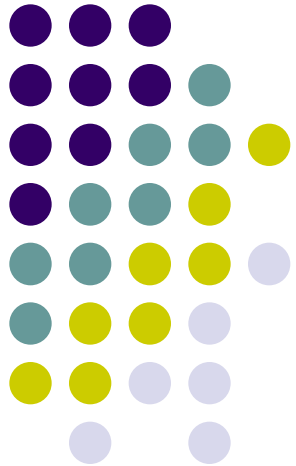
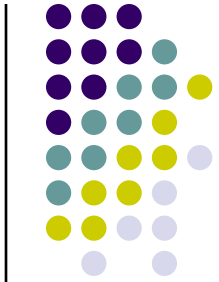
ME964
High Performance Computing for Engineering Applications



Memory Spaces
and Access Overhead

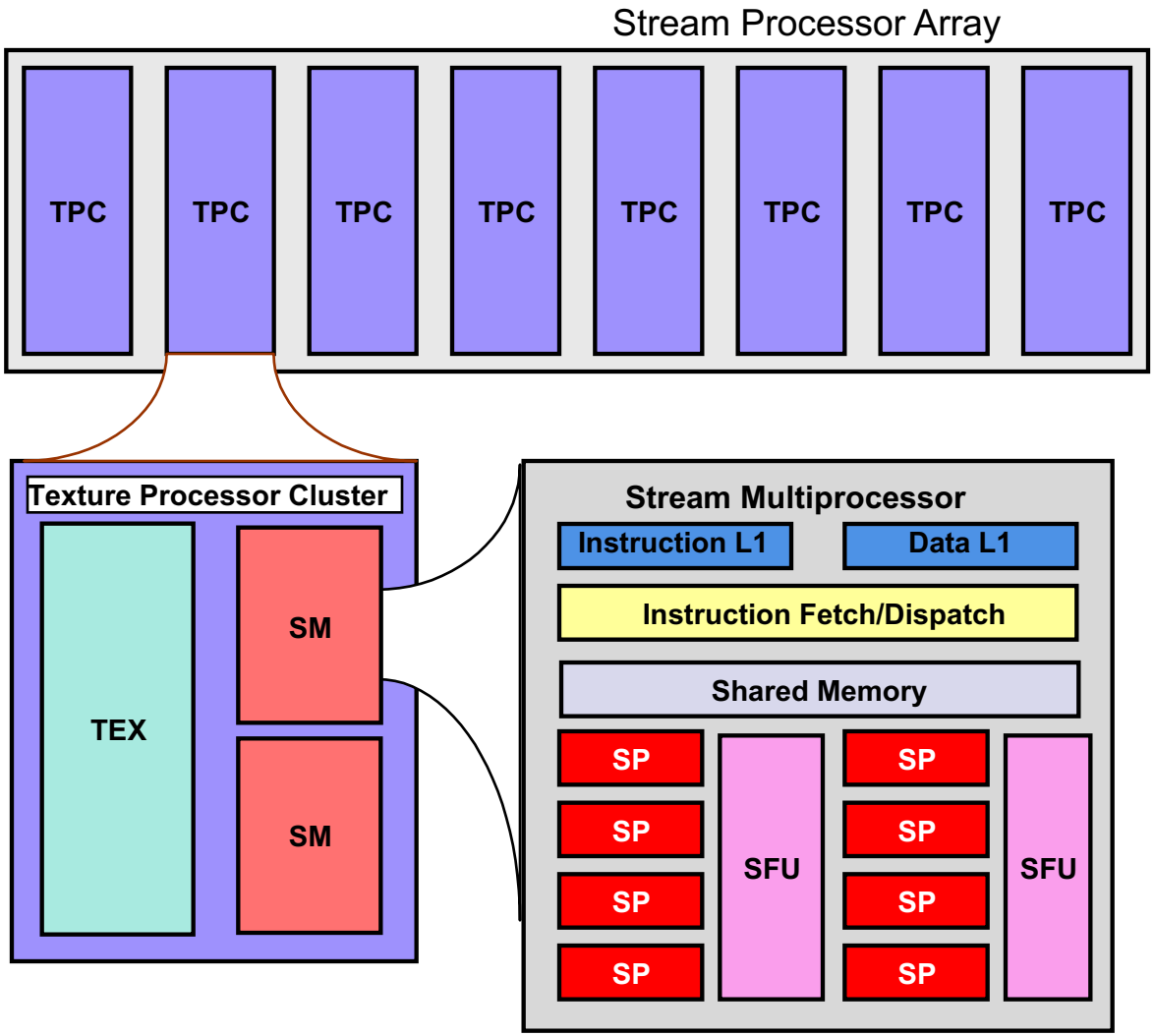
Sept. 30, 2008

Before we get started…



- Last Time
  - The CUDA execution model and the Block/Thread scheduling on the GPU hardware
  - HW4 assigned, increasing difficulty
- Today
  - Details on the CUDA memory spaces and access related overhead
    - Relevant for getting good performance out of your GPU application
- NOTE: CUDA machines in 1235ME are available, post questions on the class forum if you hit a snag

HW Overview

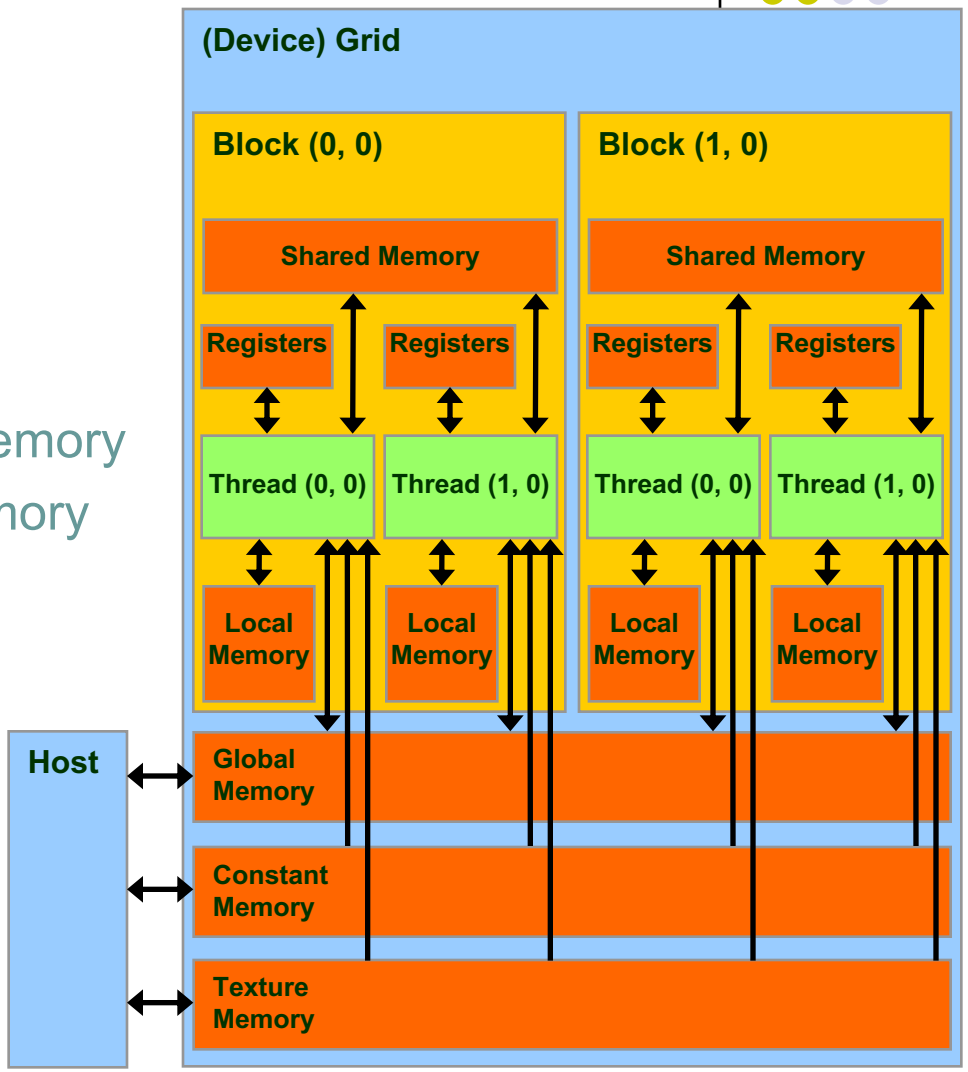

Stream Processor Array
TPC
TPC
TPC
TPC
TPC
TPC
TPC
TPC
Texture Processor Cluster
TEX
SM
SM
Stream Multiprocessor
Instruction L1
Data L1
Instruction Fetch/Dispatch
Shared Memory
SP
SP
SP
SP
SFU
SP
SP
SP
SP
SFU

CUDA Device Memory Space: Review

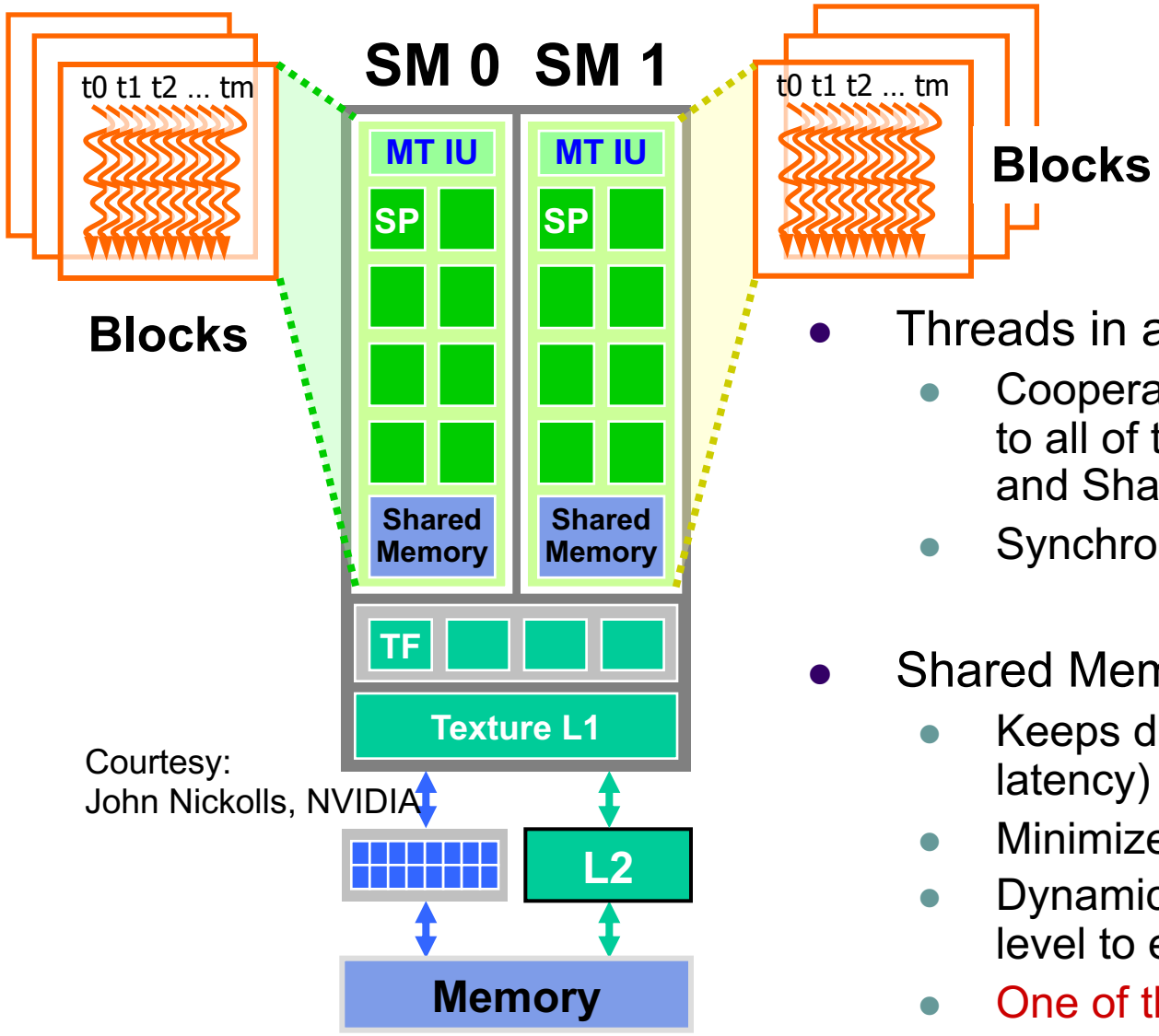
- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory

- The host can R/W global, constant, and texture memories

- Global, constant, and texture memory spaces are persistent across successive kernels calls made by the same application



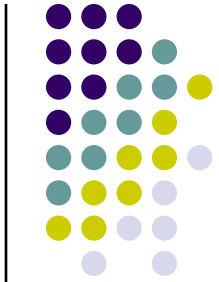
SM Memory Architecture



Courtesy:
John Nickolls, NVIDIA

- Threads in a Block:
  - Cooperate through data accessible to all of them both in Global Memory and Shared Memory
  - Synchronize at barrier instruction
- Shared Memory is very good
  - Keeps data close to processor (low latency)
  - Minimize trips to global memory
  - Dynamically allocated at the SM level to each Block
  - One of the limiting resources

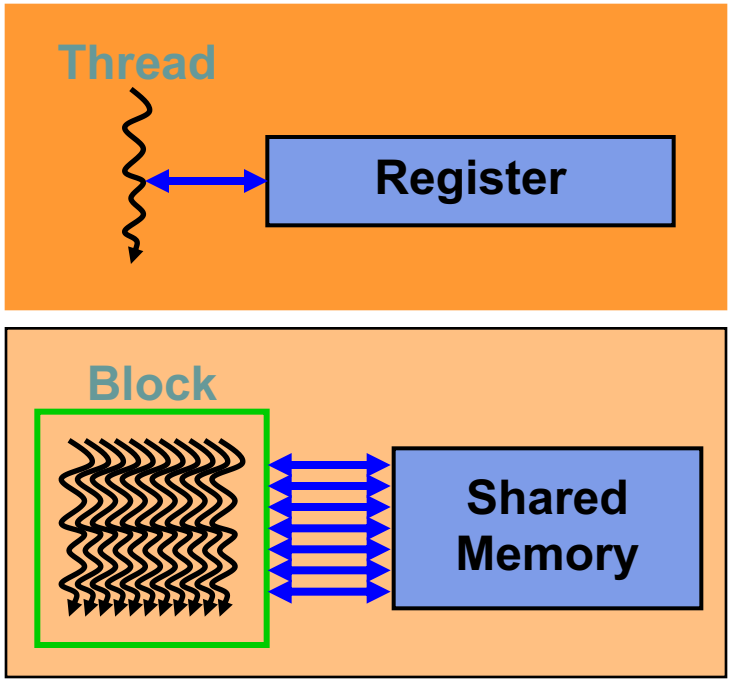
Memory Terminology: Latency vs. Bandwidth



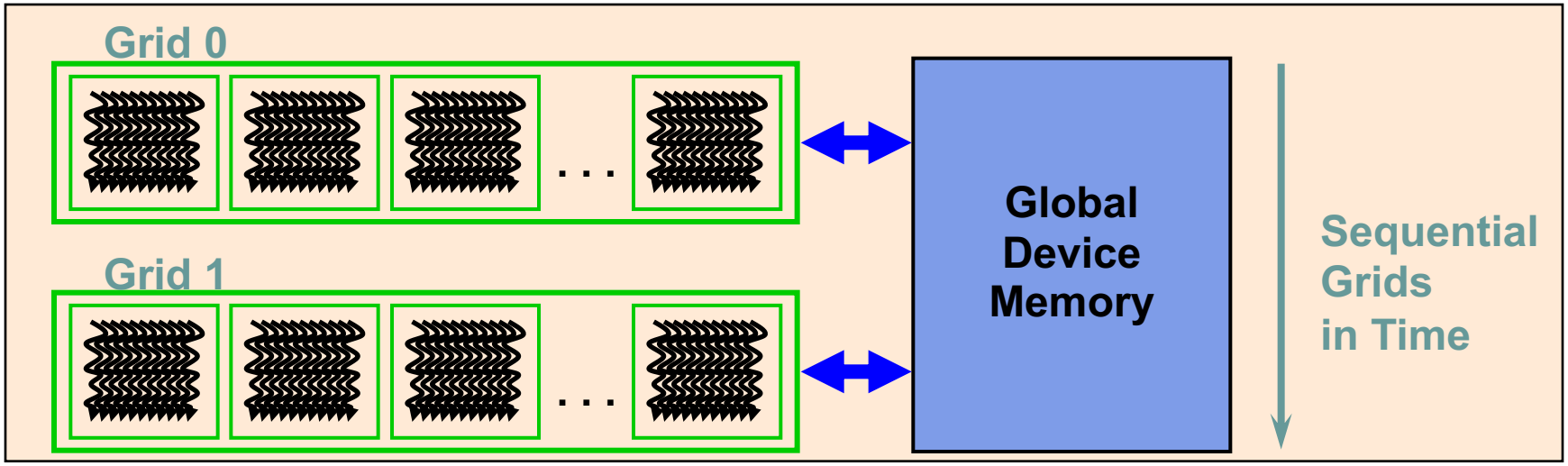
- Latency: you are hungry (for data), place an order, and get served 20 minutes later. The latency is 20 minutes.
  - Latency depends on where the food is stored (memory space):
    - If stored in the kitchen, they'll bring it out fast (on-chip storage)
    - If you ask for mahi-mahi, they have to run to the fish market… (off-chip storage)

- Bandwidth: Once the food starts coming at your table, how much food can you get in one second?
  - This depends on how food is organized in storage and what food you ask for

The Three Most Important Parallel Memory Spaces

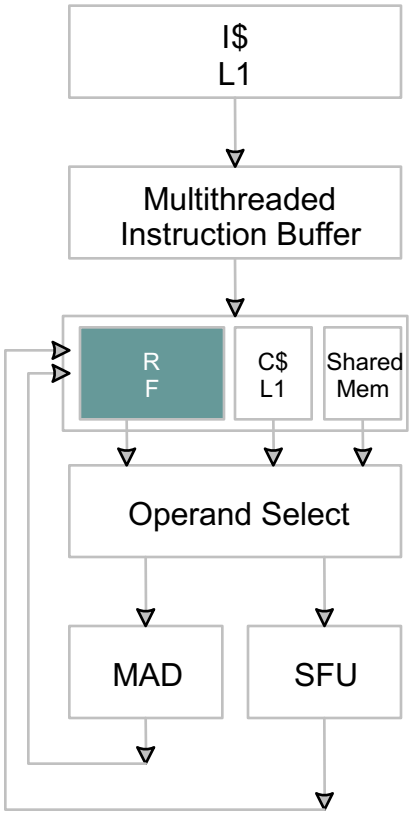


- Register: per-thread basis
  - Private per thread
  - Can spill into local memory (perf. hit)
- Shared Memory: per-Block basis
  - Shared by threads of the same block
  - Used for: Inter-thread communication
- Global Memory: per-application basis
  - Available for use to all threads
  - Also used for inter-grid communication



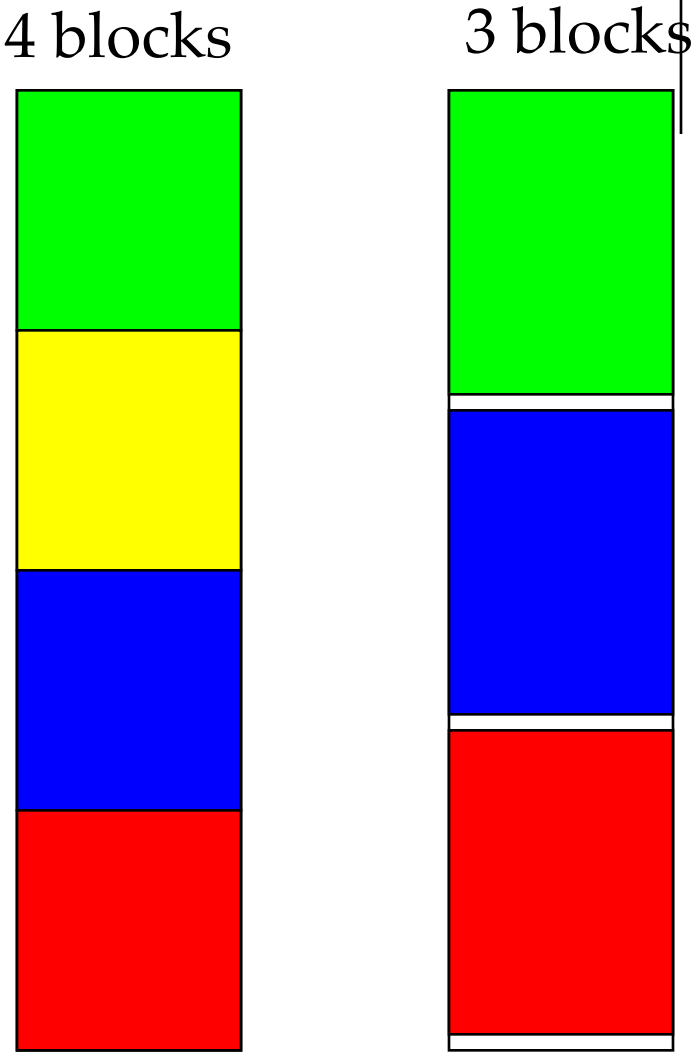
SM Register File (RF)

- Register File (RF)
  - 32 KB (8192 four byte words)
  - Provides 4 operands/clock cycle
- TEX pipe can also read/write RF
  - 2 SMs share 1 TEX
- Global Memory Load/Store pipe can also read/write RF



Programmer View of Register File

- There are 8192 registers in each SM in G80
  - This is an implementation decision, not part of CUDA
  - Registers are dynamically partitioned across all Blocks assigned to the SM
  - Once assigned to a Block, the register is NOT accessible by threads in other Blocks
  - Each thread in the same Block only access registers assigned to itself



Possible per-block partitioning scenarios of the RF available on the SM

Matrix Multiplication Example

- If each Block has 16X16 threads and each thread uses 10 registers, how many threads can run on each SM?
  - Each Block requires 10*256 = 2560 registers
  - 8192 = **3** * 2560 + change
  - So, three blocks can run on an SM as far as registers are concerned

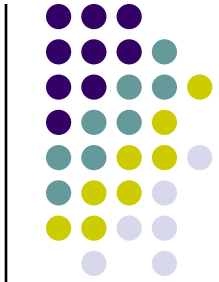
- What if each thread increases the use of registers by 1?
  - Each Block now requires 11*256 = 2816 registers
  - 8192 < 2816 *3
  - Only two Blocks can run on an SM, **1/3 reduction of parallelism**!!!

- This simple example explains why understanding the underlying hardware is essential if you want to squeeze performance out of parallelism

More on Dynamic Partitioning

- Dynamic partitioning gives more flexibility to compilers/programmers
  - One can run a smaller number of threads that require many registers each or a large number of threads that require few registers each
    - This allows for finer grain threading than traditional CPU threading models.
  - The compiler can tradeoff between instruction-level parallelism and thread level parallelism

ILP vs. TLP Example

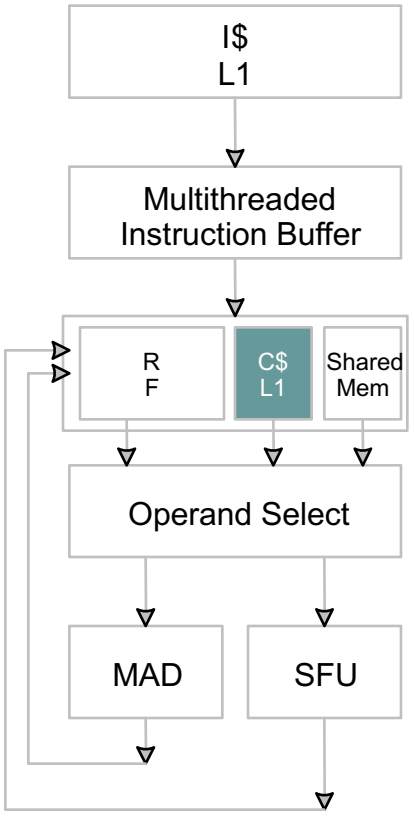


- Assume that a kernel has 256-thread Blocks, 4 independent instructions for each global memory load in the thread program, and each thread uses 10 registers, global loads have 200 cycles
  - 3 Blocks can run on each SM

- If a Compiler can use one more register to change the dependence pattern so that 8 independent instructions exist for each global memory load
  - Only two blocks can now run on each SM
  - However, one only needs 200/(8*4) = 7 Warps to tolerate the memory latency
  - Two Blocks have 16 Warps. The performance can be actually higher!

Constant Memory

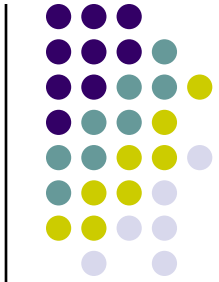
- This comes handy when all threads use the same *constant* value in their computation
  - Example: π, some spring force constant, e=2.7173, etc.
- Constants are stored in DRAM but cached on chip
  - There is a limited amount of L1 cache per SM
  - Might run into slow access if for example have a large number of constants used to compute some complicated formula (might overflow the cache…)
- A constant value can be broadcast to all threads in a Warp
  - Extremely efficient way of accessing a value that is common for all threads in a Block!
  - When all threads in a warp read the same constant memory address this is as fast as a register



```
/*********************************************
Example of how to use the device constant memory
*********************************************/
#include <cuda_runtime.h>
#include <cutil.h>


// Declare the constant device variable outside the body of any function
__device__ __constant__ float dansPI;

// Some dummy function that uses the constant variable
__global__ void myExample()
{
  float circum = 2.f*dansPI*threadIdx.x;
}

// Driver function
int main(int argc, char* argv[])
{
  float somePI = 3.141579f;
  cudaMemcpyToSymbol(&dansPI, &somePI, sizeof(dansPI));

  myExample<<< 1, 256 >>> ();
  return 0;
}
```

Example, Use of Constant Memory

Global Memory



- The global memory space is not cached
  - Very important to follow right access pattern to get maximum memory bandwidth
- Two aspects of global memory access are relevant when fetching data into shared and/or registers
  - The size/alignment of the data you try to fetch from global memory
  - The layout of the access to the global memory (the pattern of the access)

Importance of Size & Alignment

- You can bring data from global to on-chip memory in *one* instruction provided you fetch 32/64/128 bit data that is “properly” aligned

- Example:

```
__device__ TYPE  a[256];
TYPE matAentry = a[tid];
```

  - The load from global to register ends up being one instruction provided the following two conditions hold:
    - TYPE is a C-type for which sizeof(TYPE) returns 4, 8, or 16.
    - The variables of type TYPE are aligned to sizeof(TYPE); i.e., have their address be a multiple of sizeof(TYPE)
      - This condition is automatically satisfied when TYPE is float, float4, int2, etc.

Importance of Size & Alignment

- The situation is trickier when dealing with struct's.
- You can enforce through the compiler the alignment requirements
  - Example: Draws on the definition of three structs Foo, Moo, Boo

```
struct __align__(8) Foo {
        float a;
        float b;
};
```

Compiles to one 8-byte load instruction…

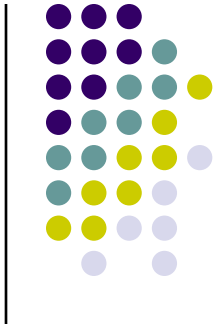
```
struct __align__(16) Moo {
        float a;
        float b;
        int jj;
};
```

Compiles to one 16-byte load instruction…

```
struct __align__(16) Boo{
        float a;
        float b;
        float c;
        float d;
        float e;
};
```

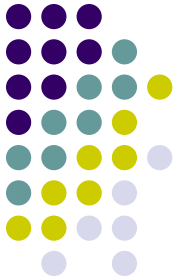
Compiles to two 16-byte load instruction…

The Memory Access Layout



- The important concept here is that of “coalesced memory access”
- The basic idea:
  - Suppose each thread in a half-warp accesses a global memory address for a load operation at some point in the execution of the kernel
  - These threads can access data that is either (a) neatly grouped or (b) scattered all over the place
  - Case (a) is called a “coalesced memory access”; if you end up with (b) this will adversely impact the overall program performance
    - The analogy with the bulldozer with a wide blade

Coalesced Loads and Stores (Cont.)



- Thread number N within the half-warp should access address

  `HalfWarpBaseAddress + N,`

- where:
  - HalfWarpBaseAddress is of type TYPE* and TYPE is such that it meets the Size & Alignment requirements discussed before
  - HalfWarpBaseAddress should be aligned to 16*sizeof(TYPE); i.e., be a multiple of 16*sizeof(TYPE)

```
int main(int argc, char* argv[])
{
  float* Ad;
  cudaMalloc((void**)&Ad, 256*sizeof(float));
  //… code that initializes Ad comes here …
  myExample<<< 1, 256 >>> (Ad);
  return 0;
}
```

```
__global__ void myExample(float* a)
{
  int idx = treadIdx.x;
  float matA_entry = a[idx];
  matA_entry += 1.f;
  a[idx] = matA_entry;
}
```