EECS 570
Accelerating 3D Ultrasound Beamforming on the Xeon Phi
Programming Assignment #1
Portable Medical Imaging Devices

• Medical imaging moving towards portability
  – MEDICS (X-Ray CT) [Dasika ‘10]
  – Handheld 2D Ultrasound [Fuller ‘09]

• Not just a matter of convenience
  – Improved patient health [Gunnarsson ‘00, Weinreb ‘08]
  – Access in developing countries

• Why ultrasound?
  – Low transmit power [Nelson ‘10]
  – No dangers or side-effects
Ultrasound: Transmit and Receive
Ultrasound: Transmit and Receive
Ultrasound: Transmit and Receive
Ultrasound: Transmit and Receive
Ultrasound: Transmit and Receive
Ultrasound: Transmit and Receive
Ultrasound: Transmit and Receive
Ultrasound: Transmit and Receive
Ultrasound: Transmit and Receive
Ultrasound: Transmit and Receive
Ultrasound: Transmit and Receive
Ultrasound: Transmit and Receive
Ultrasound: Transmit and Receive
Ultrasound: Transmit and Receive

Each transducer stores array of raw receive data
Ultrasound: Image Reconstruction

*Image reconstructed from data based on round trip delay*
Ultrasound: Image Reconstruction

Images from each transducer combined to produce full frame
Delay Index Calculation

- Iterate through all image points for each transducer and calculate delay index $\tau_P$

$$\tau_P = \frac{f_s}{c} \left( R_p + \sqrt{R_p^2 + X_i^2 - 2R_pX_i\sin\theta} \right)$$

- Often done with lookup tables (LUTs) instead
- 50 GB LUT required for target 3D system
Intel Xeon Phi Coprocessors and the MIC Architecture

- PCIe end-point device
- High Power efficiency
- ~ 1 TFLOP/s in DP
- Heterogeneous clustering

For highly parallel applications which reach the scaling limits on Intel Xeon processors
Programming Models

Native coprocessor applications
- Compile with -mmic
- Run with micnativeloadex or scp+ssh
- The way to go for MPI applications without offload

Explicit offload
- Functions, global variables require __attribute__((target(mic)))
- Initiate offload, data marshalling with #pragma offload
- Only bitwise-copyable data can be shared

Clusters and multiple coprocessors
- #pragma offload target(mic:i)
- Use threads to offload to multiple coprocessors
- Run native MPI applications
Native Execution

“Hello World” application:

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

int main()
{
    printf("Hello world! I have %ld logical cores.\n",
            sysconf(_SC_NPROCESSORS_ONLN));
}
```

Compile and run on host:

```
user@host% icc hello.c
user@host% ./a.out
Hello world! I have 32 logical cores.
user@host%
```
Native Execution

Compile and run the same code on the coprocessor in the native mode:

```
user@host% icc hello.c -mmic
user@host% scp a.out mic0:~/
a.out 100% 10KB 10.4KB/s 00:00
user@host% ssh mic0
user@mic0% pwd
/home/user
user@mic0% ls
a.out
user@mic0% ./a.out
Hello world! I have 240 logical cores.
user@mic0%
```

- Use `-mmic` to produce executable for MIC architecture
- Must transfer executable to coprocessor (or NFS-share) and run from shell
- Native MPI applications work the same way (need Intel MPI library)
Explicit Offload: Pragma-based approach

“Hello World” in the explicit offload model:

```c
#include <stdio.h>
int main(int argc, char * argv[]) {
    printf("Hello World from host!\n");
    #pragma offload target(mic)
    {
        printf("Hello World from coprocessor!\n"); fflush(0);
    }
    printf("Bye\n");
}
```

Application runs on the host, but some parts of code and date are moved (“offloaded”) the coprocessor.
Compiling and Running an Offload Application

```bash
user@host% icpc hello_offload.cpp -o hello_offload
user@host% ./hello_offload
Hello World from host!
Bye
Hello World from coprocessor!
```

- No additional arguments if compiled with an Intel compiler
- Run application on host as a regular application
- Code inside of `#pragma offload` is offloaded automatically
- Console output on Intel Xeon Phi coprocessor is buffered and mirrored to the host console
- If coprocessor is not installed, code inside `#pragma offload` runs on the host system
Functions used on coprocessor must be marked with the specifier
__attribute__((target(mic)))

Compiler produces a host version and a coprocessor version of such
functions (to enable fall-back to host)
Offloading Multiple Functions

```c
#pragma offload_attribute(push, target(mic))
void MyFunctionOne() {
    // ... implement function as usual
}

void MyFunctionTwo() {
    // ... implement function as usual
}
#pragma offload_attribute(pop)
```

- To mark a long block of code with the offload attribute, use `#pragma offload_attribute(push/pop)`. 
void MyFunction() {
    const int N = 1000;
    int data[N];
    #pragma offload target(mic)
    {
        for (int i = 0; i < N; i++)
            data[i] = 0;
    }
}

- Scope-local scalars and known-size arrays offloaded automatically
- Data is copied from host to coprocessor at the start of offload
- Data is copied back from coprocessor to host at the end of offload
- Bitwise-copyable data only (arrays of basic types and scalars)
  C++ classes, etc. should use virtual-shared memory model
Global and static variables must be marked with the offload attribute

#pragma offload_attribute(push/pop) may be used as well
Simple Applications, Native Execution

Simple CPU applications can be compiled for native execution on Xeon Phi coprocessors by supplying the flag “-mmic” to the Intel compiler:

```
user@host% icpc -c myobject1.cc -mmic
user@host% icpc -c myobject2.cc -mmic
user@host% icpc -o myapplication myobject1.o myobject2.o -mmic
```
SIMD Operations

SIMD — Single Instruction Multiple Data

Scalar Loop

```c
for (i = 0; i < n; i++)
```

SIMD Loop

```c
for (i = 0; i < n; i += 4)
    A[i:(i+4)] = A[i:(i+4)] + B[i:(i+4)];
```

Each SIMD addition operator acts on 4 numbers at a time.
## Instruction Sets in Intel Architectures

<table>
<thead>
<tr>
<th>Instruction Set</th>
<th>Year and Intel Processor</th>
<th>Vector registers</th>
<th>Packed Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMX</td>
<td>1997, Pentium</td>
<td>64-bit</td>
<td>8-, 16- and 32-bit integers</td>
</tr>
<tr>
<td>SSE</td>
<td>1999, Pentium III</td>
<td>128-bit</td>
<td>32-bit single precision FP</td>
</tr>
<tr>
<td>SSE2</td>
<td>2001, Pentium 4</td>
<td>128-bit</td>
<td>8 to 64-bit integers; SP &amp; DP FP</td>
</tr>
<tr>
<td>SSE3–SSE4.2</td>
<td>2004 – 2009</td>
<td>128-bit</td>
<td>(additional instructions)</td>
</tr>
<tr>
<td>AVX</td>
<td>2011, Sandy Bridge</td>
<td>256-bit</td>
<td>single and double precision FP</td>
</tr>
<tr>
<td>AVX2</td>
<td>2013, Haswell</td>
<td>256-bit</td>
<td>integers, additional instructions</td>
</tr>
<tr>
<td>IMCI</td>
<td>2012, Knights Corner</td>
<td>512-bit</td>
<td>32- and 64-bit integers; single &amp; double precision FP</td>
</tr>
<tr>
<td>AVX-512</td>
<td>(future) Knights Landing</td>
<td>512-bit</td>
<td>32- and 64-bit integers; single &amp; double precision FP</td>
</tr>
</tbody>
</table>
Explicit Vectorization: Compiler Intrinsics

**SSE2 Intrinsics**

```c
for (int i=0; i<n; i+=4) {
    __m128 Avec=_mm_load_ps(A+i);
    __m128 Bvec=_mm_load_ps(B+i);
    Avec=_mm_add_ps(Avec, Bvec);
    _mm_store_ps(A+i, Avec);
}
```

**IMCI Intrinsics**

```c
for (int i=0; i<n; i+=16) {
    __m512 Avec=_mm512_load_ps(A+i);
    __m512 Bvec=_mm512_load_ps(B+i);
    Avec=_mm512_add_ps(Avec, Bvec);
    _mm512_store_ps(A+i, Avec);
}
```

- The arrays float A[n] and float B[n] are aligned on a 16-byte (SSE2) and 64-byte (IMCI) boundary
- n is a multiple of 4 for SSE and a multiple of 16 for IMCI
- Variables Avec and Bvec are
  
  \[ 128 = 4 \times \text{sizeof(float)} \] bits in size for SSE2 and
  
  \[ 512 = 16 \times \text{sizeof(float)} \] bits for the Intel Xeon Phi architecture
A primer on Parallel Programming with POSIX Threads (Pthreads)
Overview of POSIX Threads

• POSIX: **Portable Operating System Interface for UNIX**
  • Interface to Operating System utilities
• Pthreads: The POSIX threading interface
  • System calls to create and synchronize threads
  • Should be relatively uniform across UNIX-like OS platforms
• Pthreads contain support for
  • Creating parallelism
  • Synchronizing
  • No explicit support for communication, because shared memory is implicit; a pointer to shared data is passed to a thread

• References
  • [https://computing.llnl.gov/tutorials/parallel_comp/](https://computing.llnl.gov/tutorials/parallel_comp/)
  • [https://computing.llnl.gov/tutorials/pthreads/](https://computing.llnl.gov/tutorials/pthreads/)
Creating Threads

Signature:

```c
int pthread_create(pthread_t * ,
const pthread_attr_t * ,
void * (*)(void *),
void *);
```

Example call:

```c
eretcode = pthread_create(&thread_id, &thread_attribute,
&thread_fun, &fun_arg);
```

- `thread_id` is the thread id or handle (used to halt, etc.)
- `thread_attribute` various attributes
  - standard default values obtained by passing a NULL pointer
- `thread_fun` the function to be run (takes and returns void*)
- `fun_arg` an argument can be passed to thread_fun when it starts
- `errorcode` will be set nonzero if the create operation fails
Important Pthreads Routines

- pthread_exit
- pthread_cancel
- pthread_join
- pthread_detach
void* SayHello(void *foo) {
    printf("Hello, world!\n");
    return NULL;
}

int main() {
    pthread_t threads[16];
    int tn;
    for(tn=0; tn<16; tn++) {
        pthread_create(&threads[tn], NULL, SayHello, NULL);
    }
    for(tn=0; tn<16 ; tn++) {
        pthread_join(threads[tn], NULL);
    }
    return 0;
}
Parallelism Overheads

- Thread creation overhead is non-trivial
- Do not spawn threads for small jobs
- Too many threads can lead to performance degradation
- Ex. Sum of squares of N integers
Useful Thread Control Functions

```c
pthread_t me; me = pthread_self();
```

- Allows a pthread to obtain its own identifier `pthread_t thread`

```c
pthread_detach(thread);
```

- Informs the library that the threads exit status will not be needed by subsequent `pthread_join` calls resulting in better threads performance. For more information consult the man pages, e.g., `man -k pthread`. 
Shared Data and Threads

- Object allocated on the heap may be shared (if pointer is passed)
- Variables on the stack are private: passing pointer to these around to other threads can cause problems

- Often done by creating a large “thread data” struct
  - Passed into all threads as argument
  - Simple example:
    ```c
    char *message = "Hello World!\n";
    
    pthread_create( &thread1, 
                  NULL, 
                  (void*) &print_fun, 
                  (void*) message);
    ```
(Details: Setting Attribute Values)

- Once an initialized attribute object exists, changes can be made. For example:
  - To change the stack size for a thread to 8192 (before calling `pthread_create`), do this:
    - `pthread_attr_setstacksize(&my_attributes, (size_t)8192);`
  - To get the stack size, do this:
    - `size_t my_stack_size;`  
      `pthread_attr_getstacksize(&my_attributes, &my_stack_size);`

- Other attributes:
  - Detached state – set if no other thread will use `pthread_join` to wait for this thread (improves efficiency)
  - Guard size – use to protect against stack overflow
  - Inherit scheduling attributes (from creating thread) – or not
  - Scheduling parameter(s) – in particular, thread priority
  - Scheduling policy – FIFO or Round Robin
  - Contention scope – with what threads does this thread compete for a CPU
  - Stack address – explicitly dictate where the stack is located
  - Lazy stack allocation – allocate on demand (lazy) or all at once, “up front”
Basic Types of Synchronization: Barrier

Barrier -- global synchronization

- Especially common when running multiple copies of the same function in parallel
  - SPMD “Single Program Multiple Data”

- Simple use of barriers -- all threads hit the same one
  
  ```
  work_on_my_problem();
  barrier;
  get_data_from_others();
  barrier;
  ```

- More complicated -- barriers on branches (or loops)
  
  ```
  if (tid % 2 == 0) {
    work1();
    barrier
  } else { barrier }
  ```
Creating and Initializing a Barrier

• To (dynamically) initialize a barrier, use code similar to this (which sets the number of threads to 3):
  
  ```c
  pthread_barrier_t b;
  pthread_barrier_init(&b, NULL, 3);
  ```

• The second argument specifies an object attribute; using NULL yields the default attributes.

• To wait at a barrier, a process executes:
  
  ```c
  pthread_barrier_wait(&b);
  ```

• This barrier could have been statically initialized by assigning an initial value created using the macro
  
  ```c
  PTHREAD_BARRIER_INITIALIZER(3).
  ```

Note: barrier is not in all pthreads implementations
Basic Types of Synchronization: Mutexes

Mutexes -- mutual exclusion aka locks
- threads are working mostly independently
- need to access common data structure

```c
lock *l = alloc_and_init(); /* shared */
acquire(l);
access data
release(l);
```
Mutexes in POSIX Threads

- To create a mutex:
  ```c
  #include <pthread.h>
  pthread_mutex_t amutex = PTHREAD_MUTEX_INITIALIZER;
  pthread_mutex_init(&amutex, NULL);
  ```

- To use it:
  ```c
  int pthread_mutex_lock(&amutex);
  **access critical data**
  int pthread_mutex_unlock(&amutex);
  ```

- To deallocate a mutex
  ```c
  int pthread_mutex_destroy(pthread_mutex_t *mutex);
  ```

- Multiple mutexes may be held, but can lead to deadlock:
  ```plaintext
  thread1   thread2
  lock(a)   lock(b)
  lock(b)   lock(a)
  ```
Shared Memory Programming

Several other thread libraries besides PTHREADS
• E.g., Solaris threads are very similar
• Other older libraries P4, Parmacs, etc.
• OpenMP can also be used for shared memory parallel programmer
  • [http://www.openMP.org](http://www.openMP.org)
  • Easier to use, i.e., just mark a loop as parallel
  • But not available everywhere
  • And performance is harder to control
Summary of Programming with Threads

• POSIX Threads are based on OS features
  • Can be used from multiple languages (need appropriate header)
  • Familiar language for most of program
  • Ability to shared data is convenient

• Pitfalls
  • Data race bugs are very nasty to find because they can be intermittent
  • Deadlocks are usually easier, but can also be intermittent

• Researchers look at transactional memory an alternative
• OpenMP is commonly used today as an alternative