LAU
الجامعة اللبنانية الأميركية
Lebanese American University

School of Arts & Sciences
Department of Computer Science & Mathematics
CSC 447—Parallel Programming for Multicore and Cluster Systems
Spring 2018

Midterm Examination

Name: _____ **Student ID:** _____

Signature: _____ **Duration:** **75 minutes**

General Instructions

- There are 9 questions and 15 pages. Make sure that you have all of them.
- Exam questions are **NOT** sorted by order of difficulty. Scan the exam before you start and budget your time over the exam questions so you can maximize your grade.
- Your answers should be *brief* and *right to the point*. There is no need for essay answers! Use the back of the previous sheet if you need additional space.
- Your handwriting should be readable so it can be graded. You are liable to have points deducted from your grade if your handwriting is excessively difficult to decipher!
- The exam is a **closed** book, **closed** notes, and **closed** neighbor exam. **Any attempts at cheating or communicating with a neighbor will lead to expulsion from the exam!**

Question	Points	Score
1	16	
2	25	
3	10	
4	10	
5	5	
6	5	
7	5	
8	12	
9	12	
Total:	100	

Multiple Choice Questions

(16) 1. Circle the appropriate answer. There maybe more than one correct answer per question!

(a) Super-linear speedup with p processors occurs when:
- A. Speedup is equal to p
- B. Speedup is equal to $2 * p$
- C. Speedup is less than p
- D. Speedup is greater than p

(b) Which of the following does not affect the cost of sending a message from a process to another process?
- A. Memory size
- B. Message size
- C. Interconnect bandwidth
- D. interconnect latency

(c) Which of the following is not a reason for the non-determinism of parallel programs?
- A. race condition
- B. coherence
- C. non-uniform memory access
- D. parallel execution of processes

(d) A code in an OpenMP program that is not covered by a `#pragma` is executed by:
- A. All threads
- B. A single thread
- C. As many threads as it is set in the `omp_set_num_threads()`
- D. Hard to know as this is compiler dependent.

(e) Which of the following provides more potential performance gain:
- A. Multiple threads within a process
- B. Multiple processes within a thread
- C. Multiple threads and no processes
- D. Multiple processes and no threads

(f) MPI is very convenient when (choose the most accurate)
- A. There are a lot of processes
- B. There are a lot of processes exchanging a lot of data
- C. There are a lot of dependent processes exchanging very few data
- D. There are a lot of independent processes exchanging very few data

(g) In the message passing approach:
- A. Serial code is made parallel by adding directives that tell the compiler how to distribute data and work across the processors.
- B. Details of how data distribution, computation, and communications are to be done are left to the compiler.
- C. It is not very flexible.
- D. It is left up to the programmer to explicitly divide data and work across the processors as well as manage the communications among them.
- E. None of the above.

(h) Which of the following is true for all send routines?
- A. It is always safe to overwrite the sent variable(s) on the sending processor after the send returns.
- B. Completion implies that the message has been received at its destination.
- C. It is always safe to overwrite the sent variable(s) on the sending processor after the send is complete.
- D. All of the above.
- E. None of the above.

Performance Analysis

2. A parallel program has the following run-times when run with problem size 10^6:

Number of processes	1	2	4	8	16
Run-time (secs)	200	100	75	50	40

(5) (a) What is the speedup when the program is run with 16 processes?

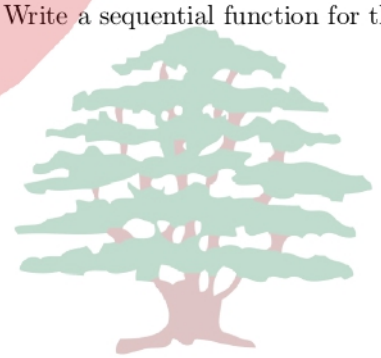
(5) (b) What is the efficiency when the program is run with 8 processes?

(15) (c) Is the scalable? Why?

Parallel Programming Using MPI: Parallel Rank Sort

(10) 3. Rank sort is a sorting technique that counts the number of *numbers* that are smaller than each selected *number*. The count provides the position of selected *number* in the sorted list; that is, its "rank." Thus, $a[0]$ is read and compared with each of the other numbers, $a[1] \ldots a[n-1]$, recording the number of numbers less than $a[0]$. Suppose this number is x. This is the index of the location in the final sorted list. The number $a[0]$ is copied into the final sorted list $b[0] \ldots b[n-1]$, at location $b[x]$. Actions repeated with the other numbers. The algorithm has an overall sorting time complexity of $O(n^2)$.

(a) Write a sequential function for the Rank Sort using C.



(b) Write a parallel program using MPI for ranksort. Give clear comments explaining the code. Analyze the code and discuss the speed up attained. Is it worth it?

Multi-Core Programming using OpenMP

4. Circle the correct answer below, given the following code segment. Assume there are four threads with ids 0, 1, 2 and 3:

```
#pragma par for private (t, j)
for (i=0; i < 1000000; i++) {
     t = my_thread_id( )
          for (j=0; j < 1000000; j++) {
               data[j] = t;
     }
}
```

(5) (a) There is no race in the above code. Justify your answer below.

A. True

B. False

(5) (b) Elements of data will all have a value of 0, 1, 2 or 3. Justify your answer below.

A. True

B. False

(5) 5. Label the scope of the variables in the following block of code and identify any problems that you may find:

```
#pragma omp parallel for private(a,b)
for (i = 0; i < N; i++) {
  int x = 0;
  c--;
  for (j = i; j < N; j++)
    x += func(c, b[j]);
  a[i] = x;
}
```

(5) 6. Identify the loop-carried dependencies in the following code block. Write a parallel version of the code in OpenMP with the dependencies removed.

```
  for (i = 0; i < N - 2; i++) {
    a[i] += a[i + 2] + 5;
    x += a[i];
}
```

(5) 7. Parallelize the following code using OpenMP:

```
  min = a[0];
  max = a[0];
  for (i = 1; i < N; i++) {
    if (a[i] < min)
      min = a[i];
    if (a[i] > max)
      max = a[i];
}
```

CUDA Programming

8. For the vector addition kernel and the corresponding kernel launch code, answer each of the sub-questions below.

```
__global__ void vecAddKernel(float* A_d, float* B_d, float* C_d, int n)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  if (i<n)
     C_d[i] = A_d[i] + B_d[i];
}
int vectAdd(float* A, float* B, float* C, int n)
{
    //assume that size has been set to the actual length of
    //arrays A, B, and C
   int size = n * sizeof(float);

    cudaMalloc ((void **) &A_d, size );
    cudaMalloc((void **) &B_d, size);
    cudaMalloc((void **) &C_d, size);
    cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);
    vecAddKernel<<<ceil(n/256), 256>>>(A_d, B_d, C_d, n);
    cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);
}
```

(4) (a) Assume that the size of A, B, and C is 1000 elements. How many thread blocks will be generated?

(4) (b) Assume that the size of A, B, and C is 1000 elements. How many warps are there in each block?

(4) (c) Assume that the size of A, B, and C is 1000 elements. How many threads will be created in the grid?

Pthread Programming

9. Assume we are given the following code:

```
   long count = 0, result = 0;
   pthread_mutex_t mutex;
   pthread_cond_t cond;

   void *P1(void *t) {
      sleep(1);
      pthread_mutex_lock(&mutex);
      pthread_cond_wait(&cond, &mutex);
      count++;
      pthread_mutex_unlock(&mutex);
      pthread_exit(NULL);
}

void *P2(void *t) {
   long j, i, res = 0;
   for (i=0 ; i < 10 ; i++) {
      for (j=0 ; j < 100000 ; j++) res += 1;
         pthread_mutex_lock(&mutex);
         result += res;
         count++;
         if (count == 12)
            pthread_cond_signal(&cond);
         pthread_mutex_unlock(&mutex);
      }
      pthread_exit(NULL);
  }

int main(int argc, char *argv[]) {
   long t1 = 1, t2 = 2, t3 = 3, i;
   pthread_t threads[3];
   pthread_mutex_init(&mutex, NULL);
   pthread_cond_init (&cond, NULL);
   pthread_create(&threads[0], NULL, P1, (void*)t1);
   pthread_create(&threads[1], NULL, P2, (void*)t2);
   pthread_create(&threads[2], NULL, P2, (void*)t3);
   for (i = 0 ; i < 3 ; i++)
      pthread_join(threads[i], NULL);
   printf("done: count=%ld result=%ld\n",count, result);
}
```

(4) (a) The code does not work as it is supposed to. What is wrong with it?

(4) (b) Fix the above code so that it produces the "correct' answer.

(4) (c) What does it print when it is correct?

OpenMP Reference Sheet for C/C++

Constructs

<parallelize a for loop by breaking apart iterations into chunks>
#pragma omp parallel for [shared(vars), private(vars), firstprivate(vars), lastprivate(vars), default(shared|none), reduction(op:vars), copyin(vars), if(expr), ordered, schedule(type[,chunkSize])]
<A,B,C such that total iterations known at start of loop>
for(A=C;A<B;A++) {
    <your code here>

    <force ordered execution of part of the code. A=C will be guaranteed to execute before A=C+1>
    #pragma omp ordered {
        <your code here>
    }
}

<parallelized sections of code with each section operating in one thread>
#pragma omp parallel sections [shared(vars), private(vars), firstprivate(vars), lastprivate(vars), default(shared|none), reduction(op:vars), copyin(vars), if(expr)] {
    #pragma omp section {
        <your code here>
    }
    #pragma omp section {
        <your code here>
    }
    ....
}

<grand parallelization region with optional work-sharing constructs defining more specific splitting of work and variables amongst threads. You may use work-sharing constructs without a grand parallelization region, but it will have no effect (sometimes useful if you are making OpenMP'able functions but want to leave the creation of threads to the user of those functions)>
#pragma omp parallel [shared(vars), private(vars), firstprivate(vars), lastprivate(vars), default(private|shared|none), reduction(op:vars), copyin(vars), if(expr)] {
    <the work-sharing constructs below can appear in any order, are optional, and can be used multiple times. Note that no new threads will be created by the constructs. They reuse the ones created by the above parallel construct.>

    <your code here (will be executed by all threads)>

    <parallelize a for loop by breaking apart iterations into chunks>
    #pragma omp for [private(vars), firstprivate(vars), lastprivate(vars), reduction(op:vars), ordered, schedule(type[,chunkSize]), nowait]
    <A,B,C such that total iterations known at start of loop>
    for(A=C;A<B;A++) {
        <your code here>

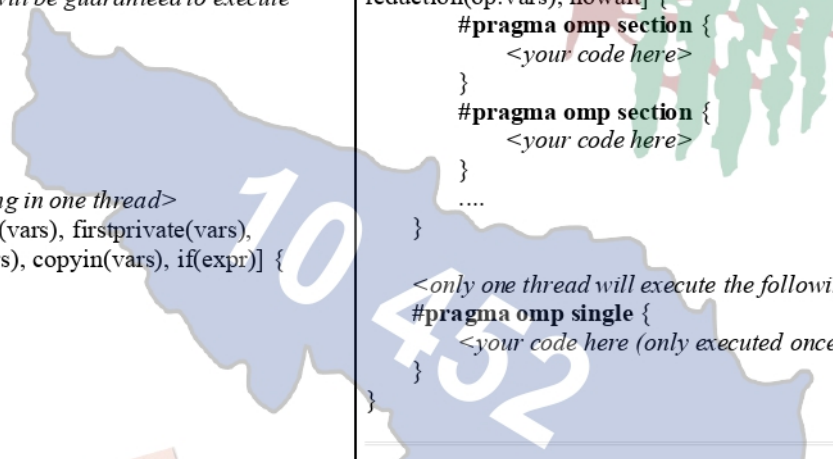
        <force ordered execution of part of the code. A=C will be guaranteed to execute before A=C+1>
        #pragma omp ordered {
            <your code here>
        }
    }

    <parallelized sections of code with each section operating in one thread>
    #pragma omp sections [private(vars), firstprivate(vars), lastprivate(vars), reduction(op:vars), nowait] {
        #pragma omp section {
            <your code here>
        }
        #pragma omp section {
            <your code here>
        }
        ....
    }

    <only one thread will execute the following. NOT always by the master thread>
    #pragma omp single {
        <your code here (only executed once)>
    }
}

Directives

shared(vars) *<share the same variables between all the threads>*
private(vars) *<each thread gets a private copy of variables. Note that other than the master thread, which uses the original, these variables are not initialized to anything.>*
firstprivate(vars) *<like private, but the variables do get copies of their master thread values>*
lastprivate(vars) *<copy back the last iteration (in a for loop) or the last section (in a sections) variables to the master thread copy (so it will persist even after the parallelization ends)>*
default(private|shared|none) *<set the default behavior of variables in the parallelization construct. shared is the default setting, so only the private and none setting have effects. none forces the user to specify the behavior of variables. Note that even with shared, the iterator variable in for loops still is private by necessity >*
reduction(op:vars) *<vars are treated as private and the specified operation(op, which can be +,*,-,&,|,&,&&,||) is performed using the private copies in each thread. The master thread copy (which will persist) is updated with the final value.>*



copyin(vars) *<used to perform the copying of threadprivate vars to the other threads. Similar to firstprivate for private vars.>*
if(expr) *<parallelization will only occur if expr evaluates to true.>*
schedule(type [,chunkSize]) *<thread scheduling model>*

type	*chunkSize*
static	*number of iterations per thread pre-assigned at beginning of loop (typical default is number of processors)*
dynamic	*number of iterations to allocate to a thread when available (typical default is 1)*
guided	*highly dependent on specific implementation of OpenMP*

nowait *<remove the implicit barrier which forces all threads to finish before continuation in the construct>*

Synchronization/Locking Constructs *<May be used almost anywhere, but will only have effects within parallelization constructs.>*

<only the master thread will execute the following. Sometimes useful for special handling of variables which will persist after the parallelization.>
#pragma omp master {
<your code here (only executed once and by the master thread).
}

<mutex lock the region. name allows the creation of unique mutex locks.>
#pragma omp critical [(name)] {
<your code here (only one thread allowed in at a time)>
}

<force all threads to complete their operations before continuing>
#pragma omp barrier

<like critical, but only works for simple operations and structures contained in one line of code>
#pragma omp atomic
<simple code operation, ex. a += 3; Typical supported operations are ++,--,+,,-,/,&,^,<<,>>,| on primitive data types>*

<force a register flush of the variables so all threads see the same memory>
#pragma omp flush[(vars)]

<applies the private clause to the vars of any future parallelize constructs encountered (a convenience routine)>
#pragma omp threadprivate(vars)

Function Based Locking *< nest versions allow recursive locking>*

void **omp_init_[nest_]lock**(omp_lock_t*) *<make a generic mutex lock>*
void **omp_destroy_[nest_]lock**(omp_lock_t*) *<destroy a generic mutex lock>*
void **omp_set_[nest_]lock**(omp_lock_t*) *<block until mutex lock obtained>*
void **omp_unset_[nest_]lock**(omp_lock_t*) *<unlock the mutex lock>*
int **omp_test_[nest_]lock**(omp_lock_t*) *<is lock currently locked by somebody>*

Settings and Control

int **omp_get_num_threads**() *<returns the number of threads used for the parallel region in which the function was called>*
int **omp_get_thread_num**() *<get the unique thread number used to handle this iteration/section of a parallel construct. You may break up algorithms into parts based on this number.>*
int **omp_in_parallel**() *<are you in a parallel construct>*
int **omp_get_max_threads**() *<get number of threads OpenMP can make>*
int **omp_get_num_procs**() *<get number of processors on this system>*
int **omp_get_dynamic**() *<is dynamic scheduling allowed>*
int **omp_get_nested**() *<is nested parallelism allowed>*
double **omp_get_wtime**() *<returns time (in seconds) of the system clock>*
double **omp_get_wtick**() *<number of seconds between ticks on the system clock>*
void **omp_set_num_threads**(int) *<set number of threads OpenMP can make>*
void **omp_set_dynamic**(int) *<allow dynamic scheduling (note this does not make dynamic scheduling the default)>*
void **omp_set_nested**(int) *<allow nested parallelism; Parallel constructs within other parallel constructs can make new threads (note this tends to be unimplemented in many OpenMP implementations)>*

<env vars- implementation dependent, but here are some common ones>
OMP_NUM_THREADS "number" *<maximum number of threads to use>*
OMP_SCHEDULE "type,chunkSize" *<default #pragma omp schedule settings>*

Legend

vars is a comma separated list of variables
[optional parameters and directives]
<descriptions, comments, suggestions>
.... above directive can be used multiple times
For mistakes, suggestions, and comments please email e_berta@plutospin.com

Pthreads API Cheat Sheet

Pthread creation

```
pthread_t threads[N]
pthread_create(&threads[i], NULL, start_routine, void *args)
pthread_join(threads[i])
```

Mutex

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex);
pthread_mutex_lock(&mutex);
pthread_mutex_unlock(&mutex);
pthread_mutex_destroy(&mutex);
```

Semaphore

```
sem_t sem;
sem_init(&sem, 0, initial) -> initial = 0: lock, initial > 0: unlocked
sem_wait(&sem) -> sem = 0: wait, sem > 0 decrement and go
sem_post(&sem) -> increment value
sem_destroy(&sem)
```

Condition Variable

```
pthread_cond_t cond
pthread_cond_init(&cond)
pthread_cond_wait(&cond, &mutex) -> unlock mutex and wait on cond
pthread_cond_signal(&cond) -> wake up threads waiting on cond
pthread_cond_destroy(&cond)
```

Common Condition Variable Usage

```
pthread_mutex_lock(&mutex);
while(isnotready()) pthread_cond_wait(&cond, &mutex);
critical section
pthread_mutex_unlock(&mutex);
pthread_cond_signal(&cond2);
```

MPI Cheat Sheet

The majority of this information is taken with permission from Henry Neeman's excellent MPI slides. These are available online from the OU Supercomputing Center for Education and Research (http://www.oscer.ou.edu/sc08workshopou.php).

Setup and Tear Down

C/C++	Fortran
`MPI_Init(&argc, &argv);`	`CALL MPI_Init(mpi_error_code)`
`MPI_Finalize();`	`CALL MPI_Finalize(mpi_error_code)`

- `MPI_Init` starts up the MPI runtime environment at the beginning of a run.
- `MPI_Finalize` shuts down the MPI runtime environment at the end of a run.

Gathering Information

C/C++	Fortran
`MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);`	`CALL MPI_Comm_Rank(my_rank, mpi_error_code)`
`MPI_Comm_size (MPI_COMM_WORLD, &num_procs);`	`CALL MPI_Comm_size(num_procs, mpi_error_code)`
`MPI_Get_processor_name (&name, &result_length)`	`CALL MPI_Get_processor_name(name, & length, mpi_error_code)`

- `MPI_Comm_size` gets the number of processes in a run, Np (typically called just after MPI_Init).
- `MPI_Comm_rank` gets the process ID that the current process uses, which is between 0 and Np-1 inclusive (typically called just after MPI_Init).
- `MPI_Get_processor_name` is not often used in real code. We used it to prove to ourselves that "Hello, World!" was running on different machines. It returns the name of the machine that the code is running on.

Communication (Message Passing)

C/C++	Fortran
`MPI_Send (message, strlen(message)+1, MPI_CHAR, destination, tag, MPI_COMM_WORLD);`	`CALL MPI_Send(message, string_len(message), & MPI_CHARACTER, destination, tag & MPI_COMM_WORLD, mpi_error_code)`

```
MPI_Recv (message,
max_message_length+1,
     MPI_CHAR, source, tag,
     MPI_COMM_WORLD, &status)
```

```
 CALL MPI_Recv(message,
maximum_message_length,
&     MPI_CHARACTER, source, tag,
&     MPI_COMM_WORLD, status, mpi_error_code);
```

```
MPI_Bcast(array, length, MPI_INTEGER,
     source, MPI_COMM_WORLD);
```

```
 CALL MPI_Bcast(array, length, MPI_INTEGER,
&     source, MPI_COMM_WORLD, mpi_error_code)
```

```
MPI_Reduce(&value, &value_sum, count,
     MPI_INT, MPI_SUM, server,
     MPI_COMM_WORLD);
```

```
 CALL MPI_Reduce(value, value_sum, count,
&     MPI_INT, MPI_SUM, server,
&     MPI_COMM_WORLD, mpi_error_code)
```

```
MPI_Allreduce(&value, &value_sum,
count,
     MPI_INT, MPI_SUM, MPI_COMM_WORLD);
```

```
 CALL MPI_Allreduce(value, value_sum, count,
&     MPI_INT, MPI_SUM, MPI_COMM_WORLD,
&     mpi_error_code)
```

- `MPI_Send` sends a message from the current process to some other process (the ***destination***).
- `MPI_Recv` receives a message on the current process from some other process (the ***source***).
- `MPI_Bcast` broadcasts a message from one process to all of the others.
- `MPI_Reduce` performs a ***reduction*** (e.g., sum, maximum) of a variable on all processes, sending the result to a *single* process.
- `MPI_Allreduce` performs a reduction of a variable on all processes, and sends result to *all* processes (and therefore takes longer)

MPI Data Types

C/C++		Fortran	
`char`	**MPI_CHAR**	`CHARACTER`	**MPI_CHARACTER**
`int`	**MPI_INT**	`INTEGER`	**MPI_INTEGER**
`float`	**MPI_FLOAT**	`REAL`	**MPI_REAL**
`double`	**MPI_DOUBLE**	`DOUBLE PRECISION`	**MPI_DOUBLE_PRECISION**

Recommended Resources

- Peter Pacheco, Parallel Programming With MPI, 1st edition, Morgan Kaufmann, 1996.
- Documentation from the makers of MPI is available online at http://www-unix.mcs.anl.gov/mpi/www/

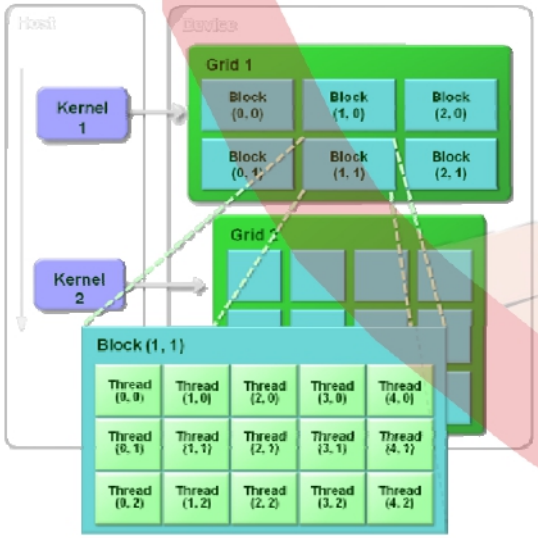
CUDA C Quick Reference

Kernels

kernel <<< dim3 Dg, dim3 Db, size_t Ns, cudaStream_t S >>> (arguments);

Dg.x*Dg.y = number of blocks, Dg.z = 1.
Db.x*Db.y*Db.z = number threads per block.
Ns = dynamically allocated shared memory, optional, default=0.
S = associated stream, optional, default=0.

Thread Hierarchy



Memory Hierarchy

Memory	Location	Cached	Access	Scope	Lifetime
Register	On-chip	N/A	R/W	Thread	Thread
Local	Off-chip	No	R/W	Thread	Thread
Shared	On-chip	N/A	R/W	Block	Block
Global	Off-chip	No	R/W	Global	Application
Constant	Off-chip	Yes	R	Global	Application
Texture	Off-chip	Yes	R	Global	Application

Device Memory

Linear Memory

cudaMalloc(void ** devptr, size_t size)

cudaFree(void * dptr)

cudaMemcpy(void *dst, const void *src, size_t size, enum cudaMemcpyKind kind)
kind = cudaMemcpyHostToHost or cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost or cudaMemcpyDeviceToDevice

CUDA Arrays

See Programming Guide for description of CUDA arrays and texture references.

Page-locked Host Memory

cudaMallocHost(void ** ptr, size_t size)

cudaFreeHost(void * ptr)

Shared Memory

Static allocation
__shared__ int a[128]

Dynamic allocation at kernel launch
extern __shared_ float b[]

Error Handling

cudaError_t cudaGetLastError(void)

const char * cudaGetErrorString(cudaError_t error)

CUDA Compilation

nvcc *flags* file.cu

A few common flags

-o	output file name
-g	host debugging information
-G	device debugging
-deviceemu	emulate on host
-use_fast_math	use fast math library
-arch	compile for specific GPU architecture
-X	pass option to host compiler

#pragma unroll *n* unroll loop *n* times.

Language Extensions

Function Qualifiers

__global__ call host, execute device.
__device__ call device, execute device.
__host__ call host, execute host (default).
__noinline__ if possible, do not inline

__host__ and __device__ may be combined to generate code for both host and device.

Variable Qualifiers

__device__ variable on device
__constant__ variable in constant memory
__shared__ variable in shared memory

Vector Types

[u]char1, [u]char2, [u]char3, [u]char4
[u]short1, [u]short2, [u]short3, [u]short4
[u]int1, [u]int2, [u]int3, [u]int4
[u]long1, [u]long2, [u]long3, [u]long4
longlong1, longlong2
float1, float2, float3, float4
double1, double2

Execution configuration

kernel <<< dim3 Dg, dim3 Db, size_t Ns, cudaStream_t S >>> (arguments)

Grids are 1D or 2D so Dg.z = 1 always
Ns optional, default 0
S optional, default 0

Built-in Variables

dim3 gridDim size of grid (1D, 2D).
dim3 blockDim size of block (1D, 2D, 3D).
dim3 blockIdx location in grid.
dim3 threadIdx location in block.
int warpSize threads in warp.

Memory Fence Functions

__threadfence(), __threadfence_block()

Synchronisation Function

__syncthreads()

Fast Mathematical Functions

__fdividef(x,y), __sinf(x), __cosf(x), __tanf(x), __sincosf(x,sinptr,cosptr), __logf(x), __log2f(x), __log10f(x), __expf(x), __exp10f(x), __powf(x,y)

Texture Functions

tex1Dfetch(), tex1D(), tex2D(), tex3D()

Timing

clock_t clock(void)

Atomic Operations

atomicAdd(), atomicSub(), atomicExch(), atomicMin(), atomicMax(), atomicInc,(), atomicDec(), atomicCAS(), atomicAnd(), atomicOr(), atomicXor().

Warp Voting Functions

int __all(int predicate)
int __any(int predicate)