HPC programming languages

Arnaud LEGRAND, CR CNRS, LIG/INRIA/Mescal

Vincent DANJEAN, MCF UJF, LIG/INRIA/Moais

October 6th, 2014

Goals of my four lectures

Today

- Overview of a few different programming languages used in HPC (Threads, OpenMP, Cuda, OpenCL, MPI)

Next lecture (about 6 weeks later)

- Understand the internal of HPC programming model implementations
- Understand why so many HPC programming models
- Understand why mixing HPC programming models is hard
- Understand why still new HPC programming models

Next two lectures

- Focussed on work the stealing model and parallel tasks model

Threads : Posix and OpenMP

General Purpose Graphical Processor Units (GPGPU)

Message Passing Interface (MPI)

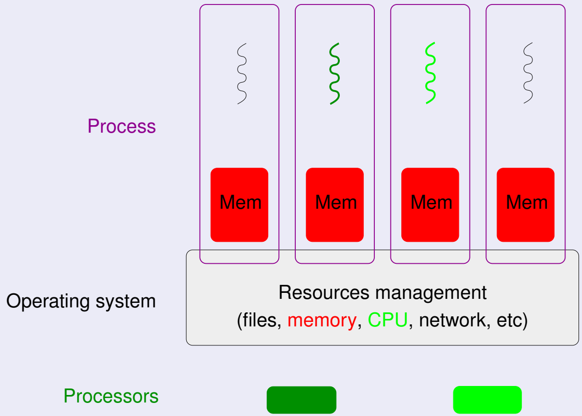
Part I

Threads : Posix and OpenMP

Outlines: Threads : Posix and OpenMP

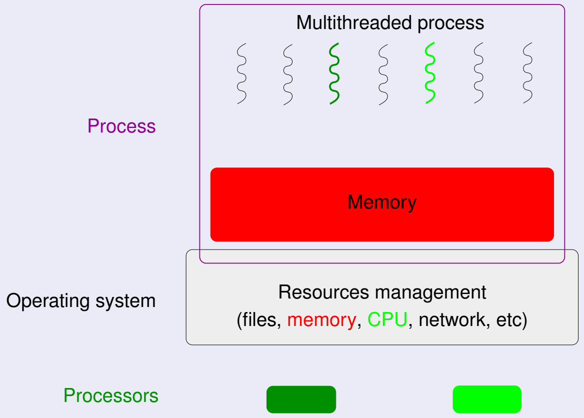
Programming on Shared Memory Parallel Machines

Using process



Programming on Shared Memory Parallel Machines

Using threads



Introduction to threads

Why threads ?

- To take profit from shared memory parallel architectures
  - SMP, hyperthreaded, multi-core, NUMA, etc. processors

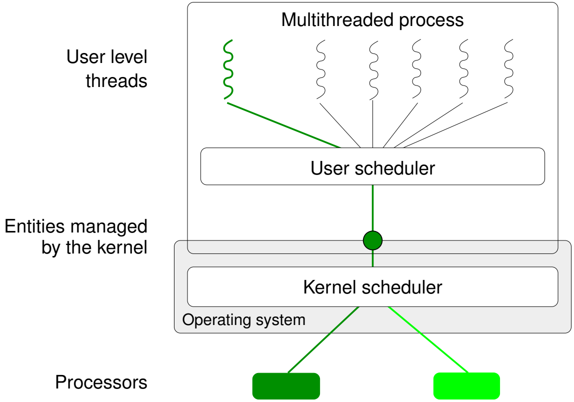
    future Intel processors: several hundreds cores
- To describe the parallelism within the applications
  - independent tasks, I/O overlap, etc.

What will use threads ?

- User application codes
  - directly (with thread libraries)

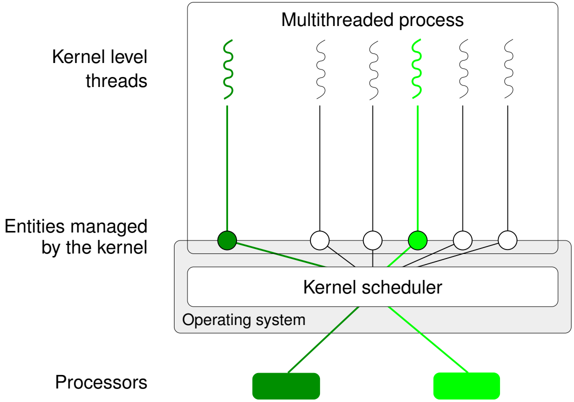
    POSIX API (IEEE POSIX 1003.1C norm) in C, C++, . . .
  - with high-level programming languages (Ada, OpenMP, . . .)
- Middleware programming environments
  - demonized tasks (garbage collector, . . .), . . .

User threads



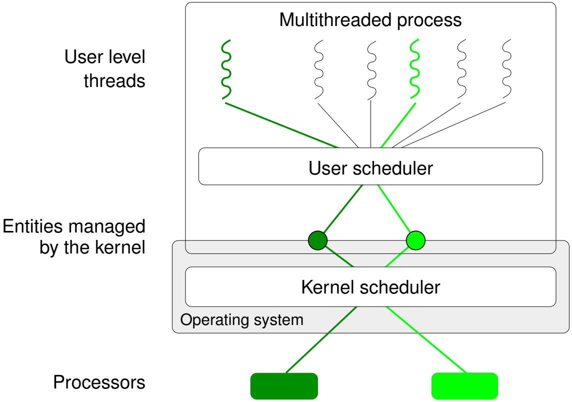
Efficiency + Flexibility + SMP - Blocking syscalls -

Kernel threads



Efficiency - Flexibility - SMP + Blocking syscalls +

Mixed models



Efficiency + Flexibility + SMP + Blocking syscalls limited

Thread models characteristics

	Characteristics			
Library	Efficiency	Flexibility	SMP	Blocking syscalls
User	+	+	-	-
Kernel	-	-	+	+
Mixed	+	+	+	limited

Summary

Mixed libraries seems more attractive however they are more complex to develop. They also suffer from the blocking system call problem.

Outlines: Threads : Posix and OpenMP

Normalisation of the thread interface

Before the norm

- each Unix had its (slightly) incompatible interface
- but same kinds of features was present

POSIX normalization

- IEEE POSIX 1003.1c norm (also called POSIX threads norm)
- Only the API is normalised (not the ABI)
  - POSIX thread libraries can easily be switched at source level but not at runtime
- POSIX threads own
  - processor registers, stack, etc.
  - signal mask
- POSIX threads can be of any kind (user, kernel, etc.)

Basic POSIX Thread API

Creation/destruction

- int **pthread_create**(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg)
- void **pthread_exit**(void *value_ptr)
- int **pthread_join**(pthread_t thread, void **value_ptr)

Synchronisation (semaphores)

- int **sem_init**(sem_t *sem, int pshared, unsigned int value)
- int **sem_wait**(sem_t *sem)
- int **sem_post**(sem_t *sem)
- int **sem_destroy**(sem_t *sem)

Basic POSIX Thread API (2)

Synchronisation (mutex)

- int **pthread_mutex_init**(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)
- int **pthread_mutex_lock**(pthread_mutex_t *mutex)
- int **pthread_mutex_unlock**(pthread_mutex_t *mutex)
- int **pthread_mutex_destroy**(pthread_mutex_t *mutex)

Synchronisation (conditions)

- int **pthread_cond_init**(pthread_cond_t *cond, const pthread_condattr_t *attr)
- int **pthread_cond_wait**(pthread_cond_t *cond, pthread_mutex_t *mutex)
- int **pthread_cond_signal**(pthread_cond_t *cond)

Basic POSIX Thread API (3)

Per thread data

- `int` **`pthread_key_create`**`(pthread_key_t *key, void (*destr_function) (void*))`
- `int` **`pthread_key_delete`**`(pthread_key_t key)`
- `int` **`pthread_setspecific`**`(pthread_key_t key, const void *pointer)`
- `void *` **`pthread_getspecific`**`(pthread_key_t key)`

Basic POSIX Thread API (3)

Per thread data

- `int` **`pthread_key_create`**`(pthread_key_t *key, void (*destr_function) (void*))`
- `int` **`pthread_key_delete`**`(pthread_key_t key)`
- `int` **`pthread_setspecific`**`(pthread_key_t key, const void *pointer)`
- `void *` **`pthread_getspecific`**`(pthread_key_t key)`

The new `__thread` C keyword

- used for global per-thread variables
- compiler + linker support at compile + execute time
- libraries can have efficient per-thread variables without disturbing the application
- `http://www.akkadia.org/drepper/tls.pdf`

Outlines: Threads : Posix and OpenMP

What is OpenMP?

An API to parallelize a program

explicitly, with threads, with shared memory

Contents of OpenMP

- compiler directives
- runtime library routines
- environment variables

OpenMP abbreviation

Short version Open Multi-Processing

Long version Open specifications for Multi-Processing via collaborative work between interested parties from the hardware and software industry, government and academia.

What is not OpenMP?

- not designed to manage distributed memory parallel systems
- implementation can vary depending on the vendor
- no optimal performance guarantee
- not a checker for data dependencies, deadlock, etc.
- not a checker for code correction
- not a automatic parallelization tool
- not designed for parallel I/O

More information

`https://computing.llnl.gov/tutorials/openMP/`
`http://openmp.org/wp/`

Goals of OpenMP

Standardization

- target a variety of shared memory architectures/platforms
- supported by lots of hardware and software vendors

Lean and Mean (less pertinent with last releases)

- simple and limited set of directives
- 3 or 4 directives enough for classical parallel programs

Ease of Use

- allows to incrementally parallelize a serial program
- allows both coarse-grain and fine-grain parallelism

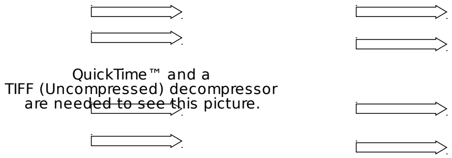
Portability (API in C/C++ and Fortran)

- public forum for API and membership
- most major platforms have been implemented

Outlines: Threads : Posix and OpenMP

Fork-Join Model

- Program begins with a Master thread
- Fork: Teams of threads created at times during execution
- Join: Threads in the team synchronize (barrier) and only the master thread continues execution


QuickTime™ and a
TIFF (Uncompressed) decompressor
are needed to see this picture.

Courtesy of Henri Casanova

OpenMP and #pragma

- **One needs to specify blocks of code that are executed in parallel**
- **For example, *a parallel section*:**

```
#pragma omp parallel [clauses]
```

  - Defines a section of the code that will be executed in parallel
  - The “clauses” specify many things including what happens to variables
  - All threads in the section execute the same code

First “Hello World” example

```
#include <omp.h>
int main(){
print(“Start\n”);
#pragma omp parallel
 { // note the {
   printf(“Hello World\n”);
 } // note the }
/* Resume Serial Code */
 printf(“Done\n”);
}
```

```
% my_program
Start
Hello World
Hello World
Hello World
Done
```

Courtesy of Henri Casanova

First “Hello World” example

```
#include <omp.h>
int main(){
print(“Start\n”);
#pragma omp parallel
 {
   printf(“Hello World\n”);
 }
/* Resume Serial Code */
 printf(“Done\n”);
}
```

```
% my_program
Start
Hello World
Hello World
Hello World
Done
```

- Questions
  - How many threads?
  - This is not useful because all threads do exactly the same thing
  - Conditional compilation?

How Many Threads?

- **Set via an environment variable**

  ```
  setenv OMP_NUM_THREADS 8
  ```

  - Bounds the maximum number of threads
- **Set via the OpenMP API**

  ```
  void omp_set_num_threads(int number);
  int omp_get_num_threads();
  ```

- **Typically, a function of the number of processors available**
  - We often take the number of threads identical to the number of processors/cores

Courtesy of Henri Casanova

Threads Doing Different Things

```
#include <omp.h>
int main()  {
  int iam =0, np = 1;
#pragma omp parallel private(iam, np)
  {
    np = omp_get_num_threads();
    iam = omp_get_thread_num();
    printf(“Hello from thread %d out of %d threads\n”, iam,
   np);
  }
}
```

```
% setenv OMP_NUM_THREADS 3
% my_program
Hello from thread 0 out of 3
Hello from thread 1 out of 3
Hello from thread 2 our of 3
```

Courtesy of Henri Casanova

Conditional Compilation

- The _OPENMP variable is defined if the code is compiled with OpenMP

```
#ifdef _OPENMP
#include <omp.h>
#ifdef _OPENMP
int main()  {
  int iam = 0, np = 1;
#pragma omp parallel private(iam, np)
  {
#ifdef _OPENMP
    np = omp_get_num_threads();
    iam = omp_get_thread_num();
#endif
    printf("Hello from thread %d out of %d threads\n", iam, np);
  }
}
```

- This code will work serially!

Data Scoping and Clauses

- **Shared: all threads access the single copy of the variable, created in the master thread**
  - it is the responsibility of the programmer to ensure that it is shared appropriately
- **Private: a volatile copy of the variable is created for each thread, and discarded at the end of the parallel region (but for the master)**
- **There are other variations**
  - firstprivate: initialization from the master’s copy
  - lastprivate: the master gets the last value updated by the last thread to do an update
  - and several others

  (Look in the on-line material if you’re interested)

Work Sharing directives

- **We have seen the concept of a parallel region, which is a brute-force SPMD directive**
- **Work Sharing directives make it possible to have threads "share work" *within a parallel region.***
  - For Loop
  - Sections
  - Single

For Loops

QuickTime™ and a
TIFF (Uncompressed) decompressc
are needed to see this picture.

- **Share iterations of the loop across threads**
- **Represents a type of "data parallelism"**
  - do the same operation on pieces of the same big piece of data
- **Program correctness must NOT depend on which thread executes which iteration**
  - No ordering!

For Loop Example

```
#include <omp.h>
#define N 1000
main ()  {
  int i, chunk;float a[N], b[N], c[N];
  for (i=0; i < N; i++)
    a[i] = b[i] = i * 1.0;
#pragma omp parallel shared(a,b,c) private(i)
  {
  #pragma omp for schedule(dynamic)
  for (i=0; i < N; i++)
    c[i] = a[i] + b[i];
  }  /* end of parallel section */
}
```

Sections

QuickTime™ and a
TIFF (Uncompressed) decompressc
are needed to see this picture.

- Breaks work into separate sections
- Each section is executed by a thread
- Can be used to implement "task parallelism"
  - do different things on different pieces of data
- If more threads than sections, then some are idle
- If fewer threads than sections, then some sections are seralized

Section Example

```
#include <omp.h>
#define N 1000
main (){
  int i;float a[N], b[N], c[N];
  for (i=0; i < N; i++)
    a[i] = b[i] = i * 1.0;
#pragma omp parallel shared(a,b,c) private(i)
  {
  #pragma omp sections
    {
    #pragma omp section
      {
      for (i=0; i < N/2; i++)
        c[i] = a[i] + b[i];
      }
    #pragma omp section
      {
      for (i=N/2; i < N; i++)
        c[i] = a[i] + b[i];
      }
    }  /* end of sections */
  }  /* end of parallel section */
}
```

Section #1

Section #2

Courtesy of Henri Casanova

Single

QuickTime™ and a
TIFF (Uncompressed) decompressc
are needed to see this picture.

- **Serializes a section of code within a parallel region**
- **Sometimes more convenient than terminating a parallel region and starting it later**
  - especially because variables are already shared/private, etc.
- **Typically used to serialize a small section of the code that's not thread safe**
  - e.g., I/O

Courtesy of Henri Casanova

Combined Directives

- It is cumbersome to create a parallel region and *then* create a parallel for loop, or sections, just to terminate the parallel region
- Therefore OpenMP provides a way to do both at the same time
  - #pragma omp parallel for
  - #pragma opm parallel sections

Synchronization and Sharing

- When variables are shared among threads, OpenMP provides tools to make sure that the sharing is correct
- Why could things be unsafe?

```
int x = 0;
#pragma omp parallel sections shared(x)
  {
  #pragma omp section
  x = x + 1
  #pragma omp section
  x = x + 2
  }
```

Synchronization directive

- `#pragma omp master`
  - Creates a region that only the master executes
- `#pragma omp critical`
  - Creates a critical section
- `#pragma omp barrier`
  - Creates a “barrier”
- `#pragma omp atomic`
  - Create a “mini” critical section

Critical Section

```
#pragma omp parallel for \
                  shared(sum)
  for(i = 0; i < n; i++){
    value = f(a[i]);
    #pragma omp critical
    {
      sum = sum + value;
    }
  }
```

Barrier

```
if (x == 2) {
  #pragma omp barrier
}
```

- All threads in the current parallel section will synchronize
  - they will all wait for each other at this instruction
- Must appear within a basic block

Atomic

```
#pragma omp atomic
  i++;
```

- **Only for some expressions**
  - x = expr   (no mutual exclusion on expr evaluation)
  - x++
  - ++x
  - x--
  - --x
- **Is about atomic access to a memory location**
- **Some implementations will just replace atomic by critical and create a basic blocks**
- **But some may take advantage of cool hardware instructions that work atomically**

Scheduling

- **When I talked about the parallel for loops, I didn't say how the iterations were shared among threads**
- **Question: I have 100 iterations. I have 5 threads. Which thread does which iteration?**
- **OpenMP provides many options to do this**
- **Choice #1: Chunk size**
  - a way to group iterations togethers
  - e.g., chunk size = 2 means that iterations are grouped 2 by 2
  - allows to avoid prohibitive overhead in some situations
- **Choice #2: Scheduling Policy**

Loop Scheduling in OpenMP

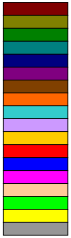
- `static`:
  - Iterations are divided into pieces of a size specified by chunk.
  - The pieces are statically assigned to threads in the team in a roundrobin fashion in the order of the thread number.
- `dynamic`:
  - Iterations are broken into pieces of a size specified by chunk.
  - As each thread finishes a piece of the iteration space, it dynamically obtains the next set of iterations.
- `guided`:
  - The chunk size is reduced in an exponentially decreasing manner with each dispatched piece of the iteration space.
  - chunk specifies the smallest piece (except possibly the last).
- `Default schedule: implementation dependent.`

Example

```
int chunk = 3

#pragma omp parallel for \
       shared(a,b,c,chunk) \
       private(i) \
       schedule(static,chunk)
  for (i=0; i < n; i++)
    c[i] = a[i] + b[i];}
```

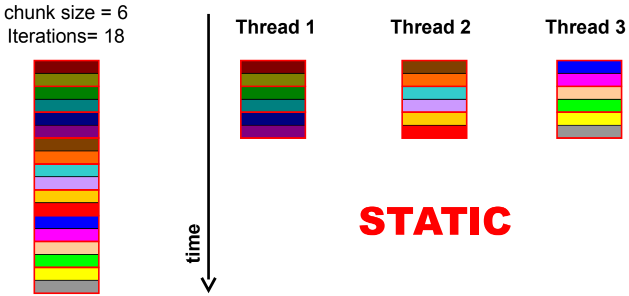
OpenMP Scheduling

chunk size = 6
Iterations = 18



Courtesy of Henri Casanova

OpenMP Scheduling



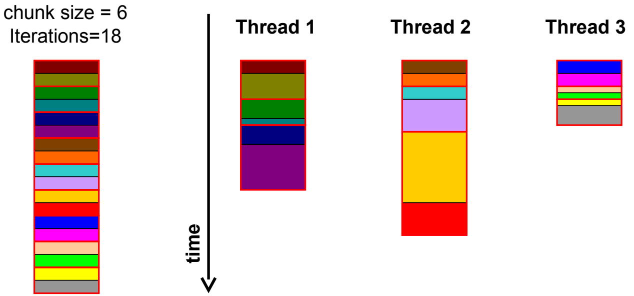
Work in different iterations is identical.

Courtesy of Henri Casanova

So, isn’t static optimal?

- **The problem is that in may cases the iterations are not identical**
  - Some iterations take longer to compute than other
- **Example #1**
  - Each iteration is a rendering of a movie’s frame
  - More complex frames require more work
- **Example #2**
  - Each iteration is a “google search”
  - Some searches are easy
  - Some searches are hard
- **In such cases, load unbalance arises**
  - which we know is bad

OpenMP Scheduling

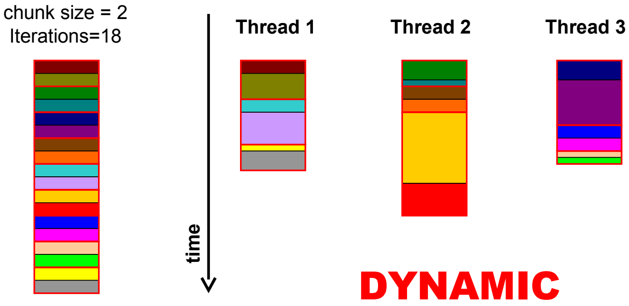


STATIC

Work in different iterations is NOT identical.

Courtesy of Henri Casanova

OpenMP Scheduling



Work in different iterations is NOT identical.

Courtesy of Henri Casanova

So isn’t dynamic optimal?

- **Dynamic scheduling with small chunks causes more overhead than static scheduling**
  - In the static case, one can compute what each thread does at the beginning of the loop and then let the threads proceed unhindered
  - **In the dynamic case, there needs to be some type of communication: “I am done with my 2 iterations, which ones do I do next?”**
    - Can be implemented in a variety of ways internally
- **Using dynamic scheduling with a large chunk size leads to lower overhead, but defeats the purpose**
  - with fewer chunks, load-balancing is harder
- **Guided Scheduling: best of both worlds**
  - start with large chunks, ends with small ones

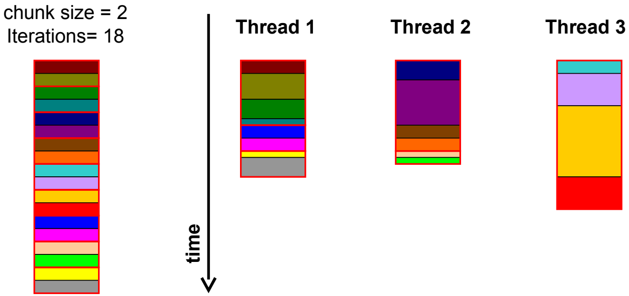
Courtesy of Henri Casanova

Guided Scheduling

- The chunk size is reduced in an exponentially decreasing manner with each dispatched piece of the iteration space.

- chunk specifies the smallest piece (except possibly the last).

OpenMP Scheduling



Guided

3 chunks of size 4
2 chunks of size 2

Courtesy of Henri Casanova

What should I do?

- Pick a reasonable chunk size
- Use static if computation is evenly spread among iterations
- Otherwise probably use guided

How does OpenMP work?

- The pragmas allow OpenMP to build some notion of structure of the code
- And then, OpenMP generates pthread code!!
  - You can see this by running the **nm** command on your executable
- OpenMP hides a lot of the complexity
- But it doesn't have all the flexibility
- The two are used in different domains
  - OpenMP: “scientific applications”
  - Pthreads: “system” applications
- But this distinction is really arbitrary IMHO

More OpenMP Information

- OpenMP Homepage: http://www.openmp.org/
- On-line OpenMP Tutorial:

  http://www.llnl.gov/computing/tutorials/openMP/

Part II

General Purpose Graphical Processor Units (GPGPU)

Outlines: General Purpose Graphical Processor Units (GPGPU)

Credits

Most of theses slides come from a Tutorial on GPU programming made last year during the Compas'2013 conference

I prepared this tutorial with João V. F. LIMA (Cuda part) and Brice VIDEAU (OpenCL part)

Parallel Programming with GPU

GPGPU: General Purpose Graphic Processing Unit

- very good ratio GFlops/price and GFlops/Watt
- GPU Tesla C2050 from NVidia : about 300 GFlops in double precision
- specialized hardware architecture:
  classical programming does not work

Two leading environments

Cuda	specific to NVidia, can use all the features of NVidia cards. Works only with NVidia GPU.
OpenCL	norm (not implementation) supported by different vendors (AMD, NVidia, Intel, Apple, etc.) Target GPUs but also CPUs.

Very similar programming concepts

Cuda and OpenCL bases

Part 1: device programs

- C code with restriction and extension (memory model, vector types, etc.)
- run in parallel by lots of threads on the targeted hardware
- functions to be run are called kernels

Part 2: host programs

- API in C/C++
- manage memory transfers
- manage kernel launches (compilations and runs)

Outlines: General Purpose Graphical Processor Units (GPGPU)

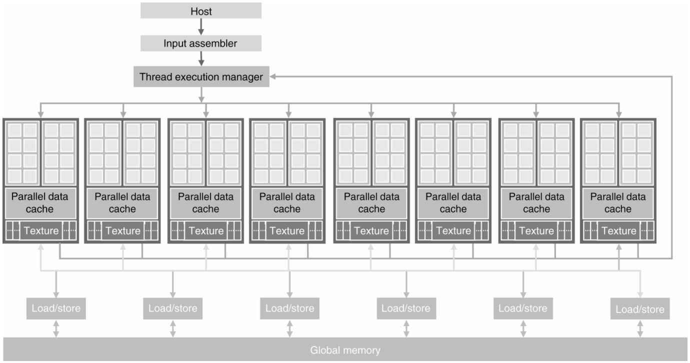
Introduction

- Terminology:
  - **Host** – The CPU and its memory (host memory)
  - **Device** – The GPU and its memory (device memory)
  - **Kernel** – C functions executed N times in parallel (CUDA threads)
- Simple CUDA API:
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - `cuda*()`
- In all examples, we assume CUDA 4.1 or later

`http://developer.nvidia.com`

CUDA-capable GPU Architecture

- The unit of execution is a **streaming processor** (SP) core
- SPs are grouped as **streaming multiprocessors** (SM)
- **Single Instruction Multiple Thread** (SIMT) architecture



Compute Capability

- Describe the architecture version
- Defined by:
  - Major revision number – core architecture
  - Minor revision number – incremental improvement
- Architecture families by major revision
  - Revision 1 – *Tesla* architecture (GeForce GTX 280)
  - Revision 2 – *Fermi* architecture (GeForce GTX 480)
  - Revision 3 – *Kepler* architecture (GeForce GTX 680)

`http://developer.nvidia.com/cuda-gpus`

Thread Hierarchy

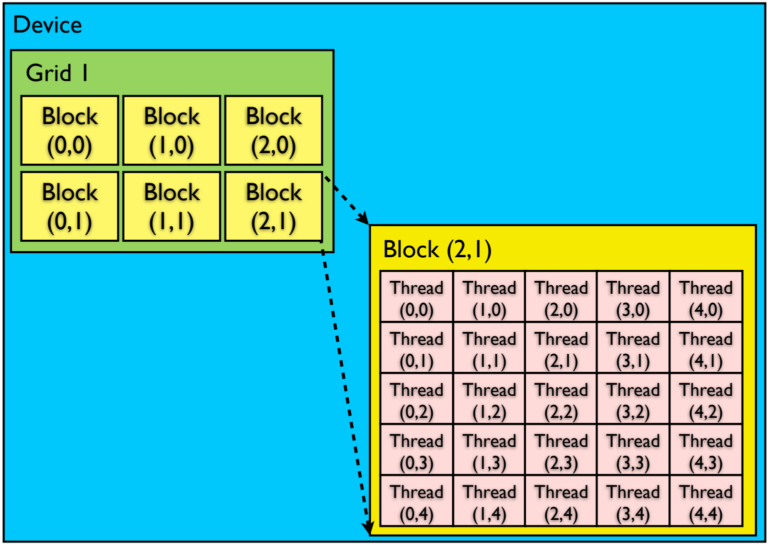
- CUDA threads are grouped in 1D, 2D, or 3D **thread blocks**
  - Block size is fixed on each kernel launch
- Blocks are organized into a 1D, 2D, or 3D **grid**
  - Blocks execute in parallel on each SM

Grid size

The number of thread blocks in a grid is usually dictated by the size of the data being processed. This number may be greater than available SMs.

- **Note:** there is a limit of threads and dimensions
  - Up to 1024 threads per block on current GPUs
  - See `deviceQuery` from CUDA SDK examples

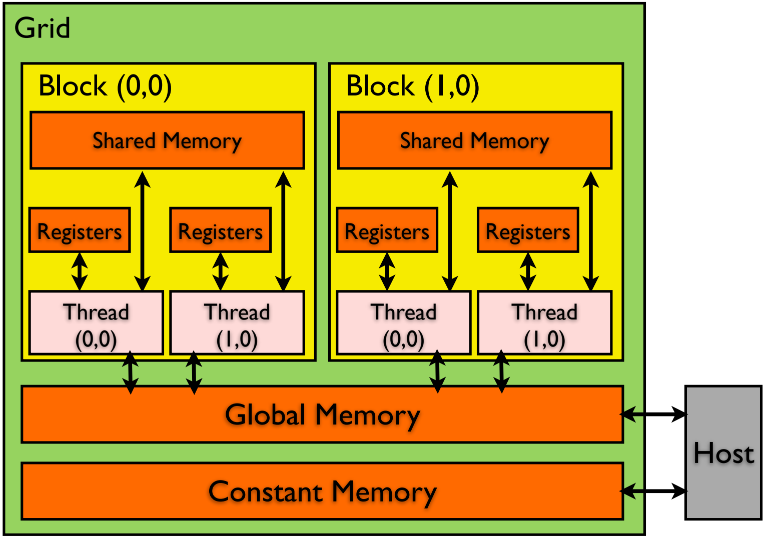
Thread Hierarchy



Memory Hierarchy

- **Global memory**
  - Device memory
- **Shared memory**
  - On-chip memory (SM)
- **Constant memory**
  - Device memory
- **Local memory**
  - Global memory (device)
  - Only occur if kernel uses more register than available

Memory Hierarchy



Hello world

Hello world in CUDA

```
__global__ void mykernel(void)
{
}

int main(void)
{
  mykernel<<<1,1>>>();
  printf("Hello World!\n");
  return 0;
}
```

Hello world

Compilation (NVCC)

```
# simple command line
$ nvcc -o hello_world hello_world.cu
$ ./hello_world
Hello World!

# with compiler options
$ nvcc --compiler-options "-Wall"  \
  -o hello_world hello_world.cu -lm
```

CUDA C/C++ Basics

- CUDA programs are composed (basically) of:

Hello world in CUDA

1. **(optional) Select device (default: 0)**
2. **Allocate device memory for inputs and outputs**
3. **Copy data to the device**
4. **Launch kernel on device**
5. **Copy back results to host memory**
6. **Free device memory**

Add One

Add One in CUDA (Kernel)

```
__global__ void addone(float* A)
{
  int i = threadIdx.x;
  A[i] = A[i] + 1;
}
```

Add One

Add One in CUDA (Host)

```
float *A, *d_A;
int N = 20;

A = (float*) malloc(N * sizeof(float));
cudaMalloc((void**)&d_A, N * sizeof(float));

cudaMemcpy(d_A, A, N * sizeof(float),
  cudaMemcpyHostToDevice);
addone<<<1, N>>>(d_A); // N CUDA threads
cudaMemcpy(A, d_A, N * sizeof(float),
  cudaMemcpyDeviceToHost);

cudaFree(d_A); free(A);
```

Memory Transfer

- **cudaMemcpy(void* dst, const void* src, size_t nbytes, cudaMemcpyKind direction);**
- cudaMemcpyKind available values:
    - cudaMemcpyHostToDevice
    - cudaMemcpyDeviceToHost
    - cudaMemcpyHostToHost
    - cudaMemcpyDeviceToDevice
- Some alternatives
    - **cudaMemcpy2D** – for 2D arrays
    - **cudaMemset** – initilizes or sets device memory to a value
    - **cudaMemcpyAsync** – asynchronous transfer
        - More details later (Asynchronous execution)

Kernel Launch

- **dim3** – three integers (1D, 2D, or 3D)
    - dim3(x, y, z)
- **mykernel**<<<**dim3 grid, dim3 block, size_t nbytes, cudaStream_t stream**>>> **(...)**
    - **grid** specifies the dimension and size of the grid
    - **block** specifies the dimension and size of each block
    - **nbytes** is the number of in shared memory per block
    - **stream** speficies the execution stream
- Use of `nvcc` mandatory
    - Separates GPU from host code

Thread and Grid Index

- Built-in variables in device code (kernel):
  - `dim3 gridDim;` – grid dimension
  - `dim3 blockDim;` – block dimension
  - `dim3 blockIdx;` – block index
  - `dim3 threadIdx;` – thread index

CUDA 2D index (Kernel)

```
int x = threadIdx.x + blockDim.x * blockIdx.x;
int y = threadIdx.y + blockDim.y * blockIdx.y;
```

Add One with Theads and Blocks

Add One in 2D

```
__global__ void addone(float* A, int N) {
  int x = threadIdx.x + blockDim.x * blockIdx.x;
  int y = threadIdx.y + blockDim.y * blockIdx.y;
  int index = x + y * blockDim.x * gridDim.x;

  if( index < N )
    A[index] = A[index] + 1;
}

dim3 block_dim( 64, 64 );
dim3 grid_dim( (N+64-1)/64, (N+64-1)/64 );

addone<<<grid_dim, block_dim>>>(d_A, N);
```

Error Checking

- All runtime functions return an error code (**cudaError_t**)
- **cudaGetLastError()** – returns the last error from a runtime call
- **cudaGetErrorString(cudaError_t error)** – returns the message string from an error code

Error checking (Host)

```
cudaError_t err = cudaGetLastError();
if( err != cudaSuccess )
  printf("CUDA Error (%d): %s\n", err,
    cudaGetErrorString(err) );
```

Asynchronous Memory Transfer

- **cudaMemcpyAsync(void* dst, const void* src, size_t nbytes, cudaMemcpyKind direction, cudaStream_t stream);**
- Requirements:
  - Compute capability 1.1 or higher
  - Page-locked host memory
  - `stream != 0` (default stream)

Page-locked Memory

- Page-locked – non-pageable host memory (pinned)
  - Not swapped by the OS
  - Consuming too much may reduces system performace
- Host interface
  - **cudaHostAlloc** – allocates page-locked memory on the host
  - **cudaFreeHost** – frees page-locked host memory
  - **cudaHostRegister** – register a host memory for use
  - **cudaHostUnregister** – unregister a host memory

Page-locked Memory

CUDA Host Memory

```
float* hostPtr;

cudaHostAlloc((void**)&hostPtr, N * sizeof(float),
  cudaHostAllocDefault);
cudaFreeHost(hostPtr);

hostPtr = (float*)malloc(N * sizeof(float));
cudaHostRegister(hostPtr, N * sizeof(float),
  cudaHostRegisterPortable);
cudaHostUnregister(hostPtr);
```

Streams

A stream is a sequence of commands that execute in order
Different streams may execute concurrently (not guaranteed)

CUDA Streams (Host)

```
cudaStream_t stream;
cudaStreamCreate(&stream);

cudaMemcpyAsync(d_A, A, N * sizeof(float),
  cudaMemcpyHostToDevice, stream);
addone<<<grid, threads, 0, stream>>>(d_A);
cudaMemcpyAsync(A, d_A, N * sizeof(float),
  cudaMemcpyDeviceToHost, stream);
cudaStreamSynchronize(stream);

cudaStreamDestroy(stream);
```

Events

- A way to monitor the device's progress
- Events allow to perform accurate timing

CUDA Events (Host)

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start, stream);
addone<<<grid, threads, 0, stream>>>(d_A);
cudaEventRecord(stop, stream);
cudaEventSynchronize(stop);

float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
```

Synchronization

- All operations on stream 0 (default) are synchronous

Synchronization Functions (Host)

```
// Synchronize all current GPU operations
cudaDeviceSynchronize(void);

cudaStreamSynchronize(cudaStream_t);

cudaEventSynchronize(cudaEvent_t);

// All future work into stream wait the end of even
cudaStreamWaitEvent(cudaStream_t, cudaEvent_t, int);
```

Control Flow

- A SM creates, manages, schedules, and executes **warps**
  - Groups of 32 parallel threads
  - A SM partitions blocks in warps and schedules them
  - Availble at build-in variable `warpSize`
- Branch divergence occurs only within a warp
  - Threads within a single warps take different paths
  - Case of *if else* conditional statements
  - A warp executes serially each branch (`if` and `else`)
- Different warps execute independently

Control Flow

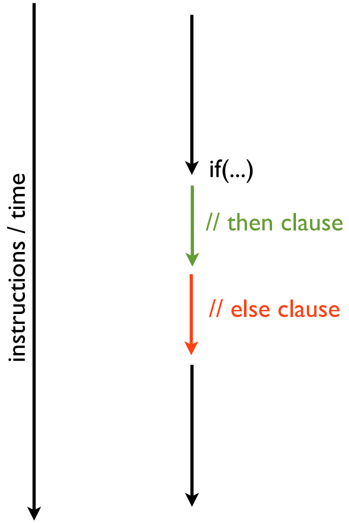
Example with divergence

```
if( threadIdx.x > 2 ) { ... }
else { ... }
```

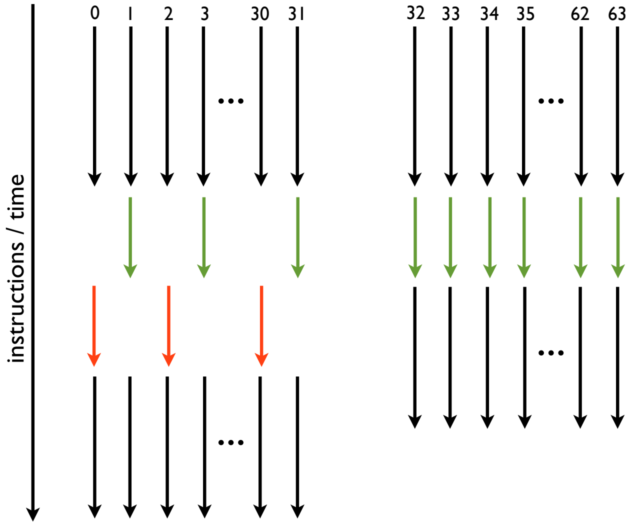
Example without divergence

```
if( (threadIdx.x/warpSize) > 2 ) { ... }
else { ... }
```

Control Flow



Control Flow



Overlap of Data Transfer and Kernel Execution

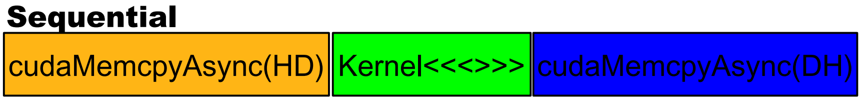
- Perform copies and kernel execution concurrently
- Compute Capability 1.1 and higher
  - `asyncEngineCount` device property greater than 0
  - `asyncEngineCount` is 1 for 1.x capability
  - `asyncEngineCount` may be 2 for 2.x capability

Overlap of Data Transfer and Kernel Execution

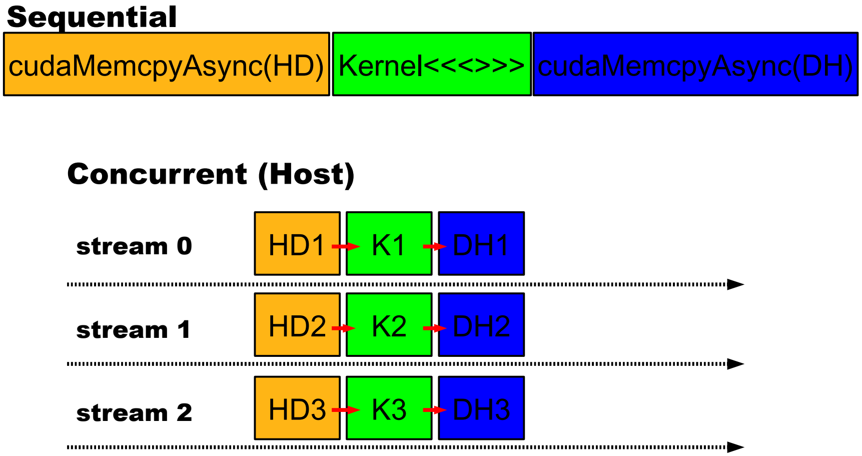
Overlapping

```
for(int i= 0; i < 3; i++)
  cudaMemcpyAsync(devicePtr + i*size,
    hostPtr + i*size, size,
    cudaMemcpyHostToDevice, stream[i]);

for(int i= 0; i < 3; i++)
  addone<<<grid, threads, 0, stream[i]>>>
     (devicePtr + i*size);

for(int i= 0; i < 3; i++)
  cudaMemcpyAsync(hostPtr + i*size,
    devicePtr + i*size, size,
    cudaMemcpyDeviceToHost, stream[i]);
```

Overlap of Data Transfer and Kernel Execution

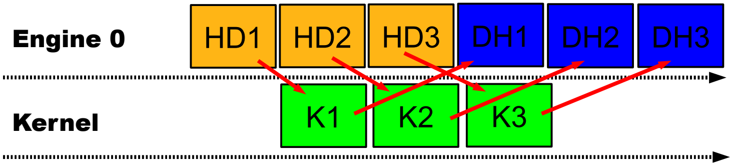


Overlap of Data Transfer and Kernel Execution

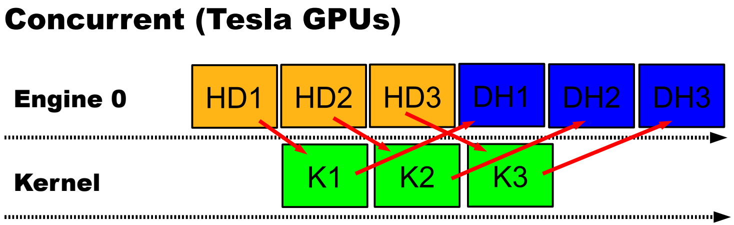


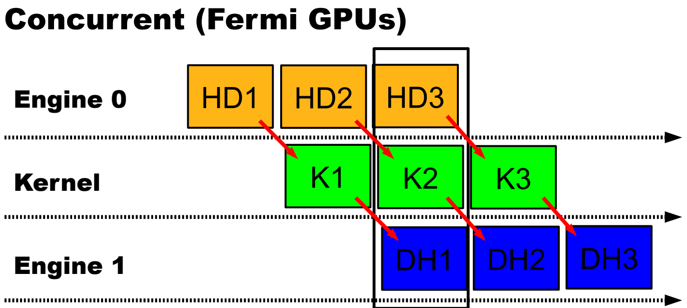
Overlap of Data Transfer and Kernel Execution

Concurrent (Tesla GPUs)



Overlap of Data Transfer and Kernel Execution





Memory Scope and Lifetime

Variable declaration		Memory	Scope	Lifetime
	`int var;`	register	thread	thread
	`int var[10];`	local	thread	thread
`__shared__`	`int var;`	shared	block	block
`__device__`	`int var;`	global	grid	application
`__constant__`	`int var;`	constant	grid	application

Function Qualifiers

- **__global__** – kernel function
  - executed on the device
  - callable from host
  - these funcions must have void return type
- **__device__** – device function
  - executed on the device
  - callable from the device only
- **__host__** – host function
  - executed on the host
  - it can be used with **__device__**

Function Qualifiers

Add One

```
__host__ __device__ float addone(const float v){
  return ( v + 1.0f );
}

__global__ void mykernel(float* A, int N){
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  if( i < N )
    A[i] = addone(A[i]);
}
```

References

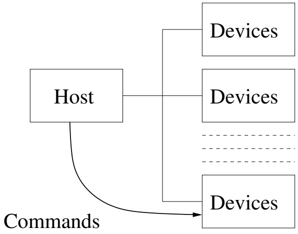
- Programming Massively Parallel Processors with CUDA, Stanford University. `http://code.google.com/p/stanford-cs193g-sp2010`.
- Programming Massively Parallel Processors: A Hands-on Approach, David Kirk and Wen-mei Hwu.
- CUDA C Progrmaming Guide, NIVIDIA.
- CUDA C Best Practices Guide, NVIDIA.

Outlines: General Purpose Graphical Processor Units (GPGPU)

OpenCL Architecture Model

Host-Devices model

- 1 host and several devices.
- Devices are connected to the host.
- Host issues commands to the devices.
- Data transport is done via memory copy.



Several devices support OpenCL

- NVIDIA for GPU and in the future for Tegra.
- AMD and Intel for CPUs and GPUs and MIC?
- IBM CELL processor.
- ARM GPUs (Mali) + CPUs

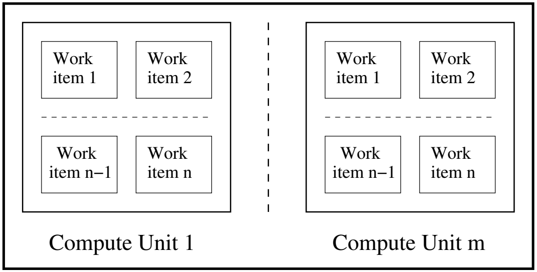
Context and Queues

- Contexts aggregate resources, programs and devices belonging to a common platform (ie NVIDIA, or ATI).
- Host and devices communicate via buffers defined in a context.
- Commands are sent to devices using command queues.
- Commands are called kernels.

Command queues

- Can be synchronous or asynchronous.
- Can be event driven.
- Several queues can point to the same device, allowing concurrent execution.

OpenCL Processing Model



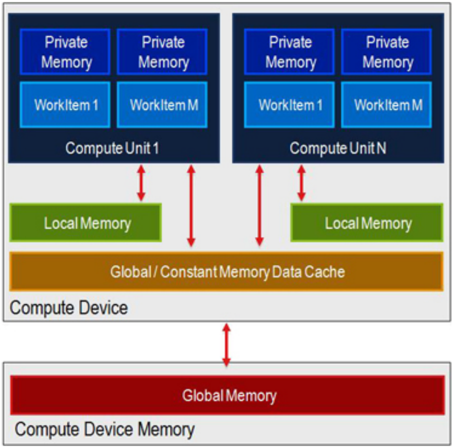
Compute Device

- Kernels are split into uni, two or three-dimensional ranges called work groups.
- Work groups are mapped to compute units.
- Individual item are processed by work items.

OpenCL Memory Model

4 different memory space defined on an OpenCL device :

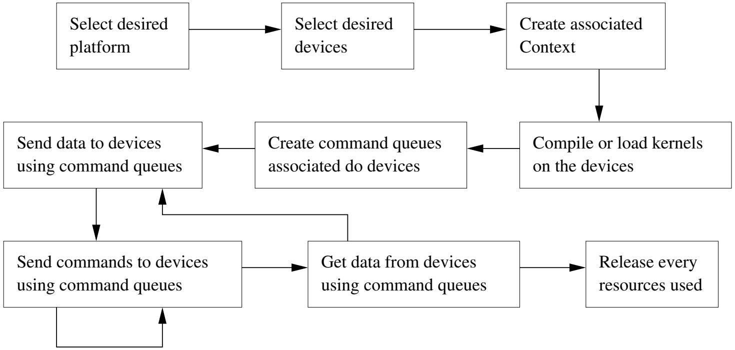
- Global memory : corresponds to the device RAM, input data are stored there.
- Constant memory : cached global memory.
- Local memory : high speed memory shared among work items of a compute unit.
- Private memory : registers of a work item.



Life and Death of OpenCL in a Program

The Host Side of OpenCL

General Workflow


Select desired platform
Select desired devices
Create associated Context
Compile or load kernels on the devices
Create command queues associated do devices
Send data to devices using command queues
Send commands to devices using command queues
Get data from devices using command queues
Release every resources used

Platform Selection

In a near future every platform will support OpenCL, but the user may not be interested in all of them: select an appropriate platform

Get Platforms

```
#include <CL/cl.h>
cl_uint num_platforms;
clGetPlatformIDs( NULL, NULL, &num_platforms);
cl_platform_id *platforms = malloc(sizeof(cl_platform_id) * num_platforms);
clGetPlatformIDs(num_platforms, platforms, NULL);
/* ... */
for(int i=0; i<num_platforms; i++){
  /* ... */
  clGetPlatformInfo(platforms[i], CL_PLATFORM_VENDOR, ... );
  /* ... */
}
```

Device Selection

Several device from the same vendor is also common: one device for the screen and one device for computations

Get Devices

```
#include <CL/cl.h>
cl_uint num_devices;
clGetDeviceIDs( platform, CL_DEVICE_TYPE_ALL, NULL, NULL, &num_devices);
cl_device_id *devices = malloc(sizeof(cl_device_id) * num_devices);
clGetDeviceIDs( platform, CL_DEVICE_TYPE_ALL, num_devices, devices, NULL);
/* ... */
for(int i=0; i<num_devices; i++){
  /* ... */
  clGetDeviceInfo(devices[i], CL_DEVICE_NAME , ... );
  /* ... */
}
```

Context Creation

Context gather devices from the same platform. Those devices will be able to share resources.

Create Context

```
cl_context_properties properties[] =
    { CL_CONTEXT_PLATFORM, (cl_context_properties)platform_id, 0 };
cl_device_id devices[] = {device_id_1, device_id_2};
cl_context context =
    clCreateContext(properties, 2, devices, NULL, NULL, NULL);
```

A shortcut exists, skipping device selection:

Create Context from Type

```
cl_context_properties properties[] =
    { CL_CONTEXT_PLATFORM, (cl_context_properties)platform_id, 0 };
cl_context context =
    clCreateContextFromType(properties, CL_DEVICE_TYPE_GPU , NULL, NULL, NULL);
```

Building Program from Source

Once the context is created, the program is to be built (or loaded from binary).

Building Program

```
/* strings is an array of string_count NULL terminated strings */
cl_program program =
   clCreateProgramWithSource(context, string_count, strings, NULL, NULL);
/* if device_list is NULL, program is built
 * for all available devices in the context */
clBuildProgram(program, num_devices, device_list, options, NULL, NULL);
cl_kernel kernel = clCreateKernel(program, "kernel_name", NULL);
```

Kernels are extracted from the built program using their name.

Creating Command Queues

A command queue is used to send commands to a device. They have to be associated with a device.

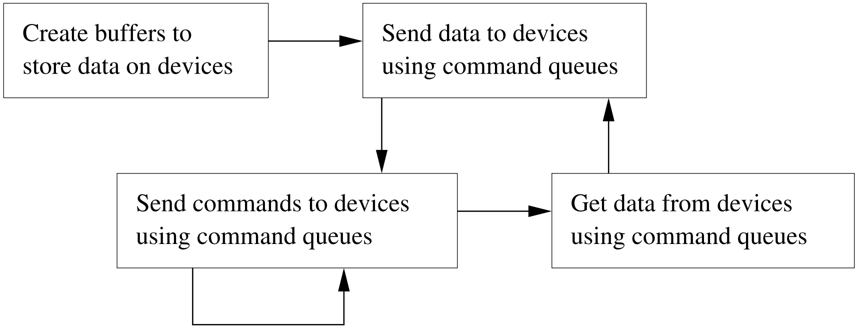
Creating Command Queues

```
cl_command_queue queue =
    clCreateCommandQueue(context, devices[chosen_device], 0, NULL);
```

Options can be specified instead of 0, CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE allows for out of order execution for instance.

Using OpenCL

Using OpenCL is (hopefully) easier than setting it up.


Create buffers to store data on devices
Send data to devices using command queues
Send commands to devices using command queues
Get data from devices using command queues

Buffer Creation

In OpenCL buffers creation and deletion are explicitly managed. As can be noted buffers are tied to a context and not a particular command queue. The implementation is free to transfer buffers from devices to host memory or to another device.

Creating Simple Buffers

```
cl_mem read_buffer =
    clCreateBuffer(context, CL_MEM_READ_ONLY, buffer_size, NULL, NULL);
cl_mem write_buffer =
    clCreateBuffer(context, CL_MEM_WRITE_ONLY, buffer_size, NULL, NULL);
```

Pinned Buffer Creation

Pinned buffer creation can offer premium performances. Here is a code sample that can be used on NVIDIA devices. The finals pointers obtained can be used to transfer data between the host and the device.

Creating Pinned Simple Buffers

```
cl_mem pinned_read_buffer =
   clCreateBuffer(context, CL_MEM_ALLOC_HOST_PTR | CL_MEM_READ_ONLY,
                  buffer_size, NULL, NULL);
cl_mem pinned_write_buffer =
   clCreateBuffer(context, CL_MEM_ALLOC_HOST_PTR | CL_MEM_WRITE_ONLY,
                  buffer_size, NULL, NULL);
unsigned char *data_in =
   clEnqueueMapBuffer(queue, pinned_read_buffer, CL_TRUE, CL_MAP_WRITE, 0,
                       buffer_size, 0, NULL, NULL, NULL);
unsigned char *data_out =
   clEnqueueMapBuffer(queue, pinned_write_buffer, CL_TRUE, CL_MAP_READ, 0,
                       buffer_size, 0, NULL, NULL, NULL);
```

Transferring Data

The implementation is free to move buffers in memory. But nonetheless, memory is often kept on the device associated to the command queue used to transfer the data.

Data Transfer

```
clEnqueueWriteBuffer(queue, read_buffer, CL_TRUE, 0,
                     buffer_size, data_in, 0, NULL, NULL);
/* Processing that reads read_buffer and writes write_buffer */
/* ... */
clEnqueueReadBuffer(queue, write_buffer, CL_TRUE, 0,
                    buffer_size, data_out, 0, NULL, NULL);
```

Performing Calculations

Once data is transfered, kernels are used to perform calculations.

Kernel Usage

```
/* Place kernel parameters in the kernel structure. */
clSetKernelArg(kernel, 0,sizeof(data_size), (void*)&data_size);
clSetKernelArg(kernel, 1,sizeof(read_buffer), (void*)&read_buffer);
clSetKernelArg(kernel, 2,sizeof(write_buffer), (void*)&write_buffer);
/* Enqueue a 1 dimensional kernel with a local size of 32 */
size_t localWorkSize[] = { 32 };
size_t globalWorkSize[] = { shrRoundUp(32, data_size) };
clEnqueueNDRangeKernel(queue, kernel, 1, NULL,
                       globalWorkSize, localWorkSize, 0, NULL, NULL);
```

Event Management

Almost all functions presented end with:

```
        ..., 0, NULL, NULL);
```

These 3 arguments are used for event management, and thus asynchronous queue handling. Functions can wait for a number of events, and can generate 1 event.

```
event_t event_list[] = {event1, event2};
event_t event;
clEnqueueReadBuffer(queue, write_buffer, CL_FALSE, 0,
                    buffer_size, data_out, 2, event_list, &event);
```

Previous buffer read waits for 2 events and generate a third that will happen when the read is completed.

Release Resources

OpenCL uses reference counts to manage memory. In order to exit cleanly from an OpenCL program all allocated resources have to be freed:

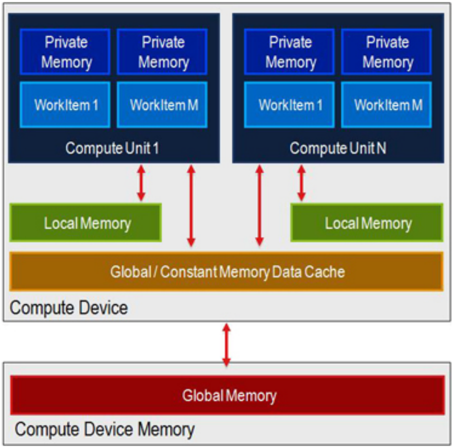
- buffers (`clReleaseMemObject`)
- events (`clReleaseEvent`)
- kernel (`clReleasekernel`)
- programs (`clReleaseProgram`)
- queues (`clReleaseCommandQueue`)
- context (`clReleaseContext`)
- etc...

Writing Kernels

Recall : OpenCL Memory Model

4 different memory space defined on an OpenCL device :

- Global memory : corresponds to the device RAM, input data are stored there.
- Constant memory : cached global memory.
- Local memory : high speed memory shared among work items of a compute unit.
- Private memory : registers of a work item.

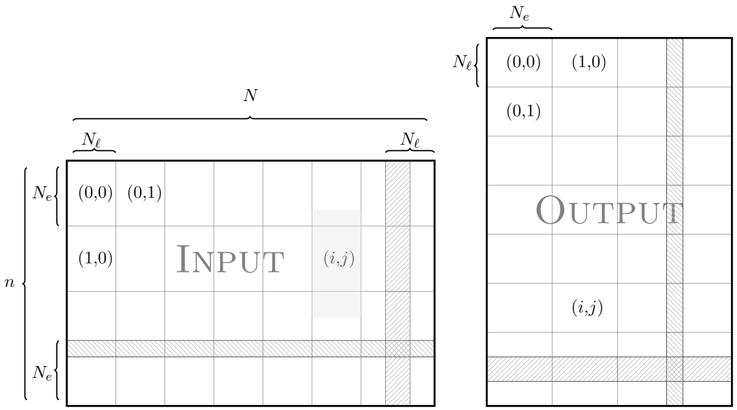


OpenCL Language : a Subset of C

Kernels are written using a C-like language

- Recursion is prohibited
- Helper functions are defined
  - Barriers
  - Work item indexes
  - Atomic operations
  - Vector operations
- New keywords :
  - `__kernel`
  - `__global`, `__local`, `__constant`, `__private`
  - `__read_only`, `__write_only`, `__read_write`

Example : Unidimensional Convolutions



One unidimensional convolution with transposition, simple but not too much. Real world code used in BigDFT an electronic structure calculation program.

Kernel Declaration

Kernel Declaration

```
/* Activate double precision support */
#pragma OPENCL EXTENSION cl_khr_fp64: enable
#define FILT_W 16
#define WG_S 16
__kernel void
__attribute__((reqd_work_group_size(WG_S,WG_S,1)))
magicfilter1dKernel_d(uint n, uint ndat,
                            __global const double *psi,
                            __global double *out){
//padded local buffer size : 33*16
__local double tmp[WG_S*(WG_S+FILT_W+1)];
```

- Works on double precision floats
- Kernel expects work group size of 16 x 16
- `n` and `ndat` are in `__local` memory
- `tmp1` is a storage buffer in local memory, shared among work items

Work with Indexes

Get Indexes and Load Data

```
//get our position in the local work group
const size_t ig = get_global_id(0);
const size_t jg = get_global_id(1);
//get our position in the result matrix
const size_t i = get_local_id(0);
const size_t j = get_local_id(1);
//transpose indexes in the work group in order to read transposed data
ptrdiff_t igt = ig − i + j − FILT_W/2;
ptrdiff_t jgt = jg − j + i;
//if we are on the outside, select a border element to load, wrapping around
//we will be loading 2 elements each
if ( igt < 0 )
  tmp[i * (WG_S+FILT_W+1) + j] = psi[jgt + ( n + igt ) * ndat];
else
  tmp[i * (WG_S+FILT_W+1) + j] = psi[jgt + igt * ndat];
igt += FILT_W;
if ( igt >= n )
  tmp[i * (WG_S+FILT_W+1) + j + FILT_W] = psi[jgt + ( igt − n ) * ndat];
else\n\
  tmp[i * (WG_S+FILT_W+1) + j + FILT_W] = psi[jgt +  igt * ndat];
```

Compute Convolution and Write Output

Performing Computations

```
//initialize result
double tt = 0.0;
//rest position in the buffer to first element involved in the convolution
tmp += j2*(WG_S+FILT_W+1) + i2;
//wait for buffer to be full
barrier(CLK_LOCAL_MEM_FENCE);

//apply filter
tt += *tmp++ * FILT0;
tt += *tmp++ * FILT1;
/* ... */
tt += *tmp++ * FILT15;
//store the result
out[(jg*n+ig)]=tt;
};
```

Test System Setup

GPU 2:

- Tesla C2070 (Fermi)
- 6 GB of RAM
- Driver version: 260.14

GPU 2:

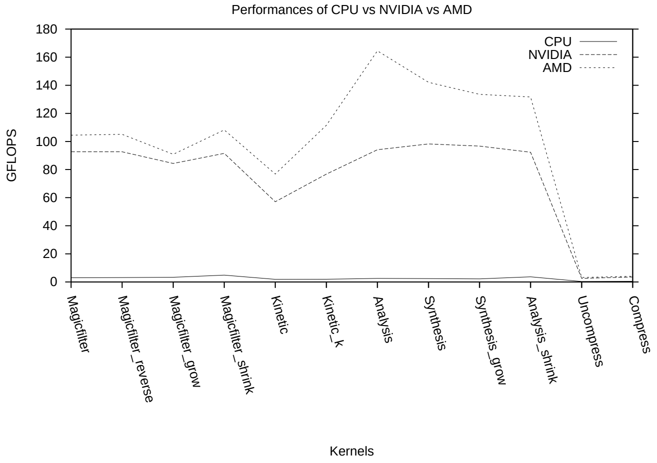
- Radeon HD6970
- 2 GB of RAM
- Driver version: 11.6

Test System Setup

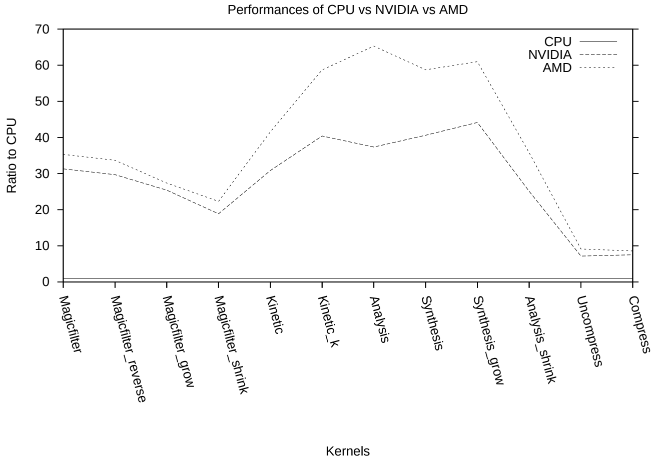
Host :

- Lenovo D20
- 1 Xeon 5550 @ 2.83 GHz (4 Nehalem cores)
- 8 GB of RAM
- Linux 2.6.38-11 x86_64
- icc 11.1

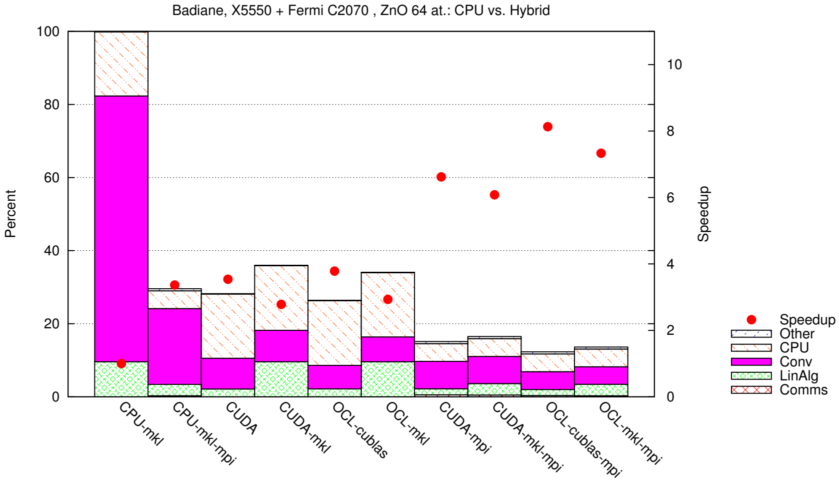
Comparison CPU,Fermi,HD6970



Comparison CPU,Fermi,HD6970



Comparison CUDA, OpenCL, CPU



Hybrid ATI + NVIDIA

Tesla C2070 + Radeon HD 6970

MPI+NVIDIA/AMD	Execution Time (s)	Speedup
1	6020	1
4	1660	3.6
1 + NVIDIA	300	20
4 + NVIDIA	160	38
1 + AMD	347	17
4 + AMD	197	30
(4 + NV) + (4 + AMD)	109	55

Table: Performance results for different configuration of BigDFT, using MPI + GPUs

Interesting Additions from OpenCL 1.1 and 1.2

clSetEventCallback

Version : OpenCL 1.1 and 1.2
The OpenCL implementation calls a C function asynchronously when the status of an event changes.

Status Changes:

- CL_COMPLETE (1.1)
- CL_SUBMITTED (1.2)
- CL_RUNNING (1.2)

clCreateSubDevices

Version : OpenCL 1.2 (Extension 1.1)
Split a device in several sub devices either arbitrarily or based on memory hierarchy.

Hierarchical Split:

- CL_DEVICE_AFFINITY_DOMAIN_NUMA
- CL_DEVICE_AFFINITY_DOMAIN_L4_CACHE
- ...
- CL_DEVICE_AFFINITY_DOMAIN_L1_CACHE
- CL_DEVICE_AFFINITY_DOMAIN_NEXT_PARTITIONABLE

clEnqueueMigrateMemObjects

Version : OpenCL 1.2

- Migrates an object to the memory associated to a device via its command queue.
- Can be event driven.

Special Uses:

- CL_MIGRATE_MEM_OBJECT_HOST
- CL_MIGRATE_MEM_OBJECT_CONTENT_UNDEFINED

Conclusions

OpenCL

- OpenCL proved easy to use.
- Performance is on-par with previous CUDA implementation.
- Kernels have been shown to run on other architectures: ATI and CPU.

Perspectives

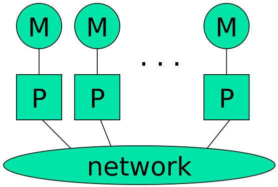
- Some OpenCL implementations are still recent and buggy.
- Best way to do multi-GPU, GPU+OpenCL CPU?
- Optimizing kernels for multiple devices?
- Automated kernel generation.

Part III

Message Passing Interface (MPI)

Outlines: Message Passing Interface (MPI)

Message Passing



- Each processor runs a process
- Processes communicate by exchanging messages
- They cannot share memory in the sense that they cannot address the same memory cells

- The above is a programming model and things may look different in the actual implementation (e.g., MPI over Shared Memory)
- Message Passing is popular because it is general:
  - Pretty much any distributed system works by exchanging messages, at some level
  - Distributed- or shared-memory multiprocessors, networks of workstations, uniprocessors
- It is not popular because it is easy (it’s not)

Courtesy of Henri Casanova

Code Parallelization

- Shared-memory programming
  - Parallelizing existing code can be very easy
    - OpenMP: just add a few pragmas
    - Pthreads: wrap work in do_work functions
  - Understanding parallel code is easy
  - Incremental parallelization is natural
- Distributed-memory programming
  - parallelizing existing code can be very difficult
    - No shared memory makes it impossible to “just” reference variables
    - Explicit message exchanges can get really tricky
  - Understanding parallel code is difficult
    - Data structured are split all over different memories
  - Incremental parallelization can be challenging

Courtesy of Henri Casanova

Programming Message Passing

- Shared-memory programming is simple conceptually (sort of)
- Shared-memory machines are expensive when one wants a lot of processors
- It’s cheaper (and more scalable) to build distributed memory machines
  - Distributed memory supercomputers (IBM SP series)
  - Commodity clusters
- But then how do we program them?
- At a basic level, let the user deal with explicit messages
  - difficult
  - but provides the most flexibility

Courtesy of Henri Casanova

Message Passing

- Isn't exchanging messages completely known and understood?
  - That's the basis of the IP idea
  - Networked computers running programs that communicate are very old and common
    - DNS, e-mail, Web, ...
- The answer is that, yes it is, we have "Sockets"
  - Software abstraction of a communication between two Internet hosts
  - Provides and API for programmers so that they do not need to know anything (or almost anything) about TCP/IP and write code with programs that communicate over the internet

Socket Library in UNIX

- Introduced by BSD in 1983
  - The “Berkeley Socket API”
  - For TCP and UDP on top of IP
- The API is known to not be very intuitive for first-time programmers
- What one typically does is write a set of “wrappers” that hide the complexity of the API behind simple function
- Fundamental concepts
  - Server side
    - Create a socket
    - Bind it to a port numbers
    - Listen on it
    - Accept a connection
    - Read/Write data
  - Client side
    - Create a socket
    - Connect it to a (remote) host/port
    - Write/Read data

Courtesy of Henri Casanova

Socket: server.c

```
int main(int argc, char *argv[])
{
   int sockfd, newsockfd, portno, clilen;
   char buffer[256];
   struct sockaddr_in serv_addr, cli_addr;
   int n;

   sockfd = socket(AF_INET, SOCK_STREAM, 0);
   bzero((char *) &serv_addr, sizeof(serv_addr));
   portno = 666;
   serv_addr.sin_family = AF_INET;
   serv_addr.sin_addr.s_addr = INADDR_ANY;
   serv_addr.sin_port = htons(portno);
   bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr))
   listen(sockfd,5);
   clilen = sizeof(cli_addr);
   newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
   bzero(buffer,256);
   n = read(newsockfd,buffer,255);
   printf("Here is the message: %s\n",buffer);
   n = write(newsockfd,"I got your message",18);
   return 0;
}
```

Courtesy of Henri Casanova

Socket: client.c

```
int main(int argc, char *argv[])
{
    int sockfd, portno, n;
    struct sockaddr_in serv_addr;
    struct hostent *server;

    char buffer[256];
    portno = 666;
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    server = gethostbyname(“server_host.univ.edu);
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    bcopy((char *)server->h_addr,(char *)&serv_addr.sin_addr.s_addr,server->h_length);
    serv_addr.sin_port = htons(portno);
    connect(sockfd,&serv_addr,sizeof(serv_addr));
    printf("Please enter the message: ");
    bzero(buffer,256);
    fgets(buffer,255,stdin);
    write(sockfd,buffer,strlen(buffer));
    bzero(buffer,256);
    read(sockfd,buffer,255);
    printf("%s\n",buffer);
    return 0;
}
```

Courtesy of Henri Casanova

Socket in C/UNIX

- The API is really not very simple
  - And note that the previous code does not have any error checking
  - Network programming is an area in which you should check ALL possible error code
  - In the end, writing a server that receives a message and sends back another one, with the corresponding client, can require 100+ lines of C if one wants to have robust code
  - This is OK for UNIX programmers, but not for everyone
  - However, nowadays, most applications written require some sort of Internet communication

Sockets in Java

- Socket class in java.net
  - Makes things a bit simpler
  - Still the same general idea
  - With some Java stuff

- Server

```
try { serverSocket = new ServerSocket(666);
} catch (IOException e) { <something> }
Socket clientSocket = null;
try { clientSocket = serverSocket.accept();
} catch (IOException e) { <something> }
PrintWriter out = new
  PrintWriter(                      clientSocket.getOutputStream()
  , true);
BufferedReader in = new BufferedReader(                       new
  InputStreamReader(clientSocket.getInputStream()));
// read from "in", write to "out"
```

Sockets in Java

- Java client

```
try {socket = new Socket("server.univ.edu", 666);}
  catch { <something> }
out = new PrintWriter(socket.getOutputStream(), true);
in = new BufferedReader(new InputStreamReader(
                                socket.getInputStream()));
 // write to out, read from in
```

- Much simpler than the C
- Note that if one writes a client-server program one typically creates a Thread after an accept, so that requests can be handled concurrently

Courtesy of Henri Casanova

Using Sockets for parallel programming?

- One could thing of writing all parallel code on a cluster using sockets
  - n nodes in the cluster
  - Each node creates n-1 sockets on n-1 ports
  - All nodes can communicate
- Problems with this approach
  - Complex code
  - Only point-to-point communication
  - No notion of types messages
  - But
    - All this complexity could be “wrapped” under a higher-level API
    - And in fact, we’ll see that’s the basic idea
  - Does not take advantage of fast networking within a cluster/ MPP
    - Sockets have “Internet stuff” in them that’s not necessary
    - TPC/IP may not even be the right protocol!

Courtesy of Henri Casanova

Message Passing for Parallel Programs

- Although “systems” people are happy with sockets, people writing parallel applications need something better
  - easier to program to
  - able to exploit the hardware better within a single machine
- This “something better” right now is MPI
  - We will learn how to write MPI programs
- Let’s look at the history of message passing for parallel computing

Outlines: Message Passing Interface (MPI)

The MPI Standard

- MPI Forum setup as early as 1992 to come up with a de facto standard with the following goals:
  - source-code portability
  - allow for efficient implementation (e.g., by vendors)
  - support for heterogeneous platforms
- MPI is not
  - a language
  - an implementation (although it provides hints for implementers)
- June 1995: MPI v1.1 (we're now at MPI v1.2)
  - http://www-unix.mcs.anl.gov/mpi/
  - C and FORTRAN bindings
  - We will use MPI v1.1 from C in the class
- Implementations:
  - well-adopted by vendors
  - free implementations for clusters: MPICH, LAM, CHIMP/MPI
  - research in fault-tolerance: MPICH-V, FT-MPI, MPIFT, etc.

Courtesy of Henri Casanova

SPMD Programs

- It is rare for a programmer to write a different program for each process of a parallel application
- In most cases, people write Single Program Multiple Data (SPMD) programs
  - the same program runs on all participating processors
  - processes can be identified by some *rank*
  - This allows each process to know which piece of the problem to work on
  - This allows the programmer to specify that some process does something, while all the others do something else (common in master-worker computations)

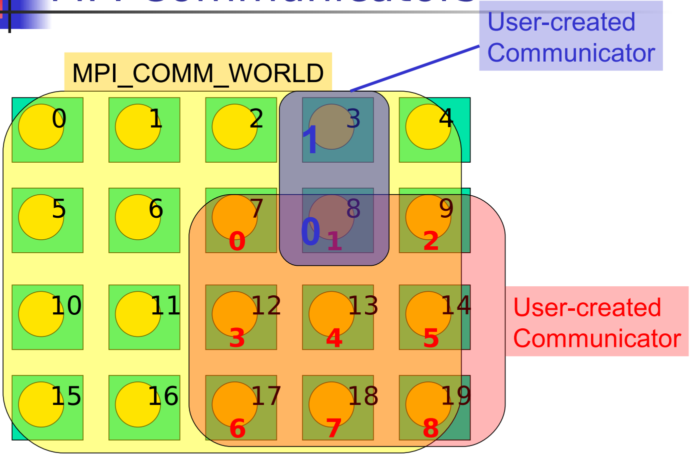
```
main(int argc, char **argv) {
   if (my_rank == 0) { /* master */
     ... load input and dispatch ...
   } else { /* workers */
     ... wait for data and compute ...
   }
```

Courtesy of Henri Casanova

MPI Concepts

- Fixed number of processors
  - When launching the application one must specify the number of processors to use, which remains unchanged throughout execution
- Communicator
  - Abstraction for a group of processes that can communicate
  - A process can belong to multiple communicators
  - Makes is easy to partition/organize the application in multiple layers of communicating processes
  - Default and global communicator: ***`MPI_COMM_WORLD`***
- Process Rank
  - The index of a process within a communicator
  - Typically user maps his/her own virtual topology on top of just linear ranks
    - ring, grid, etc.

MPI Communicators



A First MPI Program

```
#include <unistd.h>
#include <mpi.h>
int main(int argc, char **argv) {
  int my_rank, n;
  char hostname[128];
  MPI_init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
  MPI_Comm_size(MPI_COMM_WORLD,&n);
  gethostname(hostname,128);
  if (my_rank == 0) { /* master */
    printf("I am the master: %s\n",hostname);
  } else { /* worker */
    printf("I am a worker: %s (rank=%d/%d)\n",
           hostname,my_rank,n-1);
  }
  MPI_Finalize();
  exit(0);
}
```

Has to be called first, and once

Has to be called last, and once

Courtesy of Henri Casanova

Compiling/Running it

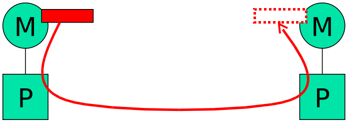
- Compile with mpicc
- Run with ***mpirun***
  ```
  % mpirun -np 4 my_program <args>
  ```
  - requests 4 processors for running ***my_program*** with command-line arguments
  - see the ***mpirun*** man page for more information
  - in particular the ***-machinefile*** option that is used to run on a network of workstations
- Some systems just run all programs as MPI programs and no explicit call to ***mpirun*** is actually needed
- Previous example program:

```
% mpirun -np 3 -machinefile hosts my_program
  I am the master: somehost1
  I am a worker: somehost2 (rank=2/2)
  I am a worker: somehost3 (rank=1/2)
```

(stdout/stderr redirected to the process calling mpirun)

Outlines: Message Passing Interface (MPI)

Point-to-Point Communication



- Data to be communicated is described by three things:
  - address
  - data type of the message
  - length of the message
- Involved processes are described by two things
  - communicator
  - rank
- Message is identified by a “tag” (integer) that can be chosen by the user

Courtesy of Henri Casanova

Point-to-Point Communication

- Two modes of communication:
  - Synchronous: Communication does not complete until the message has been received
  - Asynchronous: Completes as soon as the message is “on its way”, and hopefully it gets to destination
- MPI provides four versions
  - synchronous, buffered, standard, ready

Synchronous/Buffered sending in MPI

- Synchronous with MPI_Ssend
  - The send completes only once the receive has succeeded
    - copy data to the network, wait for an ack
    - The sender has to wait for a receive to be posted
    - No buffering of data
- Buffered with MPI_Bsend
  - The send completes once the message has been buffered internally by MPI
    - Buffering incurs an extra memory copy
    - Doe not require a matching receive to be posted
    - May cause buffer overflow if many bsends and no matching receives have been posted yet

Standard/Ready Send

- Standard with MPI_Send
  - Up to MPI to decide whether to do synchronous or buffered, for performance reasons
  - The rationale is that a correct MPI program should not rely on buffering to ensure correct semantics
- Ready with MPI_Rsend
  - May be started *only* if the matching receive has been posted
  - Can be done efficiently on some systems as no hand-shaking is required

MPI_RECV

- There is only one MPI_Recv, which returns when the data has been received.
  - only specifies the **MAX** number of elements to receive
- Why all this junk?
  - Performance, performance, performance
  - MPI was designed with constructors in mind, who would endlessly tune code to extract the best out of the platform (LINPACK benchmark).
  - Playing with the different versions of MPI_?send can improve performance without modifying program semantics
  - Playing with the different versions of MPI_?send can modify program semantics
  - Typically parallel codes do not face very complex distributed system problems and it's often more about performance than correctness.
  - You'll want to play with these to tune the performance of your code in your assignments

Example: Sending and Receiving

```
#include <unistd.h>
#include <mpi.h>
int main(int argc, char **argv) {
  int i, my_rank, nprocs, x[4];
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
  if (my_rank == 0) { /* master */
    x[0]=42; x[1]=43; x[2]=44; x[3]=45;
    MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
    for (i=1;i<nprocs;i++)
      MPI_Send(x,4,MPI_INT,i,0,MPI_COMM_WORLD);
  } else { /* worker */
    MPI_Status status;
    MPI_Recv(x,4,MPI_INT,0,0,MPI_COMM_WORLD,&status);
  }
  MPI_Finalize();
  exit(0);
}
```

destination and source

user-defined tag

Max number of elements to receive

Can be examined via calls like MPI_Get_count(), etc.

Example: Deadlock

```
...
MPI_Ssend()
MPI_Recv()
...
```

Deadlock

```
...
MPI_Ssend()
MPI_Recv()
...
```

```
...
MPI_Buffer_attach()
MPI_Bsend()
MPI_Recv()
...
```

No Deadlock

```
...
MPI_Buffer_attach()
MPI_Bsend()
MPI_Recv()
...
```

```
...
MPI_Buffer_attach()
MPI_Bsend()
MPI_Recv()
...
```

No Deadlock

```
...
MPI_Ssend()
MPI_Recv()
...
```

Courtesy of Henri Casanova

What about MPI_Send?

- MPI_Send is either synchronous or buffered....
- With , running “same” version of MPICH

Deadlock

```
...
MPI_Send()
MPI_Recv()
...
```

Data size > 127999 bytes

Data size < 128000 bytes

```
...
MPI_Send()
MPI_Recv()
...
```

No Deadlock

- Rationale: a correct MPI program should not rely on buffering for semantics, just for performance.
- So how do we do this then? ...

Courtesy of Henri Casanova

Non-blocking communications

- So far we've seen blocking communication:
  - The call returns whenever its operation is complete (MPI_SSEND returns once the message has been received, MPI_BSEND returns once the message has been buffered, etc..)
- MPI provides non-blocking communication: the call returns immediately and there is another call that can be used to check on completion.
- Rationale: Non-blocking calls let the sender/receiver do something useful while waiting for completion of the operation (without playing with threads, etc.).

Non-blocking Communication

- MPI_Issend, MPI_Ibsend, MPI_Isend, MPI_Irsend, MPI_Irecv

```
MPI_Request request;
MPI_Isend(&x,1,MPI_INT,dest,tag,communicator,&request);
MPI_Irecv(&x,1,MPI_INT,src,tag,communicator,&request);
```

- Functions to check on completion: MPI_Wait, MPI_Test, MPI_Waitany, MPI_Testany, MPI_Waitall, MPI_Testall, MPI_Waitsome, MPI_Testsome.

```
MPI_Status status;
MPI_Wait(&request, &status) /* block */
MPI_Test(&request, &status) /* doesn't block */
```

Courtesy of Henri Casanova

Example: Non-blocking comm

```
#include <unistd.h>
#include <mpi.h>
int main(int argc, char **argv) {
  int i, my_rank, x, y;
  MPI_Status status;
  MPI_Request request;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
  if (my_rank == 0) { /* P0 */
    x=42;
    MPI_Isend(&x,1,MPI_INT,1,0,MPI_COMM_WORLD,&request);
    MPI_Recv(&y,1,MPI_INT,1,0,MPI_COMM_WORLD,&status);
    MPI_Wait(&request,&status);
 } else if (my_rank == 1) { /* P1 */
    y=41;
    MPI_Isend(&y,1,MPI_INT,0,0,MPI_COMM_WORLD,&request);
    MPI_Recv(&x,1,MPI_INT,0,0,MPI_COMM_WORLD,&status);
    MPI_Wait(&request,&status);
  }
  MPI_Finalize(); exit(0);
}
```

No Deadlock

Courtesy of Henri Casanova

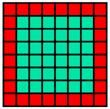
Use of non-blocking comms

- In the previous example, why not just swap one pair of send and receive?
- Example:
  - A logical linear array of N processors, needing to exchange data with their neighbor at each iteration of an application
  - One would need to orchestrate the communications:
    - all odd-numbered processors send first
    - all even-numbered processors receive first
  - Sort of cumbersome and can lead to complicated patterns for more complex examples
  - In this case: just use MPI_Isend and write much simpler code
- Furthermore, using MPI_Isend makes it possible to overlap useful work with communication delays:

```
MPI_Isend()
<useful work>
MPI_Wait()
```

Iterative Application Example

```
for (iterations)
  update all cells
  send boundary values
  receive boundary values
```



- Would deadlock with MPI_Ssend, and maybe deadlock with MPI_Send, so must be implemented with MPI_Isend
- Better version that uses non-blocking communication to achieve communication/computation overlap (aka latency hiding):

```
for (iterations)
  initiate sending of boundary values to neighbours;
  initiate receipt of boundary values from neighbours;
  update non-boundary cells;
  wait for completion of sending of boundary values;

  wait for completion of receipt of boundary values;
  update boundary cells;
```

- Saves cost of boundary value communication if hardware/software can overlap comm and comp

Courtesy of Henri Casanova

Non-blocking communications

- Almost always better to use non-blocking
  - communication can be carried out during blocking system calls
  - communication and communication can overlap
  - less likely to have annoying deadlocks
  - synchronous mode is better than implementing acks by hand though
- However, everything else being equal, non-blocking is slower due to extra data structure bookkeeping
  - The solution is just to benchmark
- When you do your programming assignments, you will play around with different communication types

More information

- There are many more functions that allow fine control of point-to-point communication
- Message ordering is guaranteed
- Detailed API descriptions at the MPI site at ANL:
  - Google “MPI”. First link.
  - Note that you should check error codes, etc.
- Everything you want to know about deadlocks in MPI communication

  http://andrew.ait.iastate.edu/HPC/Papers/mpicheck2/mpicheck2.htm

Outlines: Message Passing Interface (MPI)

Collective Communication

- Operations that allow more than 2 processes to communicate simultaneously
  - barrier
  - broadcast
  - reduce
- All these can be built using point-to-point communications, but typical MPI implementations have optimized them, and it’s a good idea to use them
- In all of these, all processes place the same call (in good SPMD fashion), although depending on the process, some arguments may not be used

Barrier

- Synchronization of the calling processes
  - the call blocks until all of the processes have placed the call
- No data is exchanged
- Similar to an OpenMP barrier

```
...
MPI_Barrier(MPI_COMM_WORLD)
...
```

Broadcast

- One-to-many communication
- Note that multicast can be implemented via the use of communicators (i.e., to create processor groups)

```
...
MPI_Bcast(x, 4, MPI_INT, 0,
  MPI_COMM_WORLD)
...
```

Rank of the root

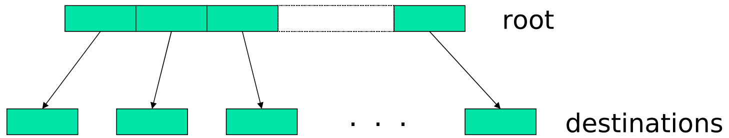
Broadcast example

- Let’s say the master must send the user input to all workers

```
int main(int argc,char **argv) {
   int my_rank;
   int input;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
   if (argc != 2) exit(1);
    if (sscanf(argv[1],"%d",&input) != 1) exit(1);
   MPI_Bcast(&input,1,MPI_INT,0,MPI_COMM_WORLD);
   ...
}
```

Scatter

- One-to-many communication
- Not sending the same message to all



```
...
MPI_Scatter(x, 100, MPI_INT, y, 100, MPI_INT, 0,
   MPI_COMM_WORLD)
...
```

Send buffer

Data to send to each

Receive buffer

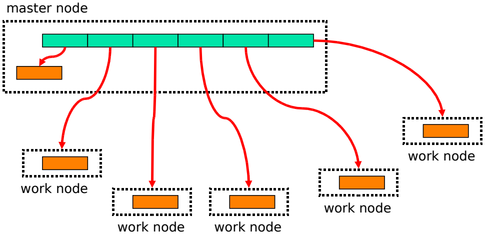
Data to receive

Rank of the root

Courtesy of Henri Casanova

This is actually a bit tricky

- The root sends data to itself!



- Arguments #1, #2, and #3 are only meaningful at the root

Courtesy of Henri Casanova

Scatter Example

- Partitioning an array of input among workers

```
int main(int argc,char **argv) {
   int *a;
   double *revbuffer;
   ...
   MPI_Comm_size(MPI_COMM_WORLD,&n);
   <allocate array recvbuffer of size N/n>

   if (my_rank == 0) { /* master */
       <allocate array a of size N>
   }
   MPI_Scatter(a, N/n, MPI_INT,
                    recvbuffer, N/n, MPI_INT,
                    0, MPI_COMM_WORLD);
   ...
}
```

Courtesy of Henri Casanova

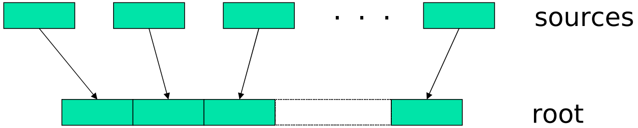
Scatter Example

- Without redundant sending at the root

```
int main(int argc,char **argv) {
   int *a;
   double *revbuffer;
   ...
   MPI_Comm_size(MPI_COMM_WORLD,&n);
   if (my_rank == 0) { /* master */
        <allocate array a of size N>
        <allocate array recvbuffer of size N/n>
        MPI_Scatter(a, N/n, MPI_INT,
                    MPI_IN_PLACE, N/n, MPI_INT,
                    0, MPI_COMM_WORLD);
   } else { /* worker */
        <allocate array recvbuffer of size N/n>
        MPI_Scatter(NULL, 0, MPI_INT,
                    recvbuffer, N/n, MPI_INT,
                    0, MPI_COMM_WORLD);
   }
   ...
}
```

Gather

- Many-to-one communication
- Not sending the same message to the root



```
...
MPI_Gather(x, 100, MPI_INT, y, 100, MPI_INT, 0, MPI_COMM_WORLD)
...
```

Send buffer

Data to send from each

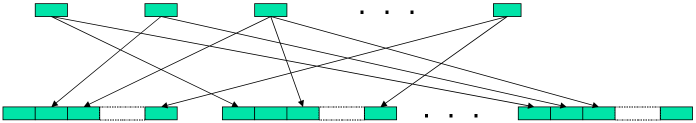
Receive buffer

Data to receive

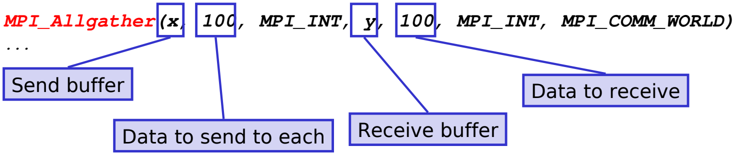
Rank of the root

Gather-to-all

- Many-to-many communication
- Each process sends the same message to all
- Different Processes send different messages



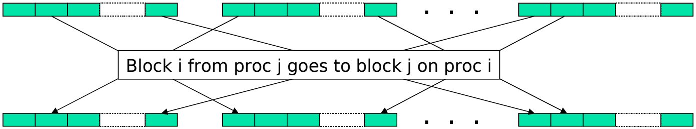
```
...
MPI_Allgather(x, 100, MPI_INT, y, 100, MPI_INT, MPI_COMM_WORLD)
...
```

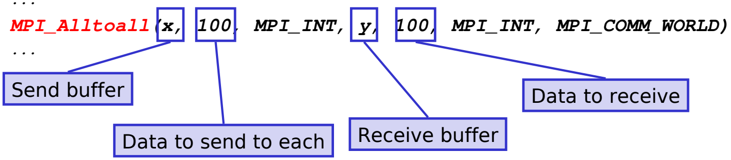


Courtesy of Henri Casanova

All-to-all

- Many-to-many communication
- Each process sends a different message to each other process





```
...
MPI_Alltoall(x, 100, MPI_INT, y, 100, MPI_INT, MPI_COMM_WORLD)
...
```

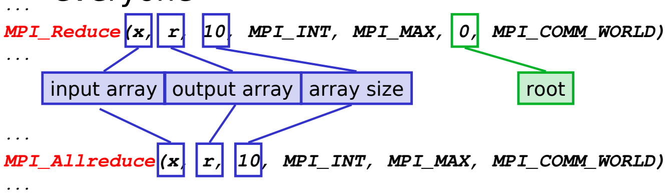
Courtesy of Henri Casanova

Reduction Operations

- Used to compute a result from data that is distributed among processors
  - often what a user wants to do anyway
    - e.g., compute the sum of a distributed array
  - so why not provide the functionality as a single API call rather than having people keep re-implementing the same things
- Predefined operations:
  - MPI_MAX, MPI_MIN, MPI_SUM, etc.
- Possibility to have user-defined operations

MPI_Reduce, MPI_Allreduce

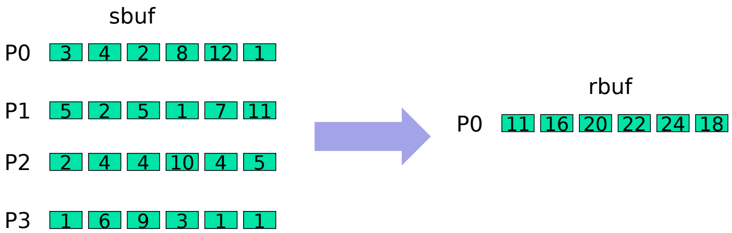
- MPI_Reduce: result is sent out to the root
  - the operation is applied element-wise for each element of the input arrays on each processor
  - An output array is returned
- MPI_Allreduce: result is sent out to everyone



```
...
MPI_Reduce(x, r, 10, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD)
...

...
MPI_Allreduce(x, r, 10, MPI_INT, MPI_MAX, MPI_COMM_WORLD)
...
```

MPI_Reduce example

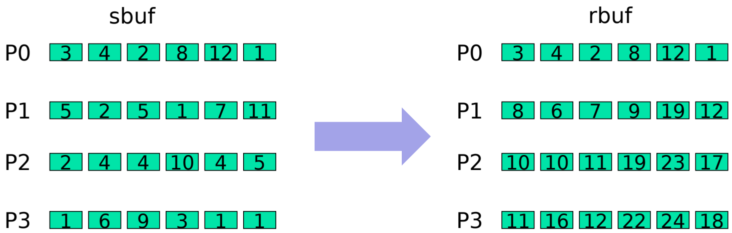
***MPI_Reduce**(sbuf,rbuf,6,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD)*



Courtesy of Henri Casanova

MPI_Scan: Prefix reduction

- Process i receives data reduced on process 0 to i.



***MPI_Scan*(sbuf,rbuf,6,MPI_INT,MPI_SUM,MPI_COMM_WORLD)**

Courtesy of Henri Casanova

And more...

- Most broadcast operations come with a version that allows for a stride (so that blocks do not need to be contiguous)
  - MPI_Gatherv(), MPI_Scatterv(), MPI_Allgatherv(), MPI_Alltoallv()
- MPI_Reduce_scatter(): functionality equivalent to a reduce followed by a scatter
- All the above have been created as they are common in scientific applications and save code
- All details on the MPI Webpage

Outlines: Message Passing Interface (MPI)

MPI-2

- MPI-2 provides for:
  - Remote Memory
    - put and get primitives, weak synchronization
    - makes it possible to take advantage of fast hardware (e.g., shared memory)
    - gives a shared memory twist to MPI
  - Parallel I/O
    - we'll talk about it later in the class
  - Dynamic Processes
    - create processes during application execution to grow the pool of resources
    - as opposed to "everybody is in MPI_COMM_WORLD at startup and that's the end of it"
    - as opposed to "if a process fails everything collapses"
    - a MPI_Comm_spawn() call has been added (akin to PVM)
  - Thread Support
    - multi-threaded MPI processes that play nicely with MPI
  - Extended Collective Communications
  - Inter-language operation, C++ bindings
  - Socket-style communication: open_port, accept, connect (client-server)
- MPI-2 implementations are now available

Outlines: Message Passing Interface (MPI)

Summary

Lots of different parallel languages in HPC

- PThread, OpenMP, Cuda, OpenCL, MPI and lots of others
- different targets, different properties (shared memory, SIMD, etc.)

Future

- Mixing these models?
- Why still new parallel language nowadays?
- How can these parallel environment be improved?