Exploring the Effect of Block and Grid Size Settings

Chapter 5.3: CUDA Programming Deep Dive

Sen. Inst. Ali M. Azarpour

Fall 2025-26

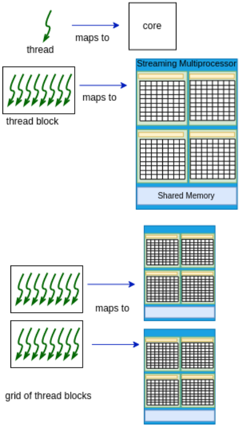
Introduction: Block and Grid Size Configuration

- This section explores different block sizes (number of threads per block) and grid sizes (blocks per grid) configurations
- Different approaches may be more effective depending on the GPU card
- Testing various configurations is important for your particular hardware
- Examples illustrate common code patterns found in CUDA programming

Note: These examples help understand how thread organization affects GPU utilization and performance. The optimal configuration is hardware-dependent and application-specific.

CUDA Programming Model Recap

- Blocks of threads are used on Streaming Multiprocessors (SMs)
- This fundamental model guides how we organize our parallel computations



Refresher: An SM is a physical processing unit on the GPU that executes blocks of threads. Modern GPUs have multiple SMs working in parallel.

Thread-to-Data Mapping

- Previous examples used one mapping approach for threads to array elements
- Different mappings can affect performance and scalability

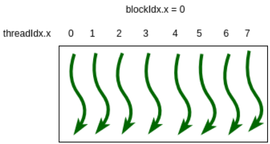
Understanding: The way you map threads to data elements directly impacts how efficiently your GPU resources are utilized. Poor mapping can leave processing power unused.

Vector Addition Example

- This example is designed to show you how you can run an experiment to find out the difference in speed between a CPU core and a single GPU core.
- **Case 1**: We have a host function to add the two arrays on the host CPU
- **Case 2**: We have a kernel function that runs on only one thread on one core of the GPU

Case 3: Single Block of Threads - Overview

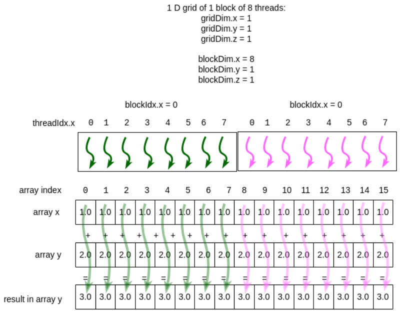
- Simplest case: One block with 8 threads
- Example: Adding 16-element arrays using one block of 8 threads



Limitation: Using only one block means only one SM is utilized, leaving other SMs idle. This severely limits parallelism on modern GPUs with many SMs.

Case 3: Single Block Execution Pattern

- Block of 8 threads (green) works on first 8 elements
- Same block (magenta) then works on next 8 elements
- Process repeats in time steps
- Less parallelism due to sequential time steps



Time-stepping: The block must complete one set of 8 elements before moving to the next set. This is sequential reuse, not true parallelism across all elements.

Case 3: Kernel Function Code

```
// Parallel version that uses threads in the block.
//
// If block size is 8, e.g.
// thread 0 works on index 0, 8, 16, 24, etc. of each array
// thread 1 works on index 1, 9, 17, 25, etc.
// thread 2 works on index 2, 10, 18, 26, etc.
//
// This is mapping a 1D block of threads onto these 1D arrays.
__global__
void add_parallel_1block(int n, float *x, float *y)
{
    int index = threadIdx.x;  // which thread am I in the block?
    int stride = blockDim.x;  // threads per block
    for (int i = index; i < n; i += stride)
        y[i] = x[i] + y[i];
}
```

Key concept: Each thread starts at its thread ID (index) and jumps by blockDim.x (stride) to process multiple elements in a strided pattern.

Case 3: Understanding Index and Stride

- `index = threadIdx.x`: Thread's position within the block (0-7 for 8 threads)
- `stride = blockDim.x`: Number of threads per block (8 in this example)
- Loop pattern: `i = index; i < n; i += stride`
- For 32 elements: 4 time steps, with 8 threads sliding along each time

Example: Thread 0 processes elements 0, 8, 16, 24. Thread 1 processes 1, 9, 17, 25. This strided access pattern continues for all threads.

Case 3: Kernel Call in Main

```
add_parallel_1block<<<1, blockSize>>>(N, x, y); // the kernel call
```

- First parameter (1): Number of blocks in grid
- Second parameter (blockSize): Number of threads per block
- CUDA runtime handles assigning the block to slide along array elements
- Programmer sets up the loop; system manages execution

Important: The $<<< 1, \mathit{blockSize} >>>$ **syntax specifies grid and block dimensions. Here, 1 block limits us to using only one SM on the GPU.**

Case 3: Limitation

- One block runs on one SM on the device
- This limits overall parallelism significantly
- Other SMs remain idle and unused
- Not an optimal use of GPU resources

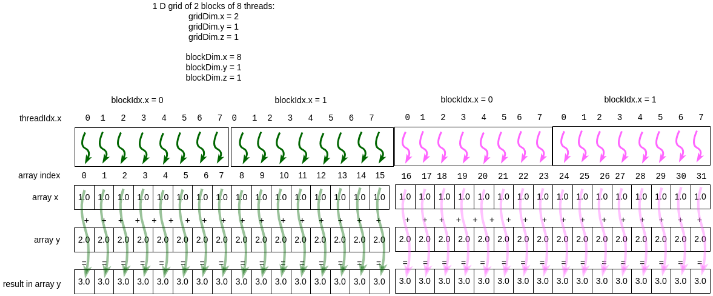
Performance impact: A typical GPU has 10-100+ SMs. Using only one means 90-99% of your GPU sits idle! This approach is rarely used in production code.

Case 4: Multiple Blocks - Introduction

- Using multiple blocks of threads in a 1D grid is most effective for NVIDIA GPUs
- Each block maps to a different streaming multiprocessor
- Enables true parallel execution across multiple SMs
- Different implementation approaches exist for using multiple blocks

Design principle: Maximize SM utilization by creating enough blocks to keep all SMs busy. This is fundamental to achieving good GPU performance.

Case 4: CUDA Programming Model for Multiple Blocks



- Each block in the grid can run on a separate SM
- Multiple blocks enable better utilization of GPU resources
- Foundation for scalable GPU computing

Architecture advantage: Modern GPUs excel at running many blocks simultaneously across their SMs, achieving massive parallelism.

Case 4: Fixed Number of Blocks Example

- Example: 2 blocks of 8 threads each (16 threads total)
- Array size: 32 elements
- First pass: 16 computations in parallel (green threads)
- Second pass: Next 16 computations in parallel (magenta threads)
- If array doubles to 64 elements: Four passes needed

Scalability consideration: As array size increases with fixed grid size, more passes are needed. Each pass is fully parallel, but total time increases.

Case 4: Kernel Function Code

```
// In this version, thread number is its block number
// in the grid (blockIdx.x) times
// the threads per block plus which thread it is in that block.
//
// Then the 'stride' to the next element in the array goes forward
// by multiplying threads per block (blockDim.x) times
// the number of blocks in the grid (gridDim.x).
__global__
void add_parallel_nblocks(int n, float *x, float *y)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
    for (int i = index; i < n; i += stride)
        y[i] = x[i] + y[i];
}
```

Formula breakdown: index gives global thread ID; stride is total threads in grid (blocks $\times$ threads per block).

Case 4: Understanding Global Thread Index

- `index = blockIdx.x * blockDim.x + threadIdx.x`
  - `blockIdx.x`: Which block in the grid (0, 1, 2, ...)
  - `blockDim.x`: Threads per block
  - `threadIdx.x`: Position within the block
- `stride = blockDim.x * gridDim.x`
  - `gridDim.x`: Number of blocks in grid
  - stride = total threads in entire grid

Example: Block 2, Thread 3, with 8 threads/block: index $= 2 \times 8 + 3 = 19$. This thread processes elements 19, 19+stride, 19+2×stride, etc.

Case 4: Kernel Call with Fixed Grid Size

```
// Number of thread blocks in grid could be fixed
// and smaller than maximum needed.
int gridSize = 16;
printf("\n----------- number of %d-thread blocks: %d\n",
       blockSize, gridSize);
t_start = clock();
// the kernel call assuming a fixed grid size and using a stride
add_parallel_nblocks<<<gridSize, blockSize>>>(N, x, y);
```

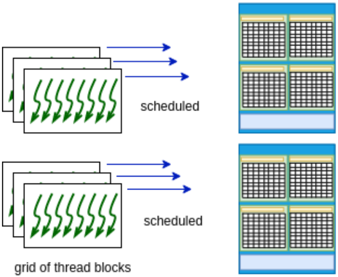
- `gridSize` and `blockSize` are simple integers (not dim3 type)
- Grid size is fixed (16 blocks) regardless of array size

Convenience: For 1D grids/blocks, CUDA allows integer parameters instead of dim3 structures, simplifying code.

Case 4: Modern Scheduling Reality

- The simple depiction may not be strictly accurate for modern GPUs
- CUDA block scheduler has become highly sophisticated
- Fixed grid size with stride often runs nearly as fast as variable grid size
- Each core in an SM can run multiple threads simultaneously
- Hardware scheduler can assign blocks efficiently across SMs

Case 4: Advanced Scheduling Behavior



- One SM can run several concurrent CUDA blocks
- Depends on resources needed by each block
- Hardware scheduler optimizes block placement dynamically

From NVIDIA documentation: SM resource limits (registers, shared memory) determine how many blocks can co-reside on an SM, enabling fine-grained parallelism.

Case 5: Variable Grid Size Method - Introduction

- Grid size computed based on array size and block size
- Adapts to different problem sizes automatically
- Useful when experimenting with different block sizes
- Common pattern in many CUDA examples
- Better theoretical model: map every thread to a specific array index

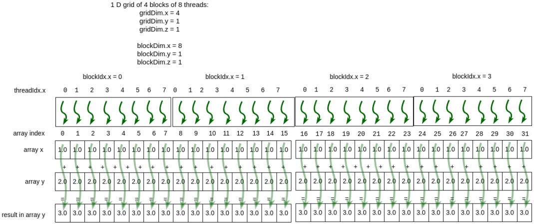
Philosophy: Create exactly as many threads as needed to cover the entire array, enabling one-to-one thread-to-element mapping.

Case 5: Execution Model

- Execution time may be similar to Case 4 (fixed grid)
- Conceptual advantage: All work done in parallel (in theory)
- Create all needed threads upfront
- Map every thread to a particular array index
- One color in diagram = all calculations in parallel

Theoretical vs. practical: While conceptually all work is parallel, hardware limitations (SM count, memory bandwidth) still constrain actual parallelism. The scheduler handles this transparently.

Case 5: Scalability with Array Size



- As array size grows, grid size increases proportionally
- Takes full advantage of GPU architecture
- All elements computed in single parallel operation (conceptually)

Scaling advantage: Doubling array size doubles thread count, maintaining full parallelism regardless of problem size (within GPU limits).

Case 5: Kernel Function Code

```
// Kernel function based on 1D grid of 1D blocks of threads
// In this version, thread number is:
// its block number in the grid (blockIdx.x) times
// the threads per block plus which thread it is in that block.
//
// This thread id is then the index into the 1D array of floats.
// This represents the simplest type of mapping:
// Each thread takes care of one element of the result
//
// For this to work, the number of blocks specified
// times the specified threads per block must
// be the same or greater than the size of the array.
__global__
void vecAdd(float *x, float *y, int n)
{
    // Get our global thread ID
    int id = (blockIdx.x * blockDim.x) + threadIdx.x;

    // Make sure we do not go out of bounds
    if (id < n)
        y[id] = x[id] + y[id];
}
```

Case 5: Key Differences from Case 4

- No `stride` variable used
- No loop within kernel function
- Direct one-to-one mapping: thread ID $=$ array index
- Bounds check required: `if (id < n)`
- Simpler kernel logic
- Grid size calculated to cover entire array

Design tradeoff: Simpler kernel code but potentially more threads created than array elements (due to rounding up to complete blocks). Bounds checking prevents out-of-bounds access.

Case 5: Kernel Call with Computed Grid Size

```
// set grid size based on array size and block size
gridSize = ((int)ceil((float)N/blockSize));
printf("\n----------- number of %d-thread blocks: %d\n",
       blockSize, gridSize);
t_start = clock();
// the kernel call
vecAdd<<<gridSize, blockSize>>>(x, y, N);
```

- Grid size formula: $\lceil N/blockSize \rceil$
- Ensures enough threads to cover entire array
- May create slightly more threads than needed
- Bounds check in kernel prevents errors

Example: N=1000, blockSize=256: gridSize = ceil(1000/256) = 4 blocks = 1024 threads. The extra 24 threads are handled by the bounds check.

Case 5: Handling Very Large Arrays

- As arrays grow very large, grid size may exceed number of SMs
- System automatically handles reassignment
- Cores on SMs are reassigned to new portions of computation
- Happens transparently to the programmer
- No explicit code changes needed

Automatic scheduling: If you have 1000 blocks but only 80 SMs, the scheduler runs blocks in waves. When a block finishes, its SM gets assigned the next pending block. This continues until all blocks complete.

Comparison Summary: Cases 3, 4, and 5

Aspect	**Case 3: Single Block**	**Case 4: Fixed Multiple Blocks**	**Case 5: Variable Grid**
Grid Size	1 block	Fixed (e.g., 16)	Computed from N
Stride	blockDim.x	blockDim.x × gridDim.x	None
Loop in Kernel	Yes	Yes	No
Parallelism	Limited (1 SM)	Better (multiple SMs)	Best (all SMs)
Scalability	Poor	Moderate	Excellent
Code Complexity	Medium	Medium	Simplest
Common Usage	Rare	Common in examples	Most common

Guideline: Case 5 is generally preferred for production code due to automatic scalability and code simplicity, though Cases 3 and 4 remain useful for understanding CUDA concepts.

Key Takeaways

- Multiple configuration strategies exist for organizing CUDA threads
- Single block (Case 3): Simple but severely limits parallelism
- Fixed grid (Case 4): Better parallelism, common in examples
- Variable grid (Case 5): Best scalability, simplest kernel code
- Modern GPU schedulers optimize execution automatically
- Always test configurations on your target hardware
- Grid size calculation is crucial for optimal performance

Practical Considerations

- Block size affects occupancy and resource usage
- Typical block sizes: 32, 64, 128, 256, 512 threads
- Larger blocks: better occupancy but more resource pressure
- Smaller blocks: more flexibility but potential underutilization
- Hardware limits: maximum threads per block (usually 1024)
- Grid size limits: maximum blocks per dimension
- Performance testing essential for optimization