Lecture 6:
Performance Optimization Part 1:
Work Distribution

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Spring 2015
Tunes

The Heavy
Colleen
(Great Vengeance and Furious Fire)

“Colleen? Ha, that wasn’t about a girl. We wrote that one about the dangers of premature program optimization. It burns everyone, and it’s certainly burned me.”

- Kelvin Swaby
Today

- Solver example in the message passing model
- Begin discussing techniques for optimizing parallel programs
- CUDA and GPU programming quick-review
The grid solver in a message passing programming model
One more time... recall the grid-based solver example

Previously expressed using mechanisms from data parallel and SPMD programming models

Update all red cells in parallel

When done updating red cells, update all black cells in parallel (respect dependency on red cells)

Repeat until convergence
Recall: data-parallel solver implementation

- **Synchronization:**
  - `forall` loop iterations are independent (can be parallelized)
  - Implicit barrier at end of outer `forall` loop body

- **Communication**
  - Implicit in loads and stores (like shared address space)
  - Special built-in primitives: e.g., `reduce`

```c
int n;
float* A = allocate(n+2, n+2);

void solve(float* A) {
    bool done = false;
    float diff = 0.f;
    while (!done) {
        for_all(red cells (i,j)) {
            float prev = A[i,j];
            reduceAdd(diff, abs(A[i,j] - prev));
        }
        if (diff/(n*n) < TOLERANCE)
            done = true;
    }
}
```

Example from: Culler, Singh, and Gupta
Recall: shared address space implementation with explicit synchronization (locks and barriers)

```c
int n;                              // grid size
bool done = false;
float diff = 0.0;
LOCK myLock;
BARRIER myBarrier;

// allocate grid
float* A = allocate(n+2, n+2);
void solve(float* A) {
    float myDiff;
    int threadId = getThreadId();
    int myMin = 1 + (threadId * n / NUM_PROCESSORS);
    int myMax = myMin + (n / NUM_PROCESSORS);

    while (!done) {
        float myDiff = 0.0;
        diff = 0.0;
        barrier(myBarrier, NUM_PROCESSORS);
        for (j=myMin to myMax) {
            for (i = red cells in this row) {
                float prev = A[i,j];
                                A[i+1,j], A[i,j+1]);
                myDiff += abs(A[i,j] - prev));
            }
        }
        lock(myLock);
        diff += myDiff;
        unlock(myLock);
        barrier(myBarrier, NUM_PROCESSORS);
        if (diff/(n*n) < TOLERANCE)       // check convergence, all threads get same answer
            done = true;
    }
}
```

I asked the class:
Could you do better than three barriers?
Shared address space solver: one barrier

Idea:

Remove dependencies by using different \texttt{diff} variables in successive loop iterations

Trade off footprint for removing dependencies!
Three variables instead of one.
But now one barrier instead of three.
(a common parallel programming technique)

```c
int n;        // grid size
bool done = false;
LOCK myLock;
BARRIER myBarrier;
float diff[3]; // global diff, but now 3 copies

float *A = allocate(n+2, n+2);

void solve(float* A) {
    float myDiff;   // thread local variable
    int index = 0;  // thread local variable

    diff[0] = 0.0f;
    barrier(myBarrier, NUM_PROCESSORS);    // one-time only: just for init

    while (!done) {
        myDiff = 0.0f;
        // perform computation (accumulate locally into myDiff)
        //lock(myLock);
        diff[index] += myDiff;    // atomically update global diff
        unlock(myLock);
        diff[(index+1) % 3] = 0.0f;
        barrier(myBarrier, NUM_PROCESSORS);
        if (diff[index]/(n*n) < TOLERANCE)
            break;
        index = (index + 1) % 3;
    }
}
```

Example from: Culler, Singh, and Gupta
Let’s think about expressing the grid solver in a message passing model

- No shared address space abstraction (i.e., no shared variables)
- Each thread has its own address space
- Threads communicate & synchronize by sending/receiving messages

One possible message passing machine configuration: a cluster of two workstations (you could make yourself such a machine in the GHC labs)
Review: assignment in a shared address space

- Grid data resided in a **single array** in shared address space
  - This array was accessible for access by all threads
    float* A = allocate(n+2, n+2);

- Each thread manipulated the region of array it was assigned to process
  - Different assignments may yield different amounts of communication, impacting performance
Message passing model: each thread operates in its own address space

This figure: four threads

So the grid data is partitioned into allocations residing in each of the four unique address spaces (four per-thread private arrays)
Data replication is now required to correctly execute the program.

**Example:**
After red cell processing is complete, thread 1 and thread 3 send row of data to thread 2 (otherwise thread 2 does not have up-to-date red cell information needed in the subsequent phase).

**Commonly used term: “ghost cells”**
“Ghost cells” are grid cells replicated from a remote address space. It’s common to say that information in ghost cells is “owned” by other threads.

**Thread 2 logic:**
```c
float local_data[(N+2)*rows_per_thread];
float ghost_row_top[N+2]; // ghost row storage
float ghost_row_bot[N+2]; // ghost row storage
int tid = get_thread_id();
int bytes = sizeof(float) * (N+2);
recv(ghost_row_top, bytes, tid-1, TOP_MSG_ID);
recv(ghost_row_bot, bytes, tid+1, BOT_MSG_ID);
// Thread 2 now has data necessary to perform computation
```
```c
int n;
int tid = get_thread_id();
int rows_per_thread = n / get_num_threads();

float* localA = allocate(sizeof(float) * (rows_per_thread+2) * (n+2));

// assume localA is initialized with starting values
// assume MSG_ID_ROW, MSG_ID_DONE, MSG_ID_DIFF are all constants

void solve() {
    bool done = false;
    while (!done) {
        float my_diff = 0.0f;
        if (tid != 0)
            send(&localA[1,0], sizeof(float)*(N+2), tid-1, MSG_ID_ROW);  // send row 0
        if (tid != get_num_threads()-1)
            send(&localA[rows_per_thread-2,0], sizeof(float)*(N+2), tid+1, MSG_ID_ROW);
        if (tid != 0)
            recv(&localA[0,0], sizeof(float)*(N+2), tid-1, MSG_ID_ROW);
        if (tid != get_num_threads()-1)
            recv(&localA[rows_per_thread-1,0], sizeof(float)*(N+2), tid+1, MSG_ID_ROW);
        for (int i=1; i<rows_per_thread-1; i++) {
            for (int j=1; j<n+1; j++) {
                float prev = localA[i,j];
                                      localA[i,j-1] + localA[i,j+1]);
                my_diff += fabs(localA[i,j] - prev);
            }
        }
        if (pid != 0) {
            send(&mydiff, sizeof(float), 0, MSG_ID_DIFF);
            recv(&done, sizeof(bool), 0, MSG_ID_DONE);
        } else {
            float remote_diff;
            for (int i=1; i<get_num_threads()-1; i++) {
                recv(&remote_diff, sizeof(float), i, MSG_ID_DIFF);
                my_diff += remote_diff;
            }
            if (my_diff/(n*n) < TOLERANCE)
                done = true;
            if (int i=1; i<gen_num_threads()-1; i++)
                send(&done, sizeof(bool), i, MSD_ID_DONE);
        }
    }
}
```

Example pseudocode from: Culler, Singh, and Gupta

Message passing solver

Similar structure to shared address space solver, but now communication is explicit in message sends and receives

Send and receive ghost rows to “neighbor threads”

Perform computation

All threads send local my_diff to thread 0

Thread 0 computes global diff, evaluates termination predicate and sends result back to all other threads
Notes on message passing example

- **Computation**
  - Array indexing is relative to local address space (not global grid coordinates)

- **Communication:**
  - Performed by sending and receiving messages
  - Communicate entire rows at a time (not individual elements)

- **Synchronization:**
  - Performed by sending and receiving messages
  - Think of how to implement mutual exclusion, barriers, flags using messages

- For convenience: message passing libraries often include higher-level primitives (implemented using send and receive)

```c
reduce_add(0, &my_diff, sizeof(float)); // add up all my_diffs, result provided to thread 0
if (pid == 0 && my_diff/(n*n) < TOLERANCE)
    done = true;
broadcast(0, &done, sizeof(bool), MSG_DONE); // thread 0 sends done to all threads
```
Variants of send and receive messages

- **Synchronous:**
  - **SEND:** call returns when sender receives acknowledgement that message data resides in address space of receiver
  - **RECV:** call returns when data from received message is copied into address space of receiver and acknowledgement sent back to sender

Sender:
- Call SEND(foo)
- Copy data from sender's address space buffer 'foo' into network buffer
- Send message
- Receive ack
- SEND() returns

Receiver:
- Call RECV(bar)
- Receive message
- Copy data into receiver's address space buffer 'bar'
- Send ack
- RECV() returns
As implemented on the prior slide, if our message passing solver uses blocking send/recv it would deadlock!

Why?

How can we fix it?
(while still using blocking send/recv)
Message passing solver

Similar structure to shared address space solver, but now communication is explicit in message sends and receives

```c
int n;
int tid = get_thread_id();
int rows_per_thread = n / get_num_threads();
float* localA = allocate(sizeof(float) * (rows_per_thread+2) * (n+2));

// assume localA is initialized with starting values
// assume MSG_ID_ROW, MSG_ID_DONE, MSG_ID_DIFF are all constants

void solve()
{
    bool done = false;
    while (!done) {
        float my_diff = 0.0f;
        if (tid != 0) {
            send(&localA[1,0], sizeof(float)*(N+2), tid-1, MSG_ID_ROW); // send row 0
        }
        if (tid != get_num_threads()-1) {
            send(&localA[rows_per_thread-2,0], sizeof(float)*(N+2), tid+1, MSG_ID_ROW);
        }
        for (int i=1; i<rows_per_thread-1; i++) {
            for (int j=1; j<n+1; j++) {
                float prev = localA[i,j];
                my_diff += fabs(localA[i,j] - prev);
            }
        }
        if (pid != 0) {
            send(&mydiff, sizeof(float), 0, MSG_ID_DIFF);
            recv(&done, sizeof(bool), 0, MSG_ID_DONE);
        } else {
            float remote_diff;
            for (int i=1; i<get_num_threads()-1; i++) {
                recv(&remote_diff, sizeof(float), i, MSG_ID_DIFF);
                my_diff += remote_diff;
            }
        }
        if (my_diff/(n*n) < TOLERANCE) {
            done = true;
        }
        if (int i=1; i<gen_num_threads()-1; i++) {
            send(&done, sizeof(bool), i, MSD_ID_DONE);
        }
    }
}
```

Send and receive ghost rows to “neighbor threads”

Perform computation

All threads send local my_diff to thread 0

Thread 0 computes global diff, evaluates termination predicate and sends result back to all other threads

Example pseudocode from: Culler, Singh, and Gupta
Variants of send and receive messages

- **Blocking async:**
  - **SEND:** call copies data from address space into system buffers, then returns
    - Does not guarantee message has been received (or even sent)
  - **RECV:** call returns when data copied into address space, but no ack sent

**Sender:**

- Call SEND(foo)
  - Copy data from sender’s address space buffer ‘foo’ into network buffer
  - SEND(foo) returns, calling thread continues execution

**Receiver:**

- Call RECV(bar)
  - Receive message
  - Copy data into receiver’s address space buffer
  - RECV(bar) returns

**RED TEXT = executes concurrently with sender’s application thread**
Variants of send and receive messages

- **Synchronous**
  - Send
  - Recv

- **Asynchronous**
  - Blocking async
  - Non-blocking async

**Non-blocking asynchronous:** ("non-blocking")

- **SEND**: call returns immediately. Buffer provided to SEND cannot be modified by calling thread since message processing occurs concurrently with thread execution.

- **RECV**: call posts intent to receive, returns immediately.

- Use SENDPROBE, RECVPROBE to determine actual send/receipt status

**Sender:**
- Call SEND(foo)
- SEND(foo) returns handle h1
- Copy data from ‘foo’ into network buffer
- Send message
- Call SENDPROBE(h1) // if message sent, now safe for thread to modify ‘foo’

**Receiver:**
- Call RECV(bar)
- RECV(bar) returns handle h2
- Receive message
- Messaging library copies data into ‘bar’
- Call RECVPROBE(h2)
  - // if received, now safe for thread
  - // to access ‘bar’

Red text = executes concurrently with application thread
Variants of send and receive messages

- Synchronous
- Asynchronous
  - Blocking async
  - Non-blocking async

The variants of send/recv provide different levels of programming complexity / opportunity to optimize performance
Solver implementation in THREE programming models

1. Data-parallel model
   - Synchronization:
     - \texttt{forall} loop iterations are independent (can be parallelized)
     - Implicit barrier at end of outer \texttt{forall} loop body
   - Communication
     - Implicit in loads and stores (like shared address space)
     - Special built-in primitives: e.g., \texttt{reduce}

2. Shared address space model
   - Synchronization:
     - Locks used to ensure mutual exclusion
     - Barriers used to express coarse dependencies (e.g., between phases of computation)
   - Communication
     - Implicit in loads/stores to shared variables

3. Message passing model
   - Synchronization:
     - Implemented via messages
     - Mutual exclusion exists by default: no shared data structures
   - Communication:
     - Explicit communication via send/recv needed for parallel program correctness
     - Bulk communication for efficiency: e.g., communicate entire rows, not single elements
     - Several variants of send/recv, each has different semantics
Optimizing parallel program performance

( how to be l33t )
Programming for high performance

- Optimizing the performance of parallel programs is an iterative process of refining choices for decomposition, assignment, and orchestration...

- Key goals (that are at odds with each other)
  - Balance workload onto available execution resources
  - Reduce communication (to avoid stalls)
  - Reduce extra work (overhead) performed to increase parallelism, manage assignment, etc.

- We are going to talk about a rich space of techniques
TIP #1: Always implement the simplest solution first, then measure/analyze performance to determine if you need to do better.

“My solution scales” = your code scales as much as you need it to (if you anticipate only running low core count machines, it may be unnecessary to implement a complex approach that creates and hundreds or thousands of pieces of independent work)
Balancing the workload

Ideally: all processors are computing all the time during program execution (they are computing simultaneously, and they finish their portion of the work at the same time)

Recall Amdahl’s Law:
Only small amount of load imbalance can significantly bound maximum speedup

P4 does 20% more work → P4 takes 20% longer to complete
→ 20% of parallel program runtime is essentially serial execution
(work in serialized section here is about 5% of a sequential implementation execution time: $S = 0.05$ in Amdahl’s law equation)
Static assignment

- Assignment of work to threads is pre-determined
  - Not necessarily determined at compile-time (assignment algorithm may depend on runtime parameters such as input data size, number of threads, etc.)
- Recall solver example: assign equal number of grid cells (work) to each thread (worker)
  - We discussed blocked and interleaved static assignments of work to workers

- Good properties of static assignment: simple, essentially zero runtime overhead (in this example: extra work to implement assignment is a little bit of indexing math)
Static assignment

- When is static assignment applicable?
- When the cost (execution time) of work and the amount of work is predictable (so the programmer can work out assignment in advance)
- Simplest example: it is known up front that all work has the same cost

In the example above:
There are 12 tasks, and it is known each have the same cost.
Statically assign three tasks to each of the four processors.
Static assignment

When is static assignment applicable?
- Example 2: predictable, but not all jobs have same cost (see example below)
- Example 3: Statistics about execution time are known (e.g., same cost on average)

Jobs have unequal, but known cost: assign to processors to ensure overall good load balance
"Semi-static" assignment

- Cost of work is predictable for near-term future
  - Recent past good predictor of near future
- Periodically profile application and re-adjust assignment
  - Assignment is static during interval between re-adjustment

Particle simulation:
Redistribute particles as they move over course of simulation
(if motion is slow, redistribution need not occur often)

Adaptive mesh:
Mesh is changed as object moves or flow over object changes, but changes occur slowly (color indicates assignment of parts of mesh to processors)

Dynamic assignment

- Assignment is determined at runtime to ensure a well distributed load.
  (The execution time of tasks, or the total number of tasks, is unpredictable.)

---

Sequential program
(independent loop iterations)

```c
int N = 1024;
int* x = new int[N];
bool* prime = new bool[N];

// initialize elements of x
for (int i=0; i<N; i++)
{
    // unknown execution time
    is_prime[i] = test_primality(x[i]);
}
```

---

Parallel program
(SPMD execution of multiple threads, shared address space model)

```c
LOCK counter_lock;
int counter = 0;  // shared variable (assume
                  // initialization to 0)

int N = 1024;
int* x = new int[N];
bool* is_prime = new bool[N];

// initialize elements of x
while (1) {
    int i;
    lock(counter_lock);
    i = counter++;
    unlock(counter_lock);
    if (i >= N)
        break;
    is_prime[i] = test_primality(x[i]);
}
```

---

atomic_incr(counter)
Dynamic assignment using work queues

Sub-problems (a.k.a. “tasks”, “work”)

Shared work queue: a list of work to do
(for now, let’s assume each piece of work is independent)

Worker threads:
Pull data from shared work queue
Push new work to queue as it is created
What constitutes a piece of work?

What is a potential problem with this implementation?

LOCK counter_lock;
int counter = 0; // shared variable (assume
  // initialization to 0)

const int N = 1024;
float* x = new float[N];
bool* prime = new bool[N];

// initialize elements of x

while (1) {
  int i;
  lock(counter_lock);
  i = counter++;
  unlock(counter_lock);
  if (i >= N)
    break;
  is_prime[i] = test_primality(x[i]);
}

Fine granularity partitioning:
Here: 1 “task” = 1 element

Likely good workload balance (many small tasks)
Potential for high synchronization cost
(serialization at critical section)

So... IS this a problem?
Increasing task granularity

```c
LOCK counter_lock;
int counter = 0; // shared variable (assume // initialization to 0)
const int N = 1024;
const int GRANULARITY = 10;
float* x = new float[N];
bool* prime = new bool[N];
// initialize elements of x

while (1) {
    int i;
    lock(counter_lock);
    i = counter;
    counter += GRANULARITY;
    unlock(counter_lock);
    if (i >= N)
        break;
    int end = min(i + GRANULARITY, N);
    for (int j=i; j<end; j++)
        is_prime[i] = test_primality(x[i]);
}
```

Coarse granularity partitioning:
1 “task” = 10 elements

Decreased synchronization cost
(Critical section entered 10 times less)

So... have we done better?
Rule of thumb

- Useful to have many more tasks* than processors
  (many small tasks enables good workload balance via dynamic assignment)
  - Motivates small granularity tasks

- But want as few tasks as possible to minimize overhead of managing the assignment
  - Motivates large granularity tasks

- Ideal granularity depends on many factors
  (Common theme in this course: must know your workload, and your machine)

* I had to pick a term. Here I’m using “task”
generally: it’s a piece of work, a sub-problem, etc.
Smarter task scheduling

Consider dynamic scheduling via a shared work queue

What happens if the system assigns these tasks to workers in left-to-right order?
Smarter task scheduling

What happens if scheduler runs the long task last? Potential for load imbalance!

One possible solution to imbalance problem:

Divide work into a larger number of smaller tasks
  — Hopefully “long pole” gets shorter relative to overall execution time
  — May increase synchronization overhead
  — May not be possible (perhaps long task is fundamentally sequential)
Smarter task scheduling

Schedule long task first to reduce “slop” at end of computation

Another solution: smarter scheduling

Schedule long tasks first

- Thread performing long task performs fewer overall tasks, but approximately the same amount of work as the other threads.
- Requires some knowledge of workload (some predictability of cost)
Decreasing synchronization overhead

- **Distributed work queues**
  - Replicate data to remove synchronization on single work queue

**Subproblems**
(a.k.a. “tasks”, “work to do”)

**Set of work queues**
(In general, one per worker thread)

**Worker threads:**
- Pull data from OWN work queue
- Push new work to OWN work queue
- When local work queue is empty...
- STEAL work from another work queue
Distributed work queues

- **Costly synchronization/communication occurs during stealing**
  - But not every time a thread takes on new work
  - Stealing occurs only when necessary to ensure good load balance

- **Leads to increased locality**
  - Common case: threads work on tasks they create (producer-consumer locality)

- **Implementation challenges**
  - Who to steal from?
  - How much to steal?
  - How to detect program termination?
  - Ensuring local queue access is fast (while preserving mutual exclusion)
Work in task queues need not be independent

A task is not removed from queue and assigned to worker thread until all task dependencies are satisfied.

Workers can submit new tasks (with optional explicit dependencies) to task system.

```c
foo_handle = enqueue_task(foo); // enqueue task foo (independent of all prior tasks, run at any time)
bar_handle = enqueue_task(bar, foo_handle); // enqueue task bar, cannot run until foo is complete.
```
Challenge: achieving good workload balance
- Want all processors working at all the time (otherwise, resources are idle!)
- But want low cost solution for achieving this balance
  - Minimize computational overhead (e.g., scheduling/assignment logic)
  - Minimize synchronization costs

Static assignment vs. dynamic assignment
- Really, it is not an either/or decision, there’s a continuum of choices
- Use up-front knowledge about workload as much as possible to reduce load imbalance and task management/synchronization costs (in the limit, if the system knows everything, use fully static assignment)

Issues discussed today span decomposition, assignment, and orchestration
CUDA and GPU programming self check
(this is also an abstraction vs. implementation self check)
(and a work scheduling self check)
(and a parallel architecture self check)
Recall the 1D convolution example

output[i] = (input[i] + input[i+1] + input[i+2]) / 3.f;
Recalling the 1D convolution in CUDA example

Simplest possible CUDA kernel for this computation.

One CUDA thread per output element.
Each thread independently loads input elements it requires.
Notice: CUDA threads in thread block do not cooperate.
(no logic based on block size in the kernel*)

* other than computing thread ‘index’
Implementation using per-block shared memory

```c
#define THREADS_PER_BLK 128

__global__ void convolve_1d_shared(int N, float* input, float* output) {
  __shared__ float support[THREADS_PER_BLK+2]; // per block allocation
  int index = blockIdx.x * blockDim.x + threadIdx.x; // thread local variable

  support[threadIdx.x] = input[index];
  if (threadIdx.x < 2) {
    support[THREADS_PER_BLK + threadIdx.x] = input[index+THREADS_PER_BLK];
  }

  __syncthreads();

  float result = 0.0f; // thread-local variable
  for (int i=0; i<3; i++)
    result += support[threadIdx.x + i];

  output[index] = result / 3.f;
}
```

**Thread Block 0 Work**
- All threads cooperatively load block's support region from global memory into shared memory
- total of 130 load instructions instead of 3 * 128 load instructions

**Thread Block (N/128)-1 Work**
-
Running the kernel

Kernel's execution requirements:
- Each thread block must execute 128 CUDA threads
- Each thread block must allocate $130 \times \text{sizeof(float)} = 520$ bytes of shared memory

Let's assume array size $N$ is very large, so the host-side kernel launch generates thousands of thread blocks.

```c
#define THREADS_PER_BLK 128
convolve_1d_shared<<<N/THREADS_PER_BLK, THREADS_PER_BLK>>>(N, input_array, output_array);
```

Let's run this program on the fictitious two-core GPU below.
Running the CUDA kernel

Kernel’s execution requirements:
Each thread block must execute 128 CUDA threads
Each thread block must allocate 130 * sizeof(float) = 520 bytes of shared memory

Step 1: host sends CUDA device (GPU) a command (“execute this kernel”)

```
EXECUTE: convolve_1d_shared
ARGS: N, input_array, output_array
NUM_BLOCKS: 1000
```

![Diagram of GPU Work Scheduler and Core 0 and Core 1 with execution context and shared memory storage]
Running the CUDA kernel

Kernel's execution requirements:
- Each thread block must execute 128 CUDA threads
- Each thread block must allocate $130 \times \text{sizeof(float)} = 520$ bytes of shared memory

Step 2: scheduler maps block 0 to core 0 (reserves execution contexts and shared storage)

EXECUTE: convolve_1d_shared
ARGS: N, input_array, output_array
NUM_BLOCKS: 1000

NEXT = 1
TOTAL = 1000

---

Core 0

Fetch/Decode

Block 0 (contexts 0-127)
Execution context storage for 384 CUDA threads

Block 0: support (520 bytes)
“Shared” memory storage (1.5 KB)

Core 1

Fetch/Decode

Execution context storage for 384 CUDA threads

“Shared” memory storage (1.5 KB)
Running the CUDA kernel

Kernel’s execution requirements:
- Each thread block must execute 128 CUDA threads
- Each thread block must allocate $130 \times \text{sizeof(float)} = 520$ bytes of shared memory

Step 3: scheduler continues to map blocks to available execution contexts (interleaved mapping shown)

EXECUTE: convolve_1d_shared
ARGS: N, input_array, output_array
NUM_BLOCKS: 1000

NEXT = 2
TOTAL = 1000
Running the CUDA kernel

Kernel’s execution requirements:

- Each thread block must execute 128 CUDA threads
- Each thread block must allocate $130 \times \text{sizeof(float)} = 520$ bytes of shared memory

Step 3: scheduler continues to map blocks to available execution contexts (interleaved mapping shown)

EXECUTE: convolve_1d_shared
ARGS: $N, input\_array, output\_array$
NUM_BLOCKS: 1000

NEXT = 3
TOTAL = 1000

GPU Work Scheduler

Core 0

Fetch/Decode

Block 0 (contexts 0-127)

Execution context storage for 384 CUDA threads

Block 2 (contexts 128-255)

“Shared” memory storage (1.5 KB)

Core 1

Fetch/Decode

Block 1 (contexts 0-127)

Execution context storage for 384 CUDA threads

Block 1: support (520 bytes @ 0x0)

Block 2: support (520 bytes 0x0x0x0)

“Shared” memory storage (1.5 KB)
Running the CUDA kernel

Kernel’s execution requirements:
Each thread block must execute 128 CUDA threads
Each thread block must allocate \[130 \times \text{sizeof(float)} = 520\] bytes of shared memory

Step 3: scheduler continues to map blocks to available execution contexts (interleaved mapping shown).
Only two thread blocks fit on a core (third block won’t fit due to insufficient shared storage \(3 \times 520 > 1536\))

EXECUTE: `convolve_1d_shared`
ARGS: `N, input_array, output_array`
NUM_BLOCKS: 1000

NEXT = 4
TOTAL = 1000

EXECUTION CONTEXT
storage for 384 CUDA threads
“Shared” memory storage (1.5 KB)
Running the CUDA kernel

Kernel’s execution requirements:
- Each thread block must execute 128 CUDA threads
- Each thread block must allocate $130 \times \text{sizeof}(\text{float}) = 520$ bytes of shared memory

Step 4: thread block 0 completes on core 0

EXECUTE: convolve_1d_shared
ARGS: N, input_array, output_array
NUM_BLOCKS: 1000

NEXT = 4
TOTAL = 1000

Core 0

Fetch/Decode

Block 2 (contexts 0-127)
- Execution context storage for 384 CUDA threads
- “Shared” memory storage (1.5 KB)

Block 2: support (520 bytes @ 0x0)

Core 1

Fetch/Decode

Block 1 (contexts 0-127)
- Execution context storage for 384 CUDA threads
- “Shared” memory storage (1.5 KB)

Block 3 (contexts 128-255)
- Block 1: support (520 bytes @ 0x520)

Block 3: support (520 bytes @ 0x520)
Running the CUDA kernel

Kernel's execution requirements:
  - Each thread block must execute 128 CUDA threads
  - Each thread block must allocate 130 * sizeof(float) = 520 bytes of shared memory

Step 5: block 4 is scheduled on core 0 (mapped to execution contexts 0-127)

EXECUTE: convolve_1d_shared
ARGS: N, input_array, output_array
NUM_BLOCKS: 1000

NEXT = 5
TOTAL = 1000

Core 0
- Block 4 (contexts 0-127)
  - Block 4: support (520 bytes @ 0x0)
  - Execution context storage for 384 CUDA threads
- Block 2 (contexts 128-255)
  - Block 2: support (520 bytes 0x520)
  - “Shared” memory storage (1.5 KB)

Core 1
- Block 1 (contexts 0-127)
  - Block 1: support (520 bytes @ 0x0)
- Block 3 (contexts 128-255)
  - Block 3: support (520 bytes @ 0x520)
  - Execution context storage for 384 CUDA threads

“Shared” memory storage (1.5 KB)
Running the CUDA kernel

Kernel's execution requirements:
- Each thread block must execute 128 CUDA threads
- Each thread block must allocate $130 \times \text{sizeof(float)} = 520$ bytes of shared memory

Step 6: thread block 2 completes on core 0

EXECUTE: convolve_1d_shared
ARGS: N, input_array, output_array
NUM_BLOCKS: 1000

GPU Work Scheduler

NEXT = 5
TOTAL = 1000
Running the CUDA kernel

Kernel's execution requirements:
Each thread block must execute 128 CUDA threads
Each thread block must allocate 130 * sizeof(float) = 520 bytes of shared memory

Step 7: thread block 5 is scheduled on core 0 (mapped to execution contexts 128-255)

EXECUTE: convolve_1d_shared
ARGS: N, input_array, output_array
NUM_BLOCKS: 1000

GPU Work Scheduler

NEXT = 6
TOTAL = 1000
What is a “warp”? 

- Before all else: a warp is a CUDA implementation detail on NVIDIA GPUs
- On modern NVIDIA hardware, groups of 32 CUDA threads in a thread block are executed simultaneously using 32-wide SIMD execution.

In this fictitious NVIDIA GPU example:
Core maintains contexts for 12 warps
Selects one warp to run each clock
What is a “warp”? 

- Before all else: a warp is a CUDA implementation detail on NVIDIA GPUs

- On modern NVIDIA hardware, groups of 32 CUDA threads in a thread block are executed simultaneously using 32-wide SIMD execution.
  - These 32 logical CUDA threads share an instruction stream and therefore performance can suffer due to divergent execution.
  - This mapping is similar to how ISPC runs program instances in a gang.

- The group of 32 threads sharing an instruction stream is called a warp.
  - In a thread block, threads 0-31 fall into the same warp (so do threads 32-63, etc.)
  - Therefore, a thread block with 256 CUDA threads is mapped to 8 warps.
  - Each “SMX” core in the GTX 680 we discussed last time is capable of scheduling and interleaving execution of up to 64 warps.
  - So a “SMX” core is capable of concurrently executing multiple CUDA thread blocks.
This program creates a histogram. It is reasonable CUDA code?

- This example: build a histogram of values in an array
  - CUDA threads atomically update shared variables in global memory

- Notice I have never claimed CUDA thread blocks were guaranteed to be independent. I only stated CUDA reserves the right to schedule them in any order.

- This use of atomics does not impact implementation’s ability to schedule blocks in any order (atomics used for mutual exclusion, and nothing more)

```c
int A[N]
int* A = {0, 3, 4, 1, 9, 2, ... , 8, 4, 1};  // array of integers between 0-9

atomicAdd(&counts[A[i]], 1);

... atomicAdd(&counts[A[i]], 1);
```
But is this reasonable CUDA code?

- Consider implementation of on a single core GPU with resources for one CUDA thread block per core
  - What happens if the CUDA implementation runs block 0 first?
  - What happens if the CUDA implementation runs block 1 first?

```c
int myFlag

// do stuff here
atomicAdd(&myFlag, 1);

while(atomicAdd(&myFlag, 0) == 0)
{
}
// do stuff here

Global memory

int myFlag

(assume myFlag is initialized to 0)
```
**“Persistent thread” CUDA programming style**

```
#define THREADS_PER_BLK 128
#define BLOCKS_PER_CHIP 15 * 12  // specific to a certain GTX 480 GPU

__device__ int workCounter = 0;  // global mem variable

__global__ void convolve(int N, float* input, float* output) {
    __shared__ int startingIndex;
    __shared__ float support[THREADS_PER_BLK+2];  // shared across block

    while (1) {
        if (threadIdx.x == 0)
            startingIndex = atomicInc(workCounter, THREADS_PER_BLK);
        __syncthreads();
        if (startingIndex >= N)
            break;

        int index = startingIndex + threadIdx.x;  // thread local
        support[threadIdx.x] = input[index];
        if (threadIdx.x < 2)
            support[THREADS_PER_BLK+threadIdx.x] = input[index+THREADS_PER_BLK];
        __syncthreads();

        float result = 0.0f;  // thread-local variable
        for (int i=0; i<3; i++)
            result += support[threadIdx.x + i];
        output[index] = result;
        __syncthreads();
    }
}

// host code ///////////////////////////////////////////////////////////////!
int N = 1024 * 1024;
cudaMalloc(&devInput, N+2);  // allocate array in device memory
cudaMalloc(&devOutput, N);  // allocate array in device memory
// properly initialize contents of devInput here ...
convolve<<BLOCKS_PER_CHIP, THREADS_PER_BLK>>>(N, devInput, devOutput);
```

Idea: write CUDA code that requires knowledge of the number of cores and blocks per core that are supported by underlying GPU implementation.

Programmer launches exactly as many thread blocks as will fill the GPU

(Program makes assumptions about GPU implementation: that GPU will in fact run all blocks concurrently. Ug!)

Now, work assignment to blocks is implemented entirely by the application (circumvents GPU thread block scheduler)

Now programmer’s mental model is that *all* threads are concurrently running on the machine at once.