EECS 570 Programming Assignment 1

Discussion

January 17  2020
Announcements

Project

• Search for Teammates! on piazza
  Register Teams: forms.gle/dctvkcbj5SroepoL6

• Fri 1/24: Discussion, Project Handout

• Wed 1/29 11:59p: Proposal due on Canvas

PA1

Due Fri 2/7 11:59p on Canvas

No GSI OH next week, only piazza
1. 3D Ultrasound Beamforming

2. Intel Xeon Phi

3. POSIX Threads (Tutorial)

PA 1 Logistics
3D Ultrasound Beamforming
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

\[ \tau \]
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
Intex Xeon Processors and the MIC Architecture

- Multi-core Intel Xeon processor
- C/C++/Fortran; OpenMP/MPI
- Standard Linux OS
- Up to 768 GB of DDR3 RAM
- $\geq 12$ cores/socket $\approx 3$ GHz
- 2-way hyper-threading
- 256-bit AVX vectors

- Many-core Intel Xeon Phi coprocessor
- C/C++/Fortran; OpenMP/MPI
- Special Linux $\mu$OS distribution
- 6-16 GB cached GDDR5 RAM
- 57-61 cores at $\approx 1$ GHz
- 4-way hyper-threading
- 512-bit IMCI vectors
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 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
Example ("Hello World" application)
#include <stdio.h>
#include <unistd.h>
int main() {
    printf("Hello world! I have %ld logical cores.\n", 
           sysconf(_SC_NPROCESSORS_ONLN ));
}

Example (compile and run on host)
user@host% icc -o hello hello.c
user@host% ./hello
Hello world! I have 32 logical cores.
user@host% _
Compile and run the same code on the coprocessor in native mode:

**Example (compile and run on coprocessor)**

```
user@host% icc -o hello.mic hello.c -mmic
user@host% micnativeloadex hello.mic -t 300 -d 0
Hello world! I have 240 logical cores.
```

- Use `-mmic` to produce executable for MIC architecture
- Use `micnativeloadex` to run the executable on the coprocessor
- Native MPI applications work the same way (need Intel MPI library)
POSIX Threads (Tutorial)
Bonus

software.intel.com/sites/landingpage/IntrinsicsGuide/
SIMD Operations

SIMD — Single Instruction Multiple Data

Scalar Loop

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

SIMD Loop

```
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 × sizeof(float) bits in size for SSE2 and 512 = 16 × sizeof(float) bits for the Intel Xeon Phi architecture