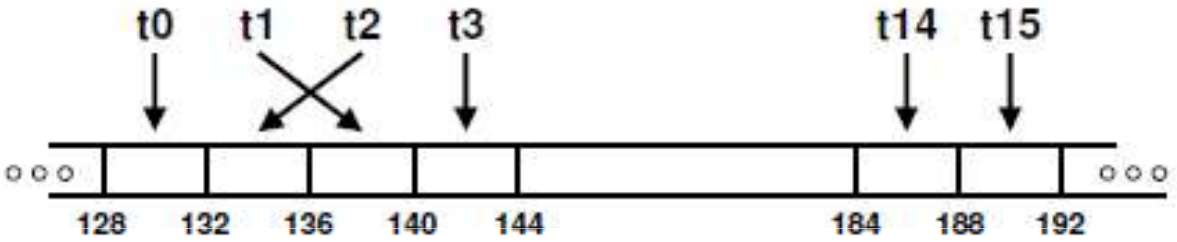


All threads participate



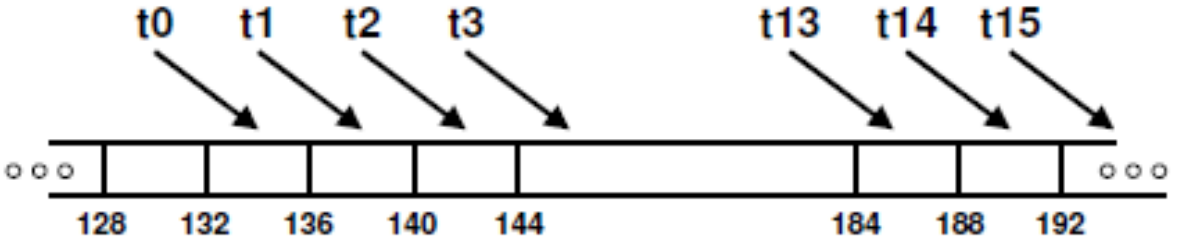
Some Threads Do Not Participate

Optimizations: coalescing, examples: NON-OK



Permuted Access by Threads

Non sequential memory access, resulting in 16 memory accesses

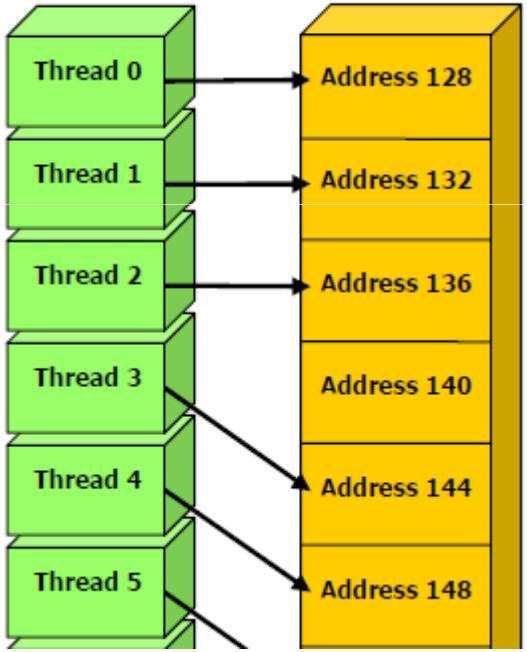


Misaligned Starting Address (not a multiple of 64)

The starting address is misaligned, the result is 16 memory accesses

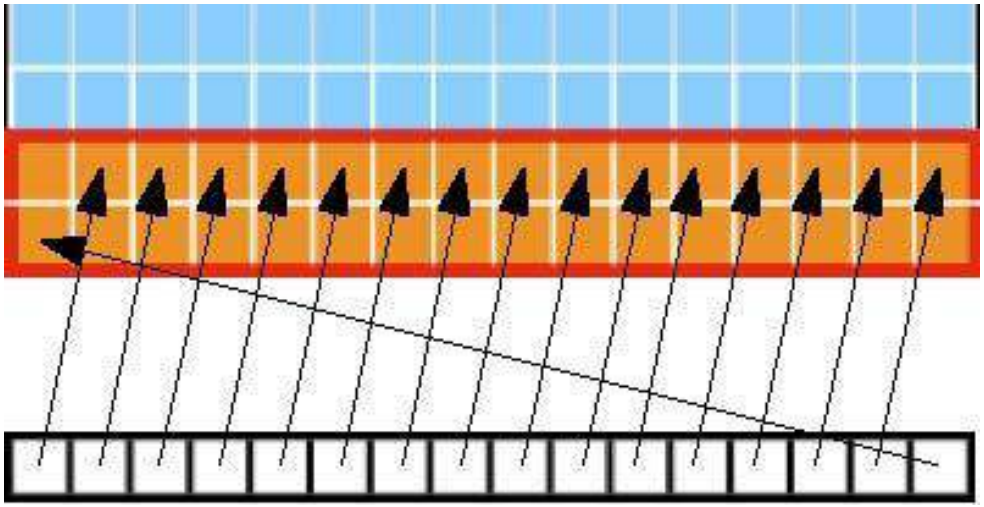
Optimizations: coalescing (5), examples: NON-OK

Non-contiguos
access will result in
16 memory accesses



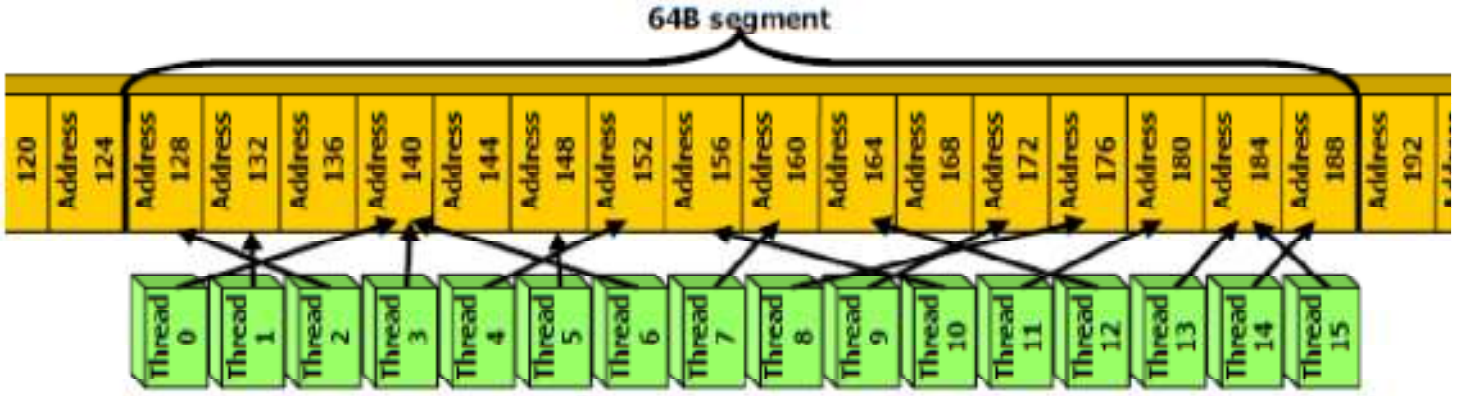
Coalescing for Capability 1.2

The memory controller of 1.2 cards is much improved, access as that in figure would occur in a single transaction



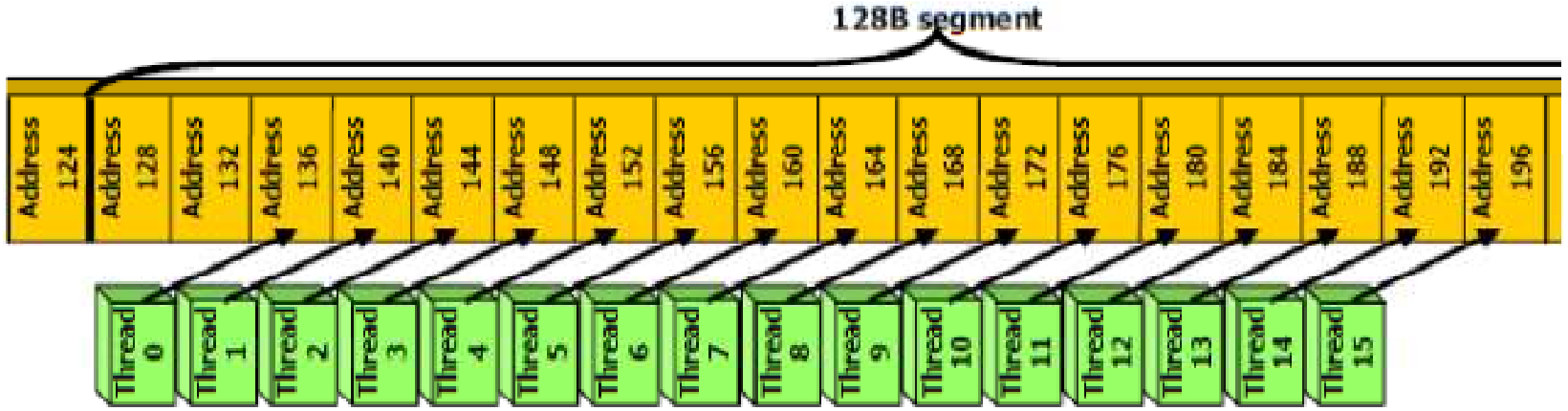
Coalescing for Capability 1.2

Random access within a segment:
single 64B transaction (if they fit)



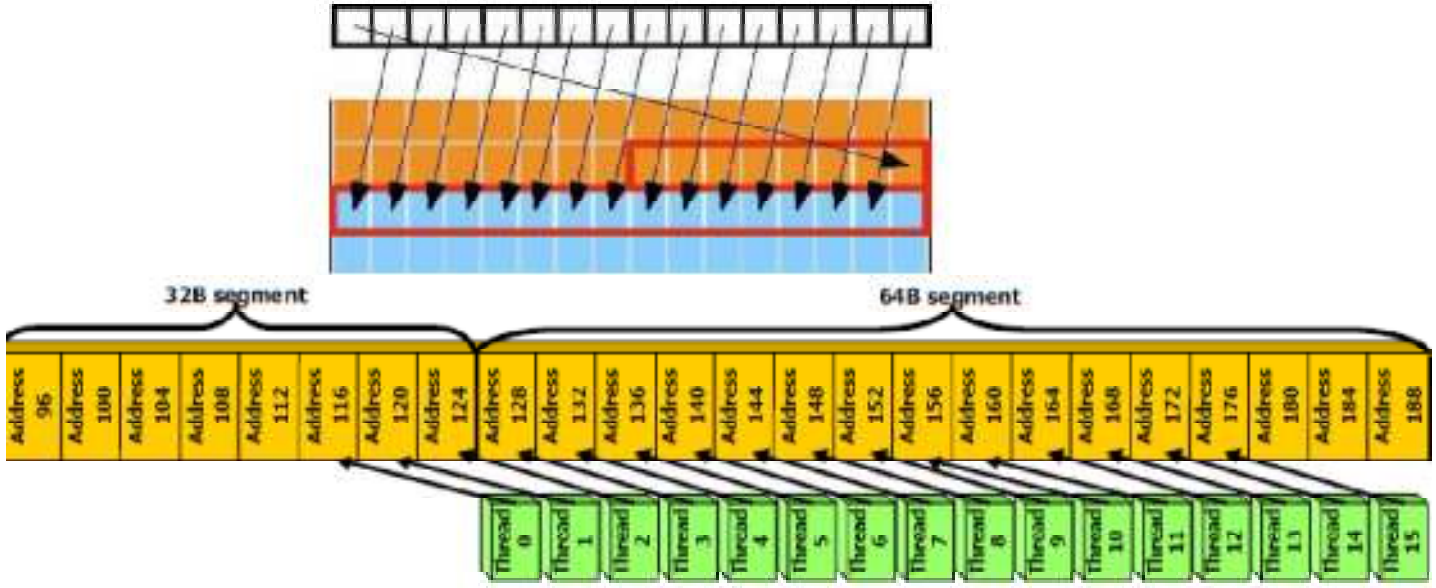
Capability 1.2

Hits misaligned, anyway, occur in a single transaction



Capability 1.2

Misaligned accesses, ending in 2 different 64B segments occur in two transactions



Coalescing for latest architectures

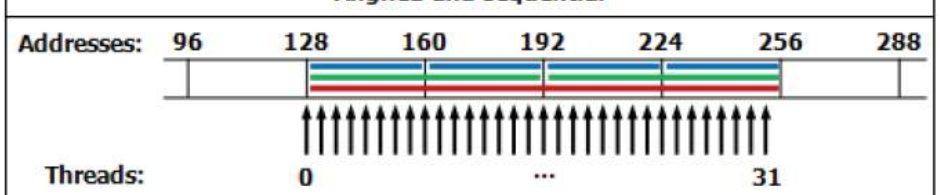
The memory controller has been vastly improved:

On devices with capability >= 2.0, memory accesses by the threads of a warp** are coalesced into the minimum number of L1 lines* that satisfies all threads.

*L1: 128B-aligned, 128B wide lines.
***half-warp* based for previous architectures.

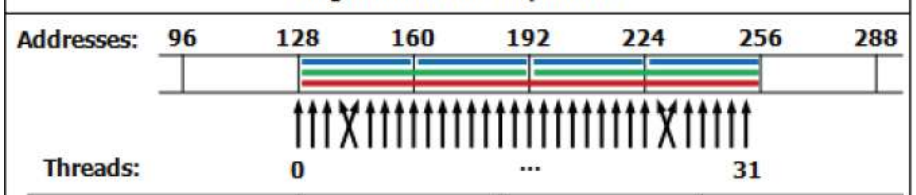
Coalescing: examples

Aligned and sequential



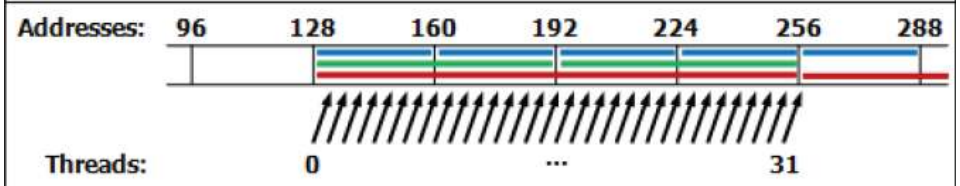
Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.0
Memory transactions:	Uncached		Cached
	1 x 64B at 128 1 x 64B at 192	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

Aligned and non-sequential



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.0
Memory transactions:	Uncached		Cached
	8 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

Misaligned and sequential



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.0
Memory transactions:	Uncached		Cached
	7 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224 1 x 32B at 256	1 x 128B at 128 1 x 64B at 192 1 x 32B at 256	1 x 128B at 128 1 x 128B at 256

Shared memory

- A sort of *explicit* cache;
- resides on the chip so it is *much* faster than the on-board memory;
- size is 16KB (48KB on Fermi by default*)
- 16 (32 for Fermi) banks can be accessed simultaneously by the same warp;
- Banks are organized such that:
  - successive 32-bit words are assigned to successive banks;
  - each bank has 32-bit per cycle bandwidth.

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

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, then the accesses are serialized (groups transactions in conflict-free accesses);

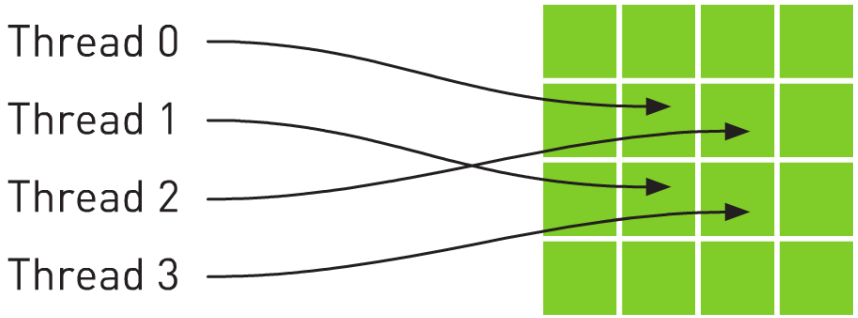
- If all the threads access the same address, a *broadcast* is performed;

- If part of the half-warp accesses the same address, a *multicast* is performed (capability >= 2.0);

Texture memory

- Load requests are cached;
- it is read only, must be set by the host;
- could bring benefits if the threads within the same block access memory using regular 2D patterns, but you need appropriate binding;
- specifically, texture memories and caches are designed for graphics applications where memory access patterns exhibit a great deal of spatial locality;
- dedicated hardware for on-the-fly interpolation.

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] =
tex.filterMode = cu
tex.normalized = tr
// Bind the array t
cudaBindTextureToAr
```

```
// 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 (bandwidths sum).

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
- Useful to off-load long argument lists from shared memory

```
__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 SM pool (32K for Fermi, 64K for Kepler) and statically allocated to each thread;
- *register pressure one of the most dangerous occupancy limiting factors.*

Registers

Some tips:

- try to fold “stack” variables (it would be less useful on LLVM)
- try to offload data to shared memory;
- use launch bounds to force the number of resident blocks;

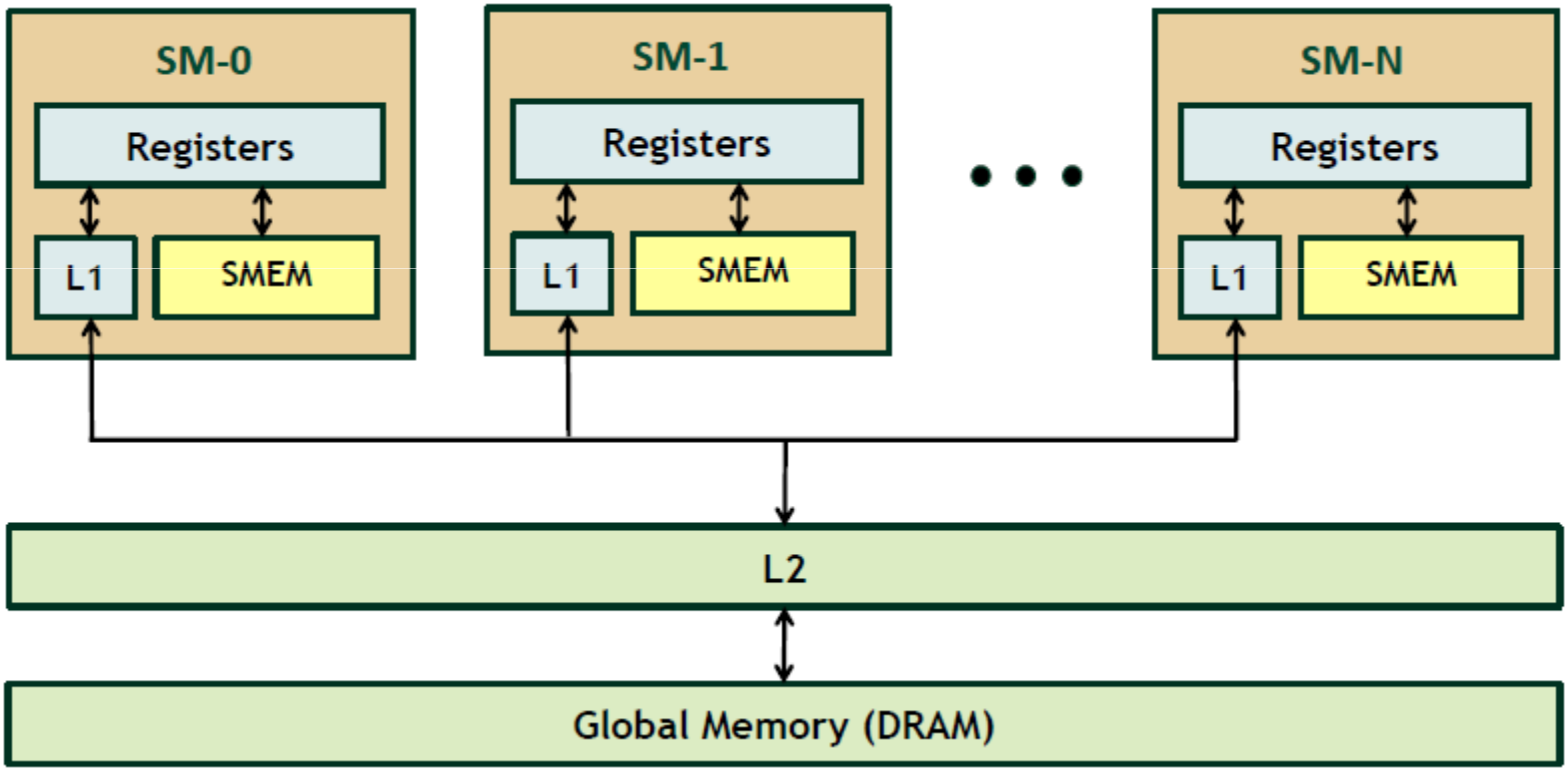
```
#define MAX_THREADS_PER_BLOCK 256
#define MIN_BLOCKS_PER_MP     2

__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 and caches



Local memory

- “Local” because it’s private on a per-thread basis;
- it’s actually a global area used to spill out data when SM runs out of resources;
- addressing is resolved by the compiler;
- cached (store only).

Caches

- FERMI architecture introduces caching mechanisms for global memory accesses (constant and texture are cached since 1.0)
- L1: private to thread, virtual cache implemented into shared memory

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

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

```
// Try to decrease spilled registers eviction from L1,
// disable L1 caching for global memory loads
$ nvcc –Xptas –dlcm=cg
```

*Kepler architecture introduced some improvements:
- New 32 KB + 32 KB partition option

- L2: 768KB, grid-coherent, 25% better latency than DRAM

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;
- grid and block sizes;

  (max threads per SM/max blocks per SM)
- used (and *spilled*) registers

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**.

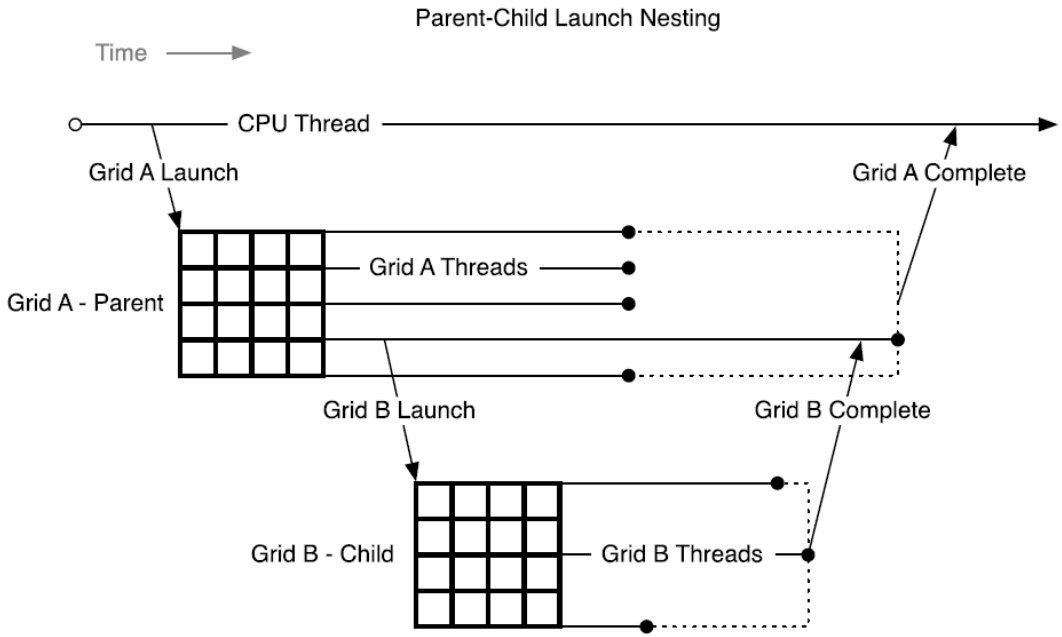
Kepler: dynamic parallelism

- 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 (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

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 the number of execution branches inside the same warp;
- make the compiler's life easier by unrolling loops (hand-coded, pragma or option);
- use signed counters for loops (would be less useful with LLVM);

Exploiting Multi-GPUs

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

1. Every thread can be bound to any board
2. Every board can be bound to an arbitrary number of threads

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
```

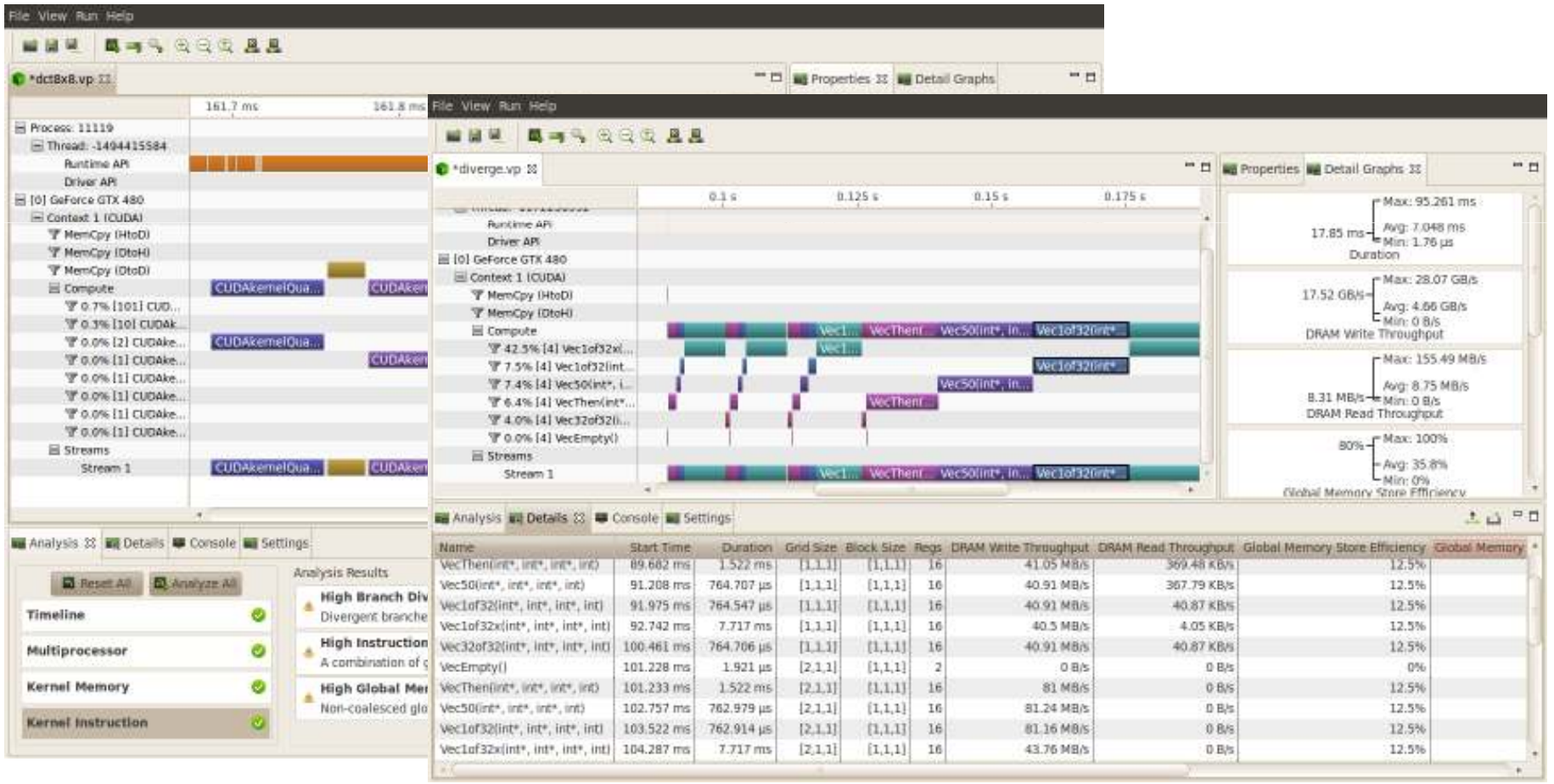
```
// Contents of config
gld_coherent
gld_incoherent
gst_coherent
gst_incoherent
```

```
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)



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
```

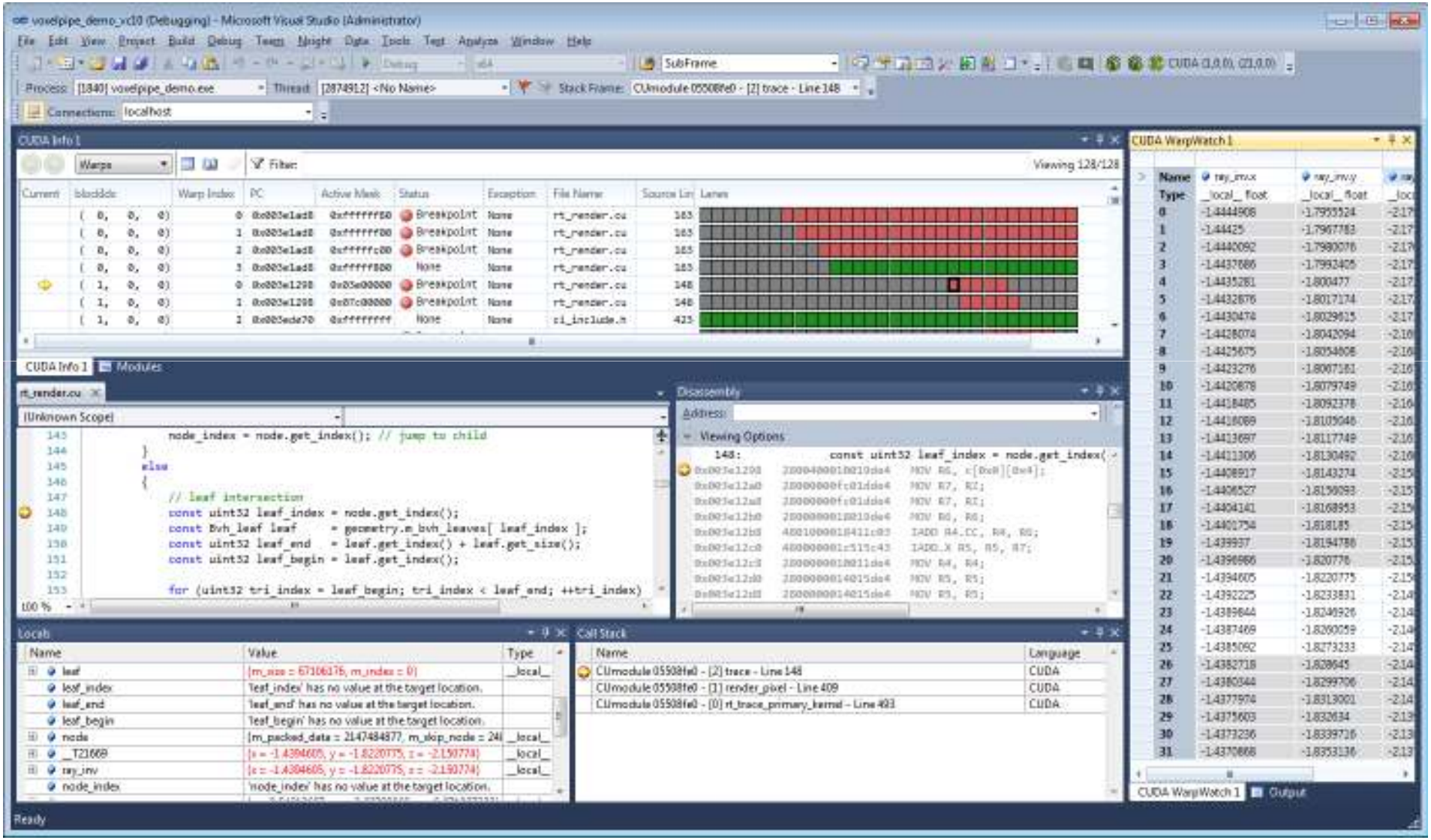
Parallel NSight

- 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

Parallel NSight



Parallel NSight