# CUDA Memory Model

Copyright © 2011 Samuel S. Cho

# Basic CUDA Memory Routines

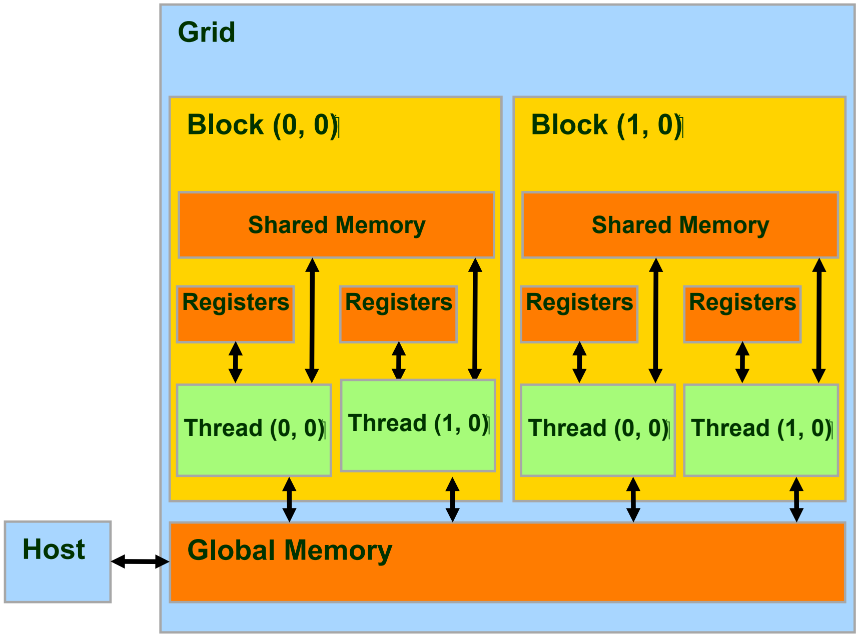
- At the host code level, there are library routines for:
  - memory allocation on graphics card
  - data transfer to/from device memory
  - constants
  - texture arrays (useful for lookup tables)
  - ordinary data
  - etc.

# CUDA Device Memory Allocation

- cudaMalloc()
  - Allocates object in the device global memory
  - Requires two parameters
    - Address of a pointer to the allocated object
    - Size of allocated object
- cudaFree()
  - Frees objects from device global memory
  - Pointer to freed object

# CUDA Memory Model

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



# CUDA Memory Rules

- Currently can only transfer data from host to global (and constant memory) and not host directly to shared.
- Constant memory used for data that does not change (i.e. read-only by GPU)
- Shared memory is said to provide up to 15x speed of global memory
- Registers have similar speed to shared memory if reading same address or no bank conflicts.

# CUDA Memory Lifetimes and Scopes

| Variable Declaration | Memory | Scope | Lifetime |
|---|---|---|---|
| `int RegisterVar;` | register | thread | kernel |
| `__device__ __local__ int LocalVar;`<br>`int ArrayVar[10];` | local | thread | kernel |
| `__device__ __shared__ int SharedVar;` | shared | block | kernel |
| `__device__ int GlobalVar;` | global | grid | application |
| `__device__ __constant__ int ConstantVar;` | constant | grid | application |

- __device__ is optional when used with __local__, __shared__, or __constant__

- Automatic variables without any qualifier reside in a register.
- Except arrays that reside in local memory

- scalar variables reside in fast, on-chip registers
- shared variables reside in fast, on-chip memories
- thread-local arrays and global variables reside in uncached off-chip memory
- constant variables reside in cached off-chip memory

# CUDA Variable Type Scales

| Variable Declaration | Instances | Visibility |
|---|---|---|
| `int RegisterVar;` | 100,000's | 1 |
| `__device__ __local__ int LocalVar;`<br>`int ArrayVar[10];` | 100,000's | 1 |
| `__device__ __shared__ int SharedVar;` | 100's | 100's |
| `__device__ int GlobalVar;` | 1 | 100,000's |
| `__device__ __constant__ int ConstantVar;` | 1 | 100,000's |

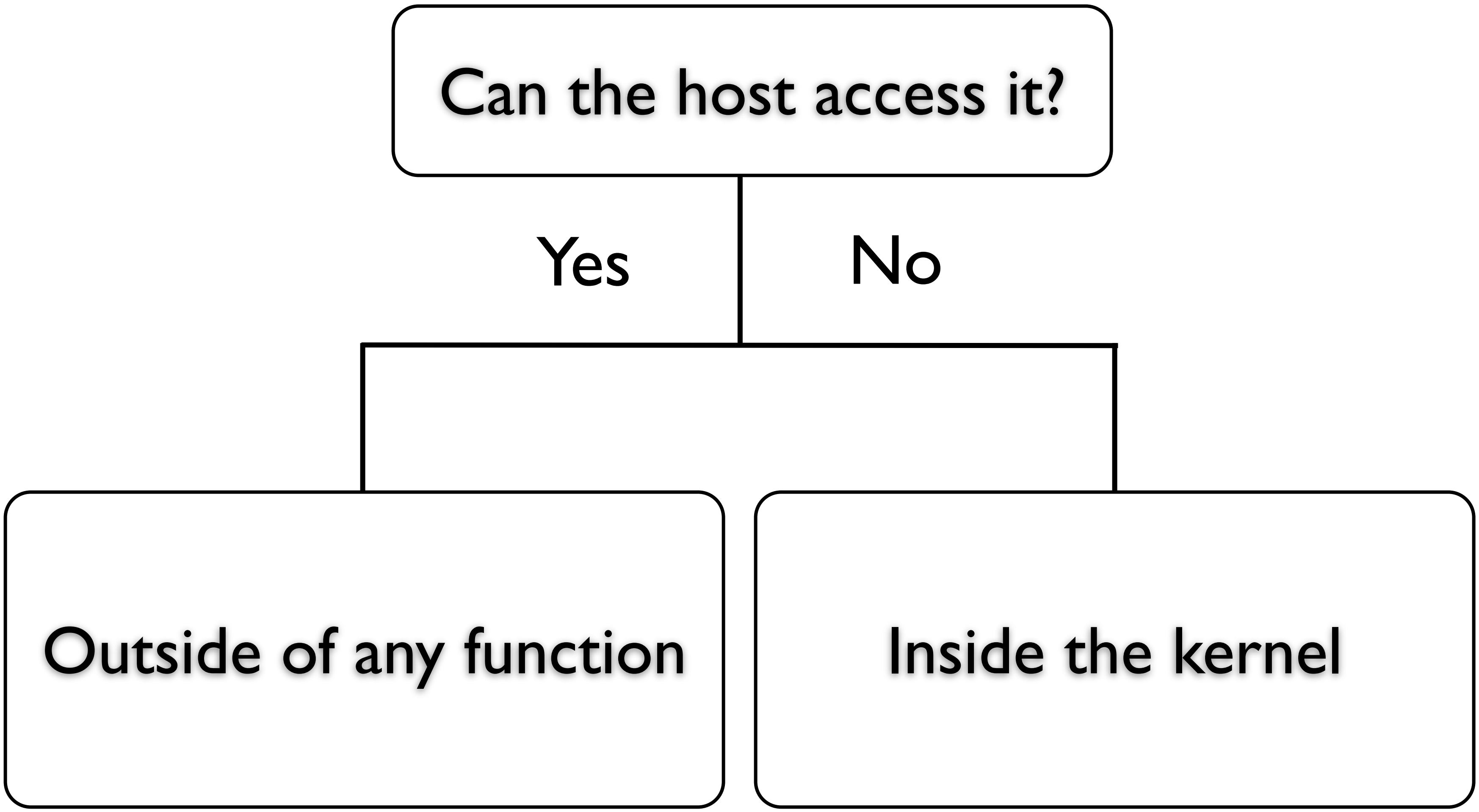
- 100Ks per-thread variables, R/W by each thread.
- 100s shared variables, each R/W by 100s of threads in each block.
- 1 global variable is R/W by 100Ks threads entire device.
- 1 constant variable is readable by 100Ks threads in entire device.

# CUDA Variable Type Performances

| Variable Declaration | Memory | Penalty |
|---|---|---|
| `int RegisterVar;` | register | 1x |
| `__device__ __local__ int LocalVar;`<br>`int ArrayVar[10];` | local | 100x |
| `__device__ __shared__ int SharedVar;` | shared | 1x |
| `__device__ int GlobalVar;` | global | 100x |
| `__device__ __constant__ int ConstantVar;` | constant | 1x |

- scalar variables reside in fast, on-chip registers
- shared variables reside in fast, on-chip memories
- thread-local arrays and global variables reside in uncached off-chip memory
- constant variables reside in cached off-chip memory

# Where to declare variables?




```
__constant__ int ConstantVar;
__device__   int GlobalVar;
```

```
int LocalVar;
int ArrayVar[10];
__shared__ int SharedVar;
```

# Example: Thread Local Variables

```
#define N 1618 // available to all threads in device

__device__ int globalVar; // global variable

__global__ void hello(float2 *ps)
{
  // localVar goes in a register
  int localVar = ps[threadIdx.x];

  // per-thread arrayVar goes in off-chip memory
  int arrayVar[10];

  // magic happens here
}

int main(int argc, char **argv) {

  // ...

}
```

# Example: Shared Variables Motivation

- Global Memory Issues:
  - Long delays, slow.
  - Access congestion.
  - Cannot synchronize accesses.
  - Need to ensure no conflicts of accesses between threads.

```
#define SIZE 628

// compute result[i] = input[i] - input[i-1]
__global__ void adj_diff_naive(int *result, int *input)
{
  // compute this thread's global index
  unsigned int i = threadIdx.x;

  if (i < N)
  {
    // each thread loads two elements from global memory:
    // once by thread i and another by thread i+1
    int x_i = input[i];
    int x_i_minus_one = input[i-1];

    result[i] = x_i - x_i_minus_one;
  }
}
```

- Idea: Eliminate redundancy by sharing data.

# Example: Shared Variables

- Shared memory is on the GPU chip and very fast.
- Separate data available to all threads in one block.
- Declared inside function bodies.
- Scope of block and lifetime of kernel call.
- So each block would have its own array s_data[BLOCK_SIZE].

```
#define BLOCK_SIZE 16

// optimized version of adjacent difference
__global__ void adj_diff(int *result, int *input)
{
  // shorthand for threadIdx.x
  int tx = threadIdx.x;

  // allocate a __shared__ array, one element per thread
  __shared__ int s_data[BLOCK_SIZE];

  // each thread reads one element to s_data
  unsigned int i = blockDim.x * blockIdx.x + tx;
  s_data[tx] = input[i];

  // avoid race condition: ensure all loads complete
  // before continuing
  __syncthreads();

  if (tx < N)
    result[i] = s_data[tx] - s_data[tx-1];
 }
```

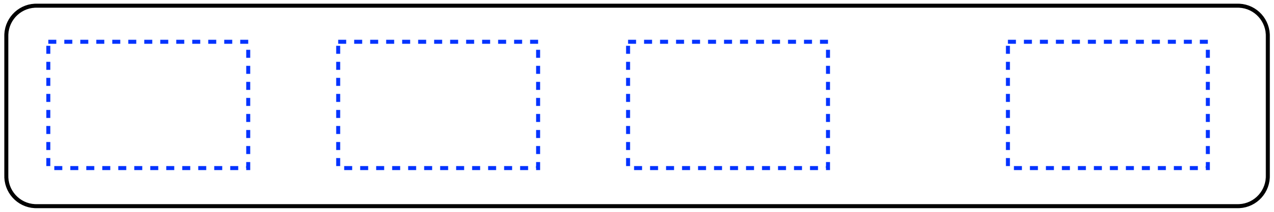
# Shared Variables Issues

- Shared memory is not immediately synchronized after access.
- Usually it is the writes that matter.
- Use __syncthreads() before you read data that has been altered.
- Shared memory is very limited (Fermi has up to 48KB per GPU core, NOT per block)
- Hence may have to divide your data into “chunks”

# Programming Strategy

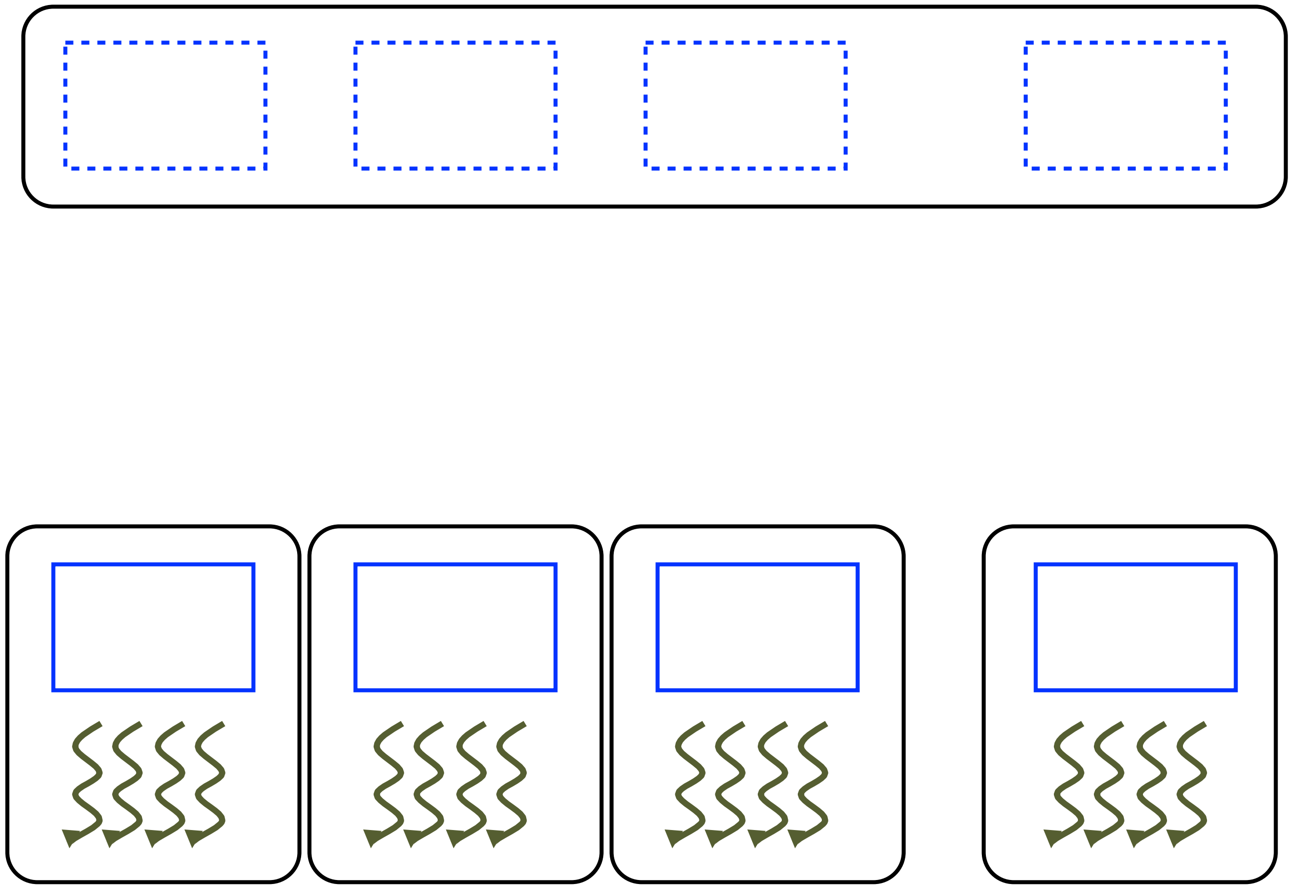
- Global memory (DRAM) is slower than shared memory.
- So, a profitable way of performing computation on the device is to **tile data** to take advantage of fast shared memory:
  - Partition data into subsets that fit into shared memory
  - Handle each data subset with one thread block by:
    - Loading the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism.
    - Performing the computation on the subset from shared memory; each thread can efficiently multi-pass over any data element.
    - Copying results from shared memory to global memory

# Programming Strategy



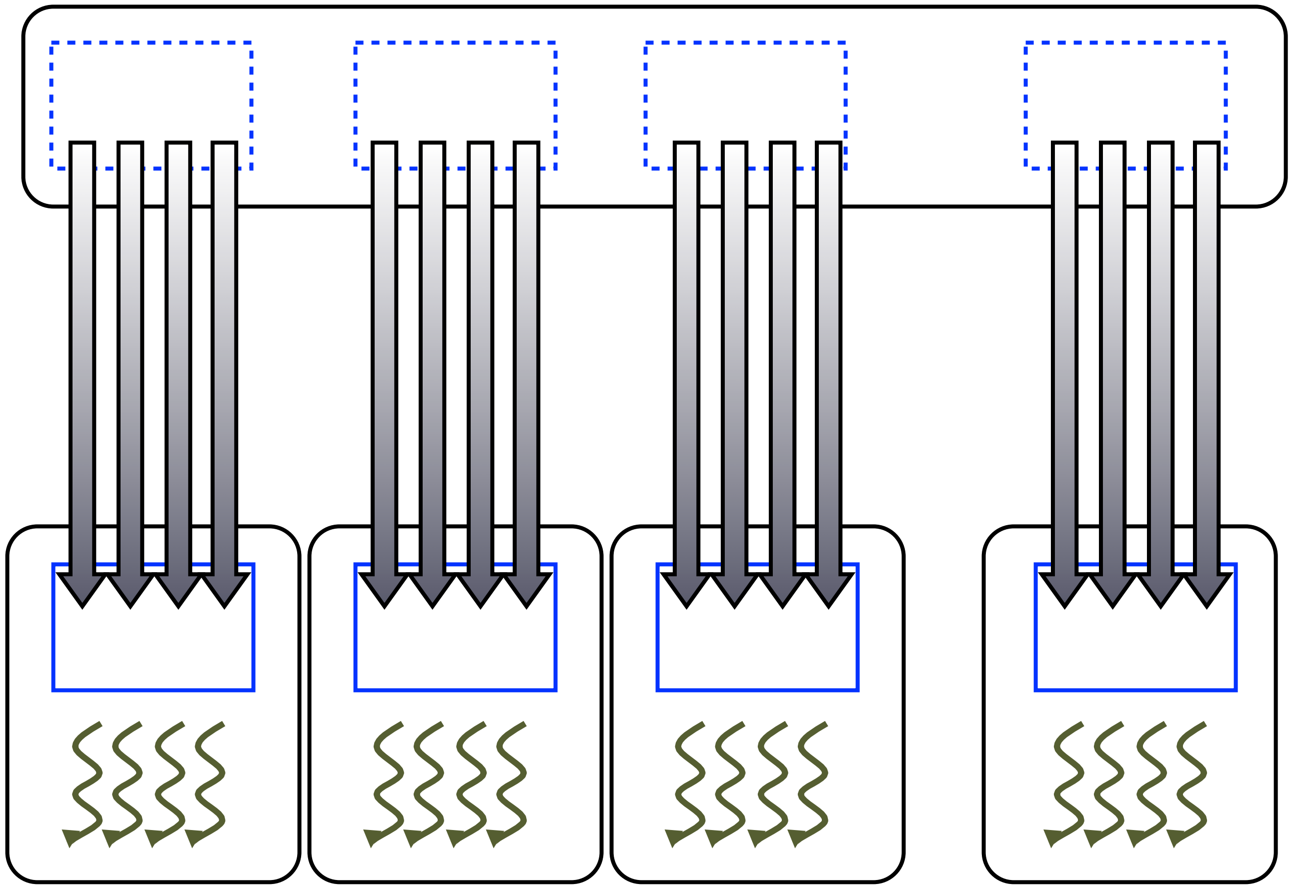
- Partition data into subsets that fit into shared memory

# Programming Strategy



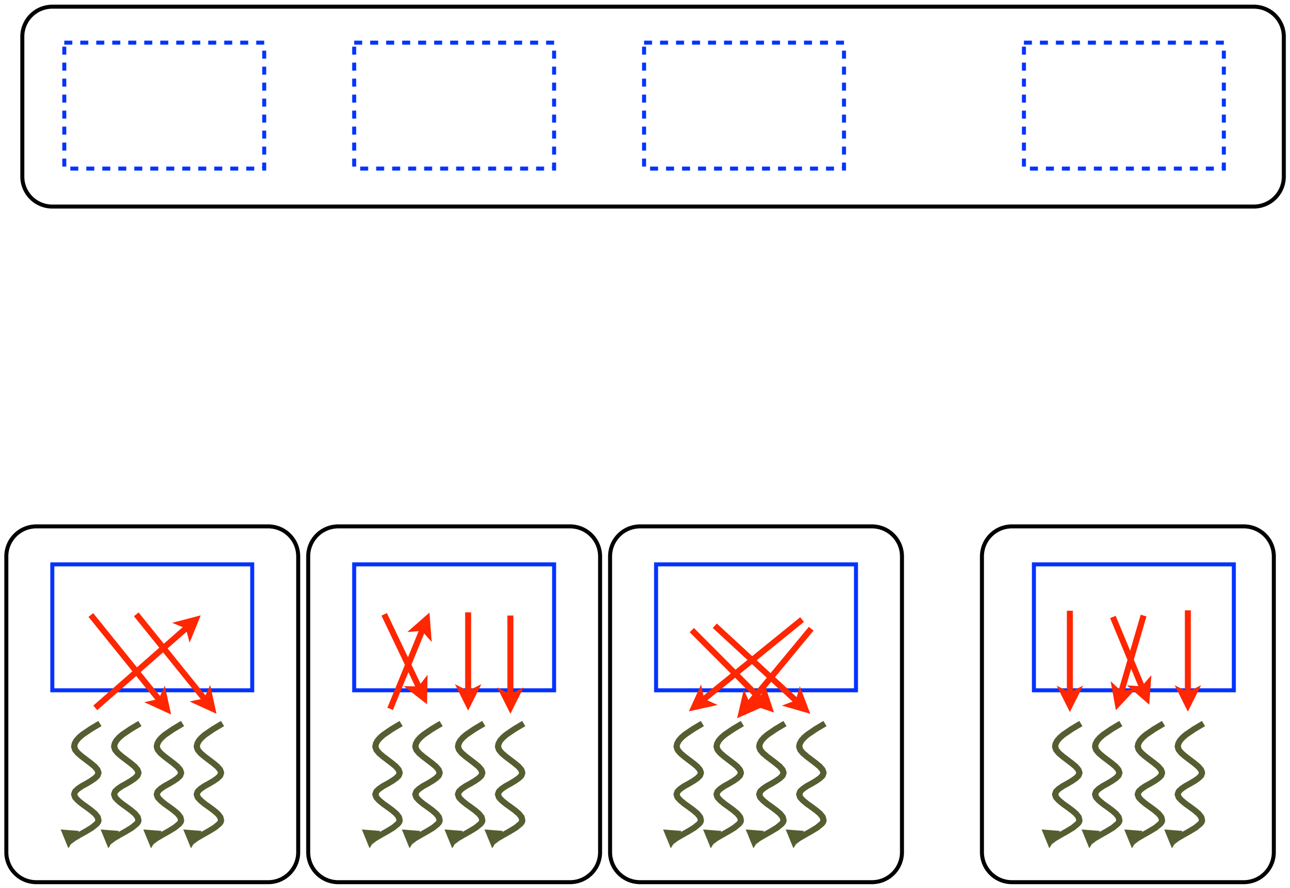
- Handle each data subset with one thread block as follows:

# Programming Strategy



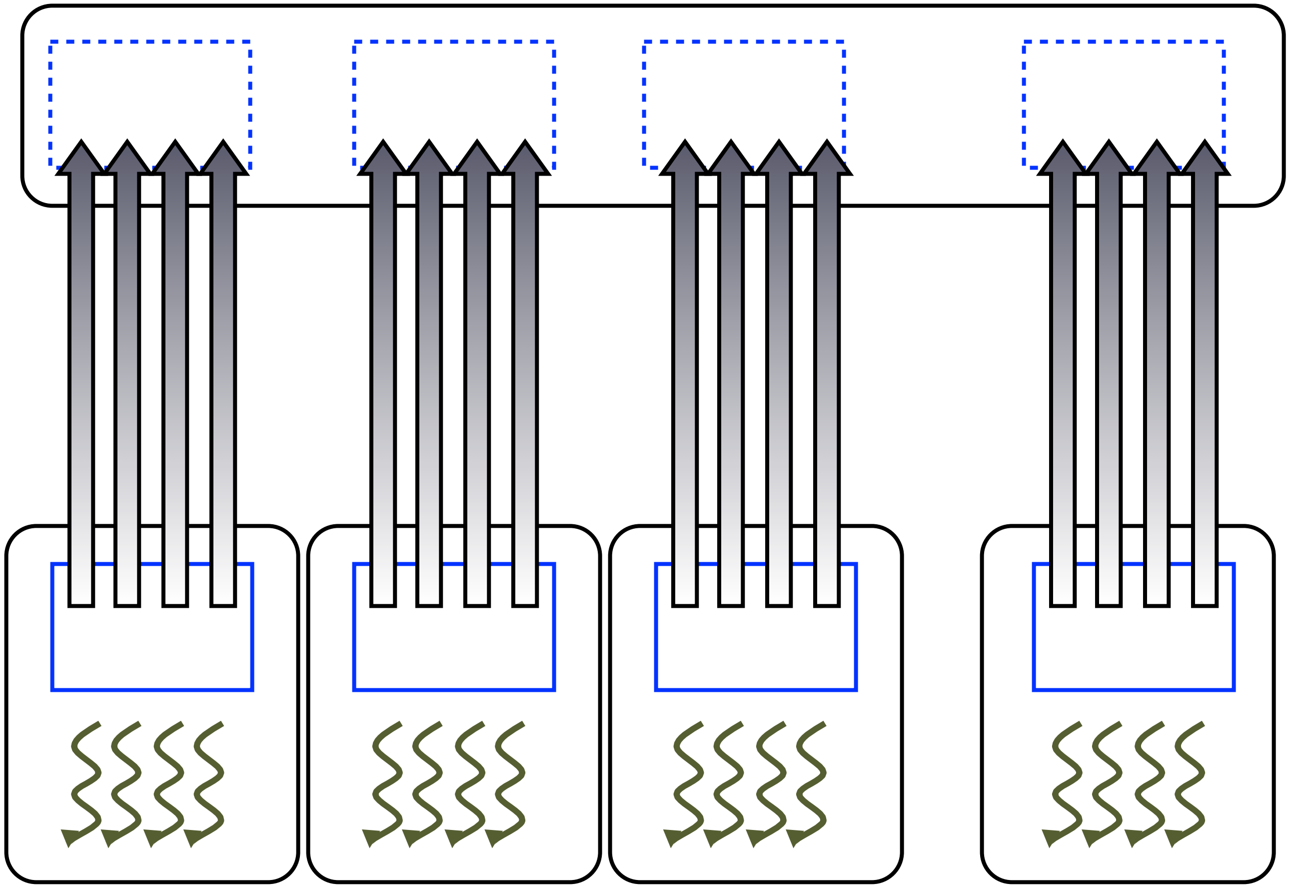
- Loading the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism.

# Programming Strategy



- Perform the computation on the subset from shared memory; each thread can efficiently multi-pass over any data element

# Programming Strategy



- Copy the results from shared memory back to global memory.

# Race Condition

```
__global__ void race(void)
{
  __shared__ int my_shared_variable;
  my_shared_variable = threadIdx.x;
}
```

- The result is undefined.
- The order in which the threads access a variable is not known without explicit coordination.

# Thread Coordination

```
__global__ void share_data(int *input)
{
  __shared__ int data[BLOCK_SIZE];
  data[threadIdx.x] = input[blockDim.x * blockIdx.x + threadIdx.x];
  __syncthreads();
}
```

- The state of the entire data array is now well-defined for all threads in the block.
- Use barriers (e.g., __syncthreads) to ensure data is ready for access.

# Atomics as Barriers

- CUDA provides atomic operations to deal with race conditions.
  - An atomic operation guarantees that only a single thread has access to a piece of memory while an operation completes.
  - The name atomic comes from the fact that it is uninterruptible. (i.e., operations which appear indivisible from the perspective of other threads.)
  - Atomic operations only work with signed and unsigned integers (except AtomicExch)
  - Different types of atomic instructions:
    - Addition/subtraction: atomicAdd, atomicSub
    - Minimum/maximum: atomicMin, atomicMax
    - Conditional increment/decrement: atomicInc, atomicDec
    - Exchange/compare-and-swap: atomicExch, atomicCAS
    - More types in Fermi: atomicAnd, atomicOr, atomicXor

# Atomic Operations

```
// assume *result is initialized to 0

__global__ void sum(int *input, int *result)
{
  atomicAdd(result, input[threadIdx.x]);
}
```

- Use atomic operations (e.g., atomicAdd) to ensure exclusive access to a variable and avoid race conditions.
- An atomic operation is capable of reading, modifying, and writing a value back to memory without the interference of any other threads, which guarantees that a race condition won't occur.
- Atomic operations in CUDA generally work for both shared memory and global memory.
  - Atomic operations in shared memory are generally used to prevent race conditions between different threads within the same thread block.
  - Atomic operations in global memory are used to prevent race conditions between two different threads regardless of which thread block they are in.
- After this kernel exits, the value of *result will be the sum of the inputs.
- Atomic operations are expensive; they imply serialized access to a variable.

# Atomic Histogram Example

```
// Determine frequency of colors in a picture
// colors have already been converted into integers
// Each thread looks at one pixel and increments
// a counter atomically
__global__ void histogram(int* color, int* buckets)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  int c = colors[i];
  atomicAdd(&buckets[c], 1);
}
```

- atomicAdd returns the previous value at a certain address.
- Useful for grabbing variable amounts of data from the list.

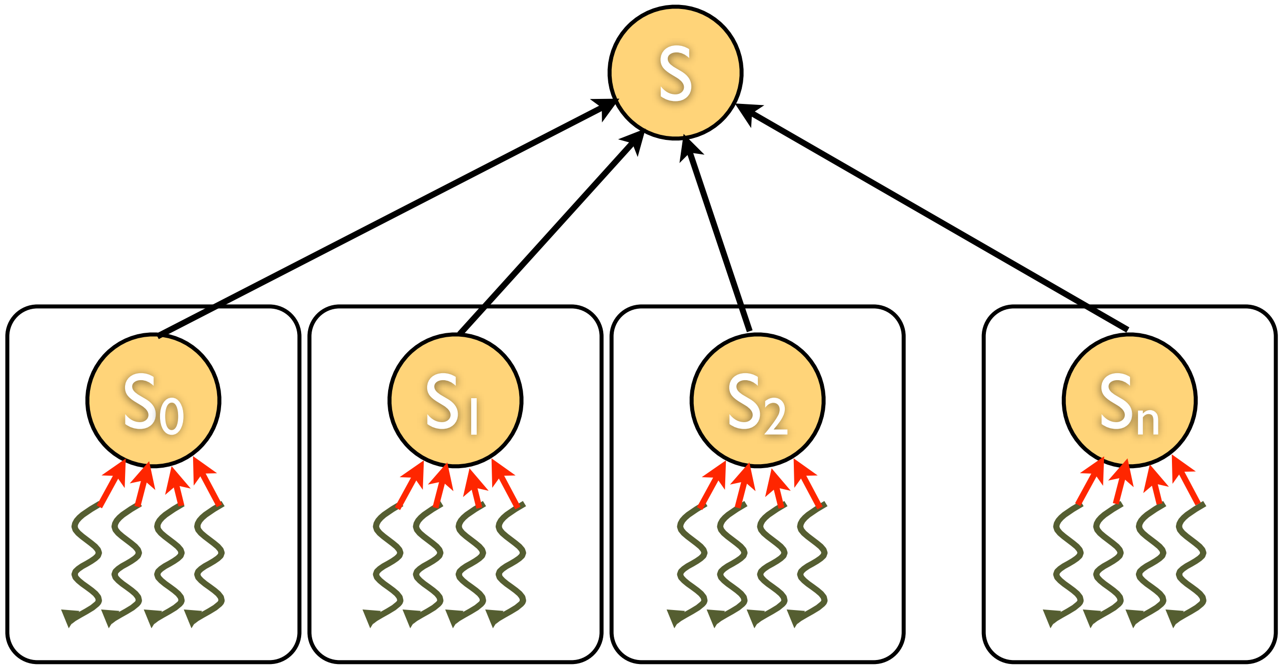
# Compare and Swap

```
int atomicCAS(int* address, int compare, int val)
```

- If <u>compare</u> equals old value stored at <u>address</u> then <u>val</u> is stored at <u>address</u> instead.
- In either case, routine returns the value of old
- Seems a bizarre routine at first sight, but can be very useful for atomic locks.
- Most general type of atomic.

```
int atomicCAS(int* address, int oldval, int val)
{
   int old_reg_val = *address;
   if (old_reg_val == compare) *address = val;
   return old_reg_val;
}
```

# Hierarchical Atomics




- Divide and Conquer
  - Per-thread atomicAdd to a __shared__ partial sum.
  - Per-block atomicAdd to the total sum.

# Hierarchical Atomics

```
__global__ void sum(int *input, int *result)
{
  __shared__ int partial_sum;

  // thread 0 is responsible for initializing partial_sum
  if (threadIdx.x == 0)
    partial_sum = 0;

  __syncthreads();

  // each thread updates the partial sum
  atomicAdd(&partial_sum, input[threadIdx.x]);

  __syncthreads();

  // thread 0 updates the total sum
  if (threadIdx.x == 0)
    atomicAdd(result, partial_sum);
}
```

- Divide and Conquer
  - Per-thread atomicAdd to a __shared__ partial sum.
  - Per-block atomicAdd to the total sum.

# Global Min/Max

```
// If you require the maximum across all threads
// in a grid, you could do it with a single global
// maximum value, but it will be VERY slow
__global__ void global_max_naive(int* values, int* gl_max)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  atomicMax(gl_max,values[i]);
}
```

- Single value causes serial bottleneck.
- Create hierarchy of values for more parallelism.
- Performance will still be slow, so use judiciously.

```
__global__ void global_max(int* values, int* gl_max,
                           int *local_max, int num_local)
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;

   int val = values[i];

   int ilocal = i % num_local;

   int old_max = atomicMax(&local_max[ilocal], val);

   // update global maximum only if new local maximum is found
   if (old_val < val) {
     atomicMax(gl_max, local_max[ilocal]);
   }
}
```

# Atomics Overview

- Atomics are slower than normal load/store.
- Most of these are operations on signed/unsigned integers (floats available for some):
  - quite fast for data in shared memory
  - slower for data in device memory
- Note: You can have the whole machine queuing on a single location in memory.