Introduction to Multicore Programming

Minsoo Ryu

Department of Computer Science and Engineering
Hanyang University

Outline

1. Multithreaded Programming
2. Synchronization
3. Automatic Parallelization and OpenMP
4. GPGPU
5. Q & A

Multithreaded Programming

Processes

- **Process: a program in execution**
  - **Unit of execution (unit of scheduling)**

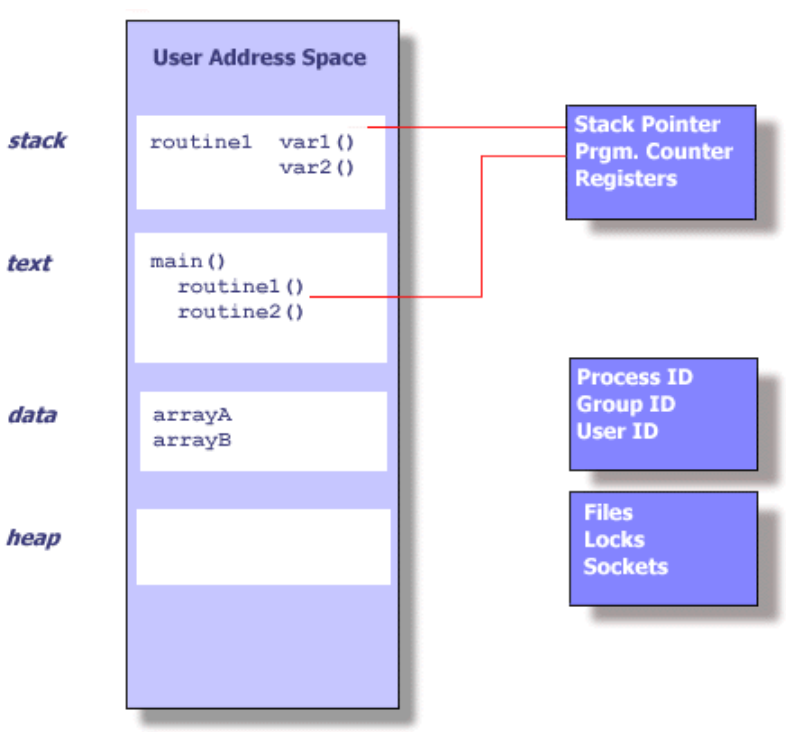
- **A process includes**
  - **PID (process id)**
    - Non-negative integer value
  - **Address space (not accessible from other processes)**
    - Code, data, stack, heap, …
  - **States**
    - Ready, running, waiting, …
  - **Current activity**
    - Current values of processor registers

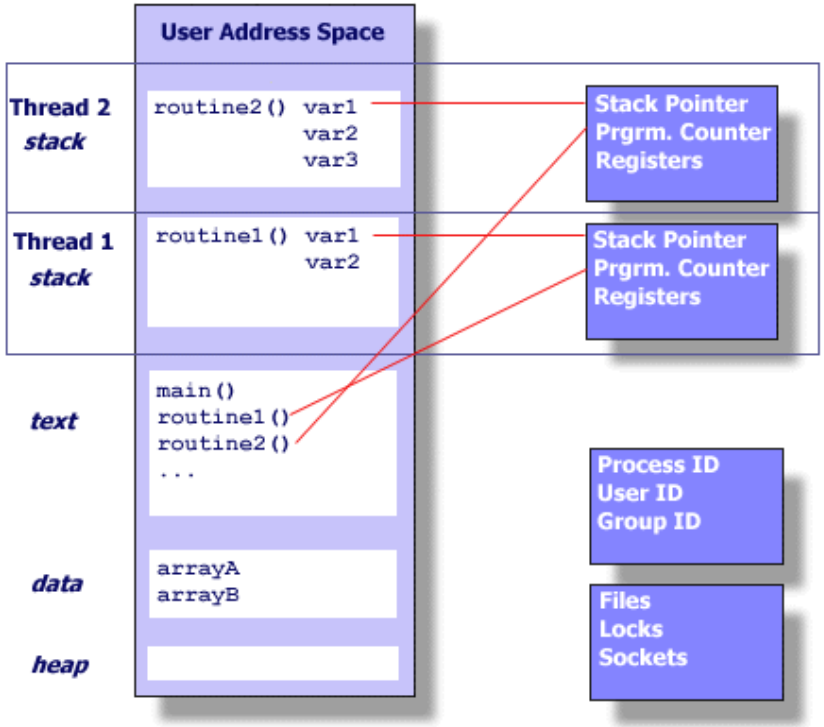
Threads

- **Thread is a unit of concurrent execution**
  - Also can be viewed as a unit of CPU scheduling
  - Sometimes called lightweight process
  - Thread has a thread ID, a program counter, a register set, and a stack

- **POSIX threads**
  - POSIX (Portable Operating System Interface) is a family of standards specified by the IEEE
  - POSIX threads, Pthreads, is a POSIX standard for threads

Single- vs. Multi-Threaded Application





Creating a New thread

```
#include <pthread.h>
#include <stdio.h>

void* thread_code( void * param )
{
  printf( "In thread code\n" );
}

int main()
{
  pthread_t thread;
  pthread_create( &thread, 0, &thread_code, 0 );
  printf( "In main thread\n" );
}
```

Passing and Reading Data

- Passing a value into a created thread

```
for ( int i=0; i<10; i++ )
pthread_create( &thread, 0, &thread_code, (void *)i );
```

- Reading the parameter passed to the new thread

```
void* child_thread( void* value )
{
  int id = (int)value;
  ...
}
```

Waiting for a Thread to Terminate

➢ Waiting with the pthread_join( ) function

```
void* child_thread( void * param )
{
  int id = (int)param;
  printf( "Start thread %i\n", id );
  return (void *)id;
}

int main()
{
  pthread_t thread[10];
  int return_value[10];
  for ( int i=0; i<10; i++ )
  {
    pthread_create( &thread[i], 0, &child_thread, (void*)i );
  }
  for ( int i=0; i<10; i++ )
  {
    pthread_join( thread[i], (void**)&return_value[i] );
    printf( "End thread %i\n", return_value[i] );
  }
}
```

Thread Private Data

- Variables held on the stack are thread-private data
  - Parameters passed into functions are also thread private data

```
double func( double a )
{
  double b;
...
```

  - Both variables a and b are private to a thread

Thread Private Data

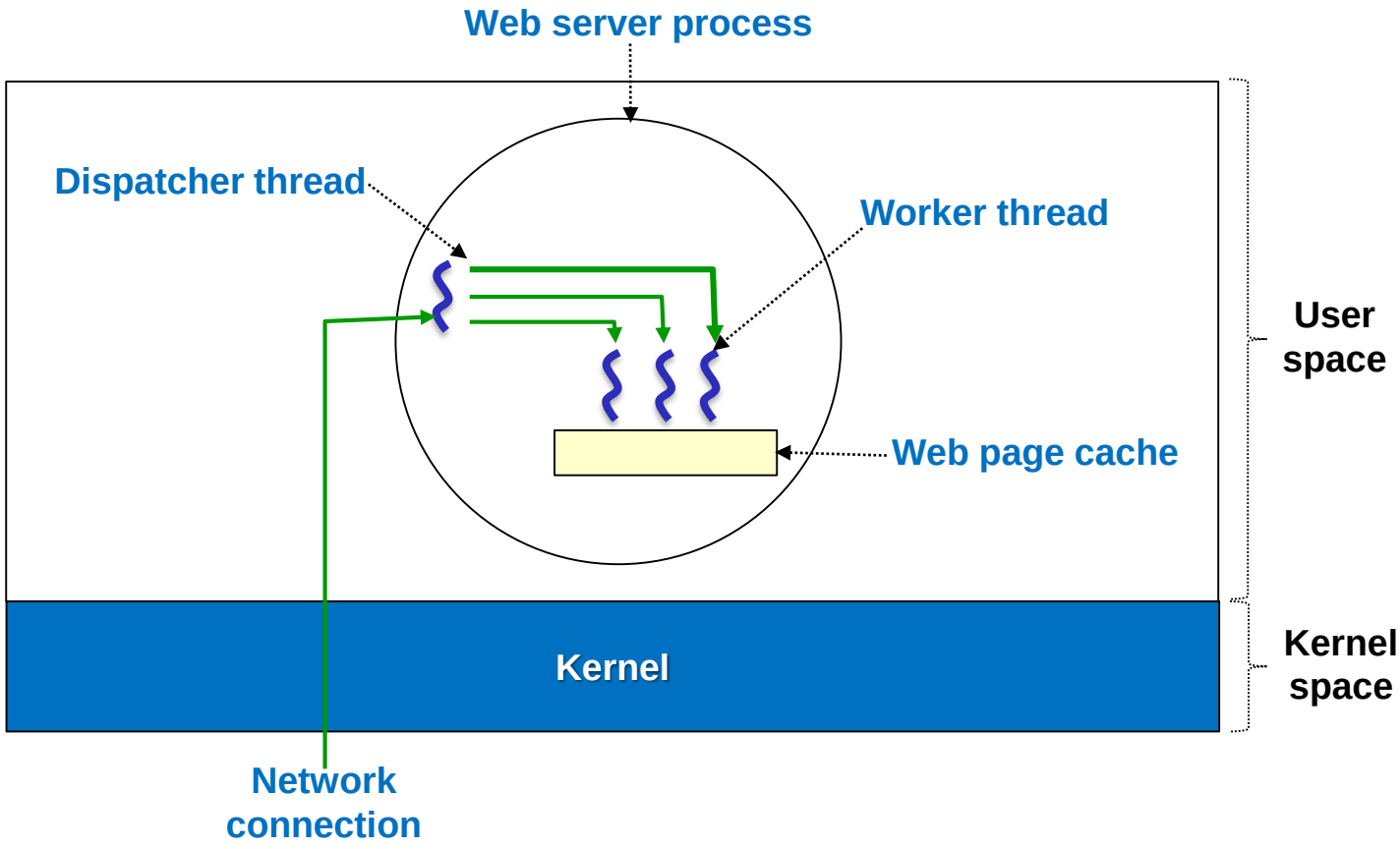
- A thread-local variable is global data that is visible to all the routines, but each thread sees only its own private copy of the data

```
__thread void * mydata;

void * threadFunction( void * param )
{
  mydata = param;
...
```

  - The variable mydata is local to the thread, so each thread can see a different value for the variable

Example: Multithreading for Web Server



Example: Multithreading for Web Server

```
while (TURE){
   get_next_request(&buf);
   dispatch_work(&buf);
}
```

Dispatcher Thread

```
while (TURE){
   wait_for_work(&buf)
   look_for_page_in_cache(&buf, &page);
   if(page_not_in_cache(&page))
      read_page_from_disk(&buf, &page);
   return_page(&page);
}
```

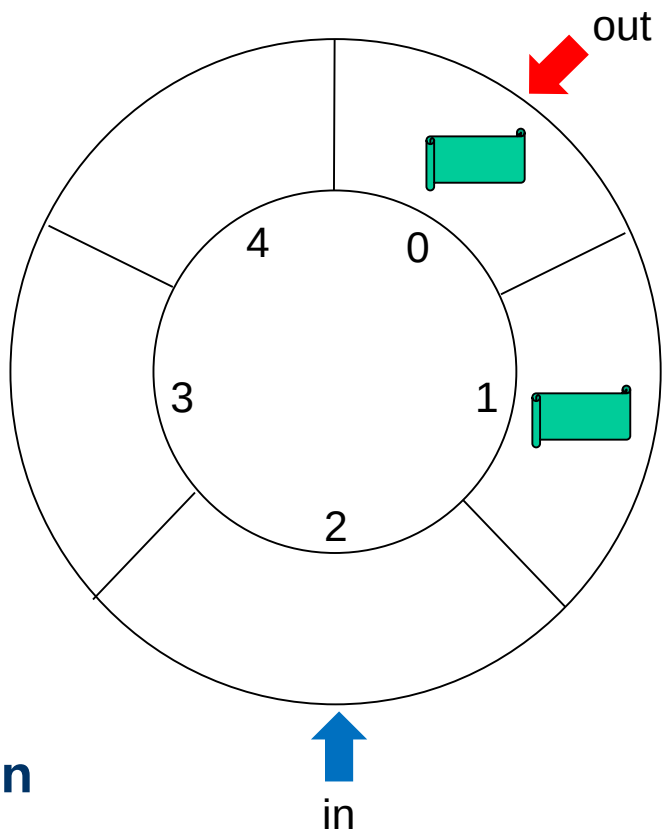
Worker Thread

Synchronization

Shared Buffer

- **Shared buffer**

```
#define BUFFER_SIZE 5
typedef struct {
   . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```



- **in: points to the next free position**
- **out: points to the first full position**

Two Tasks with Shared Buffer

Producer

```
item nextProduced;

while (1) {
    while ( /* buffer is full */ )
        ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
}
```

Consumer

```
item nextConsumed;

while (1) {
    while ( /* buffer is empty */ )
        ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
}
```

Using a Counter Variable

Producer

```
item nextProduced;

while (1) {
    while (counter == BUFFER_SIZE)
        ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

Consumer

```
item nextConsumed;

while (1) {
    while (counter == 0)
        ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
}
```

Atomicity

- **The statement "count++" may be implemented in machine language as:**

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **The statement "count--" may be implemented as:**

```
register2 = counter
register2 = register2 – 1
counter = register2
```

Race Condition

- **Assume** counter **is initially 5. One interleaving of statements is:**

  producer: register1 = counter (***register1 = 5***)
  producer: register1 = register1 + 1 (***register1 = 6***)
  consumer: register2 = counter (***register2 = 5***)
  consumer: register2 = register2 – 1 (***register2 = 4***)
  producer: counter = register1 (***counter = 6***)
  consumer: counter = register2 (***counter = 4***)

- **The value of** count **may be either 4 or 6, where the correct result should be 5**

Synchronization with Semaphore

Producer

```
item nextProduced;

if (user_wants_to_write == 1) {
        while (counter == BUFFER_SIZE)
                ; /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        wait(S1);
        counter++;
        signal(S1);
}
```

Consumer

```
item nextConsumed;

If (user_wants_to_read == 1) {
        while (counter == 0)
                ; /* do nothing */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        wait(S1);
        counter--;
        signal(S1);
}
```

Automatic Parallelization and OpenMP

Automatic Parallelization

- An ideal compiler could be able to manage everything about parallelization
  - From identifying parallel parts to running them in parallel
- However, current compilers can only automatically parallelize loops
  - Just as another compiler optimization
  - Without some help from the developer, compilers will rarely be able to exploit all the parallelism

Parallelization with Autopar

The Solaris Studio C compiler

```
void setup( double *vector, int length )
{
  int i;
  for ( i=0; i<length; i++ )        // Line 6
  {
     vector[i] += 1.0;
  }
}

int main()
{
  double *vector;
  vector = (double*)malloc( sizeof(double)*1024*1024 );
  for ( int i=0; i<1000; i++ )      // Line 16
  {
    setup( vector, 1024*1024 );
  }
}
```

Parallelization with Autopar

- The compiler can parallelize the first loop

```
$ cc -g -xautopar -xloopinfo -O -c omp_vector.c
"setvector.c", line 6: PARALLELIZED, and serial version generated
"setvector.c", line 16: not parallelized, call may be unsafe
```

- It is not able to parallelize the second loop
  - Because of the function call in the second loop

```
void setup( double *vector, int length )
{
  int i;
  for ( i=0; i<length; i++ )       // Line 6
  {
     vector[i] += 1.0;
  }
}

int main()
{
  double *vector;
  vector = (double*)malloc( sizeof(double)*1024*1024 );
  for ( int i=0; i<1000; i++ )     // Line 16
  {
    setup( vector, 1024*1024 );
  }
}
```

No standard way to denote that a function call can be safely parallelized

```
for ( int i=0; i<1000; i++ )     // Line 16
{
  setup( vector, 1024*1024 );
}
```

Parallelization with the Intel Compiler

➢ Example code for matrix multiplication

```
void matVec( double **mat, double *vec, double *out,
             int *row, int *col )
{
  int i, j;
  for ( i=0; i<*row; i++ )          // Line 5
  {
     out[i]=0;
     for ( j=0; j<*col; j++ )       // Line 8
     {
       out[i] += mat[i][j] * vec[j];
     }
  }
}
```

```
$ cc -g -xautopar -xloopinfo -O -c fploop.c
"fploop.c", line 5: not parallelized, not a recognized for loop
"fploop.c", line 8: not parallelized, not a recognized for loop
```

Parallelization with the Intel Compiler

- **Reason for the failure of parallelization**
  - The possibility of aliasing between the store to out[i] and the values of the loop bound, *row and *col
  - For the compiler, it is safe to assume aliasing

```
void matVec( double **mat, double *vec, double *out,
             int row, int col )
{
  int i, j;
  for ( i=0; i<row; i++ )          // Line 5
  {
     out[i]=0;
     for ( j=0; j<col; j++ )       // Line 8
     {
       out[i] += mat[i][j] * vec[j];
     }
  }
}
```

Modified code

Still exists the possibility of aliasing due to out, mat, and vec.

OpenMP

- **OpenMP is the most commonly used language extension for parallelism**
  - **It defines an API that enables a developer to add directives for the compiler to parallelize the application**
  - **It follows a fork-join type model**

```
void calc( double* array1, double * array2, int length )
{
  #pragma omp parallel for
  for ( int i=0; i<length; i++ )
  {
    array1[i] += array2[i];
  }
}
```

Parallelization of a loop with OpenMP

Parallel Region

- When a parallel region is encountered, the work will be divided between a group of threads

```
void main()
{
   #pragma omp parallel
   {
     printf( "Thread\n" );
   }
}
```

```
$ cc -O -xopenmp -xloopinfo omptest.c
$ export OMP_NUM_THREADS=2
$ ./a.out
Thread
Thread
```

- The # of threads is set by the environment variable OMP_NUM_THREADS, or by the application at runtime by calls into the runtime support library

Variable Scoping

- The variables can be scoped in two ways
  - Shared: each thread shares the same variable
  - Private: each thread gets its own copy of the variable

```
void calc( double * array1, double * array2, int length )
{
   for( int i=0; i<length; i++ )
   {
     array1[i] += array2[i];
   }
}
```

The scoping rules in OpenMP are quite complex

- The simplified summary of OpenMP rules
  - The loop induction variable as being private
  - Variables defined in the parallel code as being private
  - Variables defined outside the parallel region as being shared

Explicit Variable Scoping

- The simplified rules should be appropriate in simple situations, but may not be appropriate in complex ones

- In these situations, it is better to manually define the variable scoping

```
void calc( double* array1, double * array2, int length )
{
  int i;
  #pragma omp parallel for private(i) shared(length, array1, array2)
  for ( i=0; i<length; i++ )
  {
    array1[i] += array2[i];
  }
}
```

Reductions

- **Not all variables can be scoped as either shared or private**

```
double calc( double* array, int length )
{
  double total = 0.0;
  for ( int i=0; i<length; i++ )
  {
    total += array[i];
  }
  return total;
}
```

  - **Can we scope `total` as shared?**
    - What if we use a semaphore to serialize access to `total`

Reductions

- OpenMP allows for a reduction operation
  - Each thread has a private copy of the reduction variable in the parallel region
  - At the end of the parallel region, the private copies are combined to produce the final result

```
double calc( double* array, int length )
{
  double total = 0.0;
  #pragma omp parallel for reduction( +: total )
  for ( int i=0; i<length; i++ )
  {
    total += array[i];
  }
  return total;
}
```

Other operations include subtraction; multiplication; the bitwise operations AND, OR, and XOR; and the logical operations AND and OR.

Static Work Scheduling

- The default scheduling for a parallel for loop is called static scheduling
  - The iterations are divided evenly, in chunks of consecutive iterations, between the threads
- In some cases, a different amount of work can be performed in each iteration
  - Consequently, both threads may complete the same number of iterations, but one may have more work to do in those iterations

Dynamic Work Scheduling

- The work is divided into multiple chunks of work and each thread takes the next chunk of work when it completes a chunk of work

```
int main()
{
  double data[200][100];
  int i, j;
  #pragma omp parallel for private(i,j) shared(data) schedule(dynamic)
  for ( int i=0; i<200; i++ )
  {
    for ( int j=0; j<200; j++ )
    {
      data[i][j] = calc(i+j);
    }
  }
  return 0;
}
```

GPGPU

GPGPU

- **General-Purpose Computing on GPUs**
  - **GPGPU is the utilization of a GPU to perform computation in applications traditionally handled by CPU**
  - **GPUs can be viewed as compute co-processors**

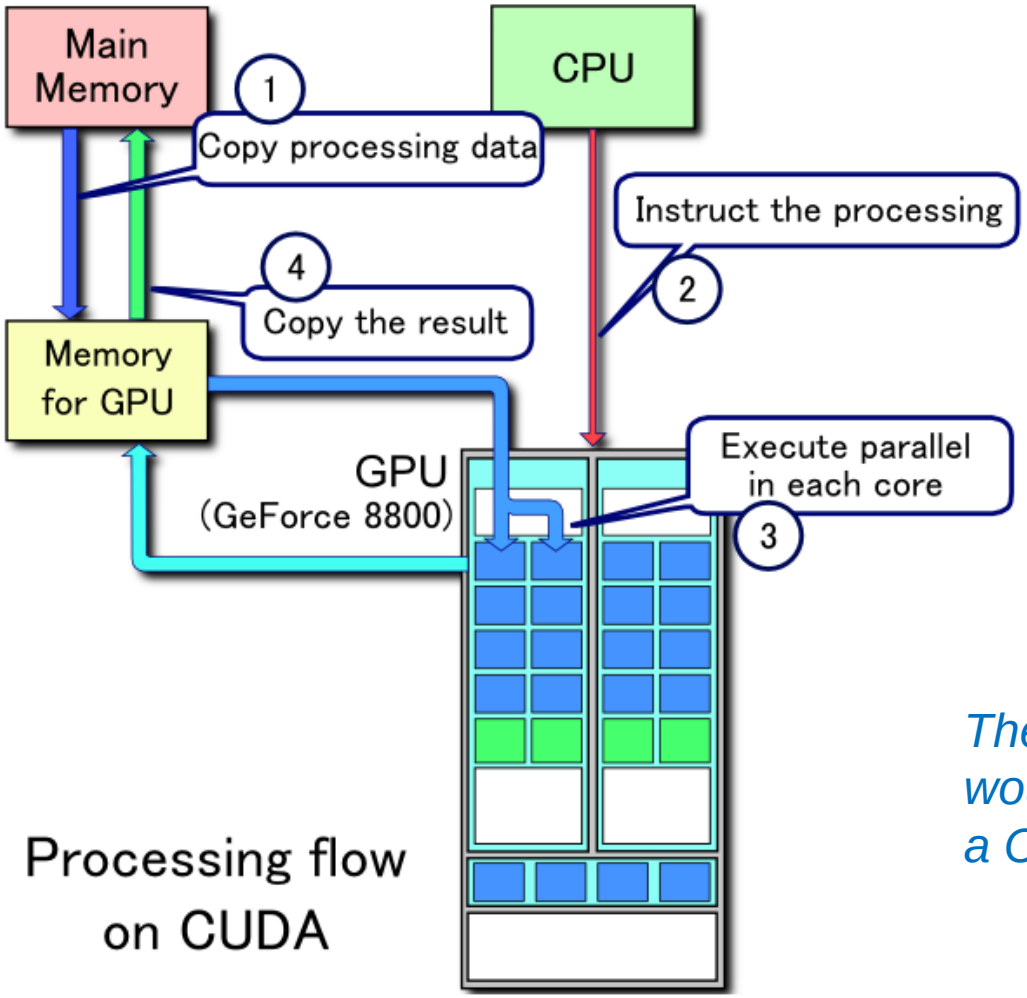
- **GPGPU technology**
  - **Compute Unified Device Architecture (CUDA) from Nvidia**
  - **Open Computing Language (OpenCL) from ATI and Nvidia**
    - Adopted by Apple, Intel, Qualcomm, …

Two Considerations in GPGPU

- **Different instruction sets**
  - **GPUs do not share the instruction set of the host processor**
  - **This has to produce code for the host processor, together with code for the GPU**

- **Different address spaces**
  - **Data needs to be copied across to the GPU**
  - **The act of copying is time consuming**
    - The problem should be large enough to justify the cost of the copy operation

CUDA Processing Flow



The program written in OpenCL would look broadly similar to a CUDA program.

Simple CUDA Program

```
#include "cuda.h"

#define LEN 100000

// GPU code
__global__ void square( float *data, int length )
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  if ( index < length )
  {
     data[index] = data[index] * data[index];
  }
}

//Host code
int main()
{
  float *host_data, *gpu_data;
  int ThreadsPerBlock, Blocks;

  // Allocate memory
  host_data = (float*)malloc( LEN * sizeof(float) );
  cudaMalloc( &gpu_data, LEN*sizeof(float) );

  // Initialize data on host
  for( int i=0; i<LEN; i++ )
  {
    host_data[i] = 2*i;
  }

  // Copy host data to GPU
  cudaMemcpy( gpu_data, host_data, LEN*sizeof(int),
              cudaMemcpyHostToDevice );

  // Perform computation on GPU
  ThreadsPerBlock = 128;
  Blocks = (int)( (LEN-1) / ThreadsPerBlock ) + 1;
  square <<<Blocks, ThreadsPerBlock>>>( gpu_data, LEN );

  // Copy GPU data back to host
  cudaMemcpy( gpu_data, host_data, LEN*sizeof(int),
              cudaMemcpyDeviceToHost);

  // Free allocated memory
  cudaFree( gpu_data );
  free( host_data );
}
```

Simple CUDA Program

```
#include "cuda.h"

#define LEN 100000

// GPU code
__global__ void square( float *data, int length )
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  if ( index < length )
  {
     data[index] = data[index] * data[index];
  }
}

//Host code
int main()
{
  float *host_data, *gpu_data;
  int ThreadsPerBlock, Blocks;

  // Allocate memory
  host_data = (float*)malloc( LEN * sizeof(float) );
  cudaMalloc( &gpu_data, LEN*sizeof(float) );

  // Initialize data on host
  for( int i=0; i<LEN; i++ )
```

The routine square( ) is executed by the GPU.

Each hardware thread on the GPU executes the same routine.

The routine main( ) is executed by the host processor.

Allocate memory on the host.

Allocate memory on the GPU.

Simple CUDA Program

Copy the data from the host to The GPU.

The bus bandwidth may be 8GB/s to 16GB/s.

Threads are arranged in groups called blocks.

The # of threads per block and the # of blocks are specified.

Once the call to square() completes, the host copies the resulting data back from the device into host memory.

```
  {
    host_data[i] = 2*i;
  }

  // Copy host data to GPU
  cudaMemcpy( gpu_data, host_data, LEN*sizeof(int),
              cudaMemcpyHostToDevice );

  // Perform computation on GPU
  ThreadsPerBlock = 128;
  Blocks = (int)( (LEN-1) / ThreadsPerBlock ) + 1;
  square <<<Blocks, ThreadsPerBlock>>>( gpu_data, LEN );

  // Copy GPU data back to host
  cudaMemcpy( gpu_data, host_data, LEN*sizeof(int),
              cudaMemcpyDeviceToHost);

  // Free allocated memory
  cudaFree( gpu_data );
  free( host_data );
}
```


Q&A