High Performance Computing for Science and Engineering

UMA/NUMA, OpenMP Part 2, Pipelining and Vectorization

Fabian Wermelinger
Computational Science & Engineering Laboratory
OUTLINE

• Lecture 1
  • Uniform and Non-Uniform Memory Access
  • OpenMP Processor Binding and Performance
  • Amdahl’s Law and Strong Scaling Analysis

• Lecture 2
  • Instruction Set Architecture (ISA)
  • Pipelining
  • ISA Extensions and Data Alignment
  • Intel SPMD Program Compiler
In first lecture about shared memory:

Each processor accesses the same physical memory:

**Uniform Memory Access (UMA)**

The discussion we shared so far was concerned with **UMA** platforms.

https://computing.llnl.gov/tutorials/openMP/
A set of processors have direct access to their own physical memory:

**Non-Uniform Memory Access (NUMA)**

Two different sockets with separate memory management units (MMU)
A set of processors have direct access to their own physical memory:

**Non-Uniform Memory Access (NUMA)**

Two different sockets with separate memory management units (MMU)

- UMA typically features few cores (up to 8, e.g. your laptop or desktop at home).
- This allows for hardware implementations with symmetric access to memory (the bus can still service the memory requests)
- UMA is also referred to as symmetric multiprocessors (SMP)

- Supporting large processor counts requires distributed shared memory
- Otherwise the pressure on the bus to service memory requests becomes too large (avoiding very high latencies)
- Such architectures have multiple sockets with separate memory modules
- NUMA is also referred to as distributed shared memory (DSM)

https://computing.llnl.gov/tutorials/openMP/
NUMA Platforms

• Both UMA and NUMA are shared memory architectures. They share the same memory space. The physical memory may be distributed.
• Distributing the memory increases the bandwidth and reduces the latency to local memory.
• Memory references to a remote memory module have a higher latency than local references (how does this impact performance?)
• Note the difference: In distributed programming (MPI) it is not possible to access the remote memory without the assistance of software protocols running on both nodes (this is exactly what MPI does)
• However, also NUMA requires additional effort in software to benefit from the higher bandwidth
A common workflow in a code:

1.) Memory allocation on the heap

2.) Initialize the data to some value

3.) Distribute the work among available processor cores

Assume we are working on a NUMA platform. In which memory module(s) will the allocation in step 1.) be placed?
• Memory affinity is not decided by the memory allocation but by the initialization!
• **First touch policy**: Memory is mapped to the NUMA domain that first touches it.

---

Credit: C. L. Luengo Hendriks

---

Core 1 first touched this data element. It will be placed in the NUMA domain of core 1.

Reads from this memory location issued from a core that is not in this NUMA domain are slower (**higher latency**).
NUMA First Touch Policy

**NUMA touch with OpenMP:**

```
int main(int argc, char* argv[]) {
    double* const A = new double[1000];

    // initialize data: NUMA first touch policy
    #pragma omp parallel for
    for (int i = 0; i < 1000; ++i)
        A[i] = 0.0;

    // other code...

    // perform work in parallel
    #pragma omp parallel for
    for (int i = 0; i < 1000; ++i)
        A[i] = do_work(i, A);

    return 0;
}
```

1.) Memory allocation on the heap. No memory is touched yet.

2.) Distribute the data and perform the first touch with the corresponding core.

3.) Perform the computations. Ensure memory references hit in the correct NUMA node.

**Note:** Depending on how A is accessed inside `do_work` you might access memory outside the NUMA domain. In such case, a different data representation may be more beneficial (e.g. reorder data in a block-structured layout)
**NUMA First Touch Policy**

**NUMA touch with OpenMP:**

```cpp
#include <vector>
using namespace std;

int main(int argc, char* argv[]) {
    // vector<double> A(1000, 0.0); // violates first touch
    vector<double> A(1000); // depends on what default constructor does

    // might not be first touch!
    #pragma omp parallel for
    for (int i = 0; i < 1000; ++i)
        A[i] = 0.0;

    // other code...

    // perform work in parallel
    #pragma omp parallel for
    for (int i = 0; i < 1000; ++i)
        A[i] = do_work(i, A);

    return 0;
}
```

Careful with container types. The constructor may (without knowing what your intention is) touch the data.
The Euler nodes you have access to (Intel Xeon E5 2680v3):

**Query host**: `lshosts -l eu-c7-045-15`

<table>
<thead>
<tr>
<th>Node</th>
<th>Stat</th>
<th>r15s</th>
<th>r1m</th>
<th>r15m</th>
<th>Perf</th>
<th>Mem</th>
<th>MemMax</th>
<th>ISA</th>
<th>Model</th>
<th>S</th>
<th>C/S</th>
<th>T/C</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>*eu-c7-045-15 ok</td>
<td>0.1</td>
<td>0.0</td>
<td>0.7</td>
<td>1%</td>
<td>60672.0M</td>
<td>63.8G</td>
<td>X86_64</td>
<td>XeonE5_2680v3</td>
<td>2</td>
<td>12</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of cores on node</th>
<th>Node name</th>
<th>Memory free</th>
<th>Max. Memory</th>
<th>CPU model</th>
<th>Number of sockets (2 CPUs, NUMA)</th>
<th>Cores per socket</th>
<th>Logical threads per core (hyperthreading)</th>
</tr>
</thead>
</table>

**NUMA on Euler**
NUMA on Euler

The Euler nodes you have access to (Intel Xeon E5 2680v3):

**Query host:** `lshosts -l eu-c7-045-15`

<table>
<thead>
<tr>
<th>Node</th>
<th>Stat</th>
<th>r15s</th>
<th>r1m</th>
<th>r15m</th>
<th>Perf</th>
<th>Mem</th>
<th>MemMax</th>
<th>ISA</th>
<th>Model</th>
<th>S</th>
<th>C/S</th>
<th>T/C</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>24*eu-c7-045-15</td>
<td>ok</td>
<td>0.1</td>
<td>0.0</td>
<td>0.7</td>
<td>1%</td>
<td>60672.0M</td>
<td>63.8G</td>
<td>X86_64</td>
<td>XeonE5_2680v3</td>
<td>2</td>
<td>12</td>
</tr>
</tbody>
</table>

**Get interactive node (for 1 hour):** `bsub -W 01:00 -n 24 -R fullnode -Is bash`
NUMA on Euler

The Euler nodes you have access to (Intel Xeon E5 2680v3):

Another useful tool: hwloc-1s
https://www.open-mpi.org/projects/hwloc/

NUMA nodes and attached memory

Cache hierarchy

Cores with physical ID and hyperthreads

I/O devices

L1 data and instruction caches
Test NUMA First Touch Policy on Euler

Measurements on Euler using the STREAM benchmark: [https://www.cs.virginia.edu/stream/](https://www.cs.virginia.edu/stream/)

```c
/* Get initial value for system clock. */

#pragma omp parallel for
for (j=0; j<STREAM_ARRAY_SIZE; j++) {
  a[j] = 1.0;
  b[j] = 2.0;
  c[j] = 0.0;
}

t = mysecond();

#pragma omp parallel for
for (j = 0; j < STREAM_ARRAY_SIZE; j++)
  a[j] = 2.0E0 * a[j];

t = 1.0E6 * (mysecond() - t);
```
NUMA on Euler

Roofline for this Euler node (see Exercise 1):

You can only reach the upper ceiling in the memory bound region with a NUMA aware code!
OUTLINE

• Lecture 1
  • Uniform and Non-Uniform Memory Access
  • OpenMP Processor Binding and Performance
  • Amdahl’s Law and Strong Scaling Analysis

• Lecture 2
  • Instruction Set Architecture (ISA)
  • Pipelining
  • ISA Extensions and Data Alignment
  • Intel SPMD Program Compiler
OpenMP Processor Binding and Affinity

Where are OpenMP threads mapped to on a NUMA architecture?

- Mapping considerations are important for NUMA architectures
- OpenMP supports processor binding since the 3.0 specification

Easiest way with environment variables

Affinity control:
- `OMP_PROC_BIND='true', 'false'`

There are other possibilities not discussed in class. See the specification for all details
You can tell OpenMP where it can map threads with the `OMP_PLACES` environment variable.

**Affinity control:**
- `OMP_PLACES=.blitlist of places`

**Example: Euler node**
- `OMP_PLACES=‘threads(48)’`
- `OMP_PLACES=‘cores(24)’`
- `OMP_PLACES=‘sockets(2)’`

There are many possibilities! See the specification for all details.

Specify a number of hardware **threads** for possible places. This includes hyper-threads.

Specify a number of **cores** for possible places. Each core may have a certain number of hardware threads.

Specify a number of **sockets** for possible places. Each socket may consist of a certain number of cores.
OpenMP Processor Binding and Affinity

You can tell OpenMP your desired processor affinity in your code.

```c
#pragma omp parallel proc_bind(master|close|spread) new-line
structured block
```

- **master**: This thread affinity policy instructs the execution environment to assign every thread in the team to the same place as the master thread.
- **close**: This thread affinity policy instructs the execution environment to assign the threads in the team to places close to the place of the parent thread (may not necessarily be the master in nested parallelism).
- **spread**: This thread affinity policy creates a sparse distribution of the threads in the available places.

Note: You may use other clauses here, of course.
Let’s test this:

```cpp
int main(int argc, char* argv[])
{
    int nThreads;
    #pragma omp parallel
    {
        #pragma omp master
        nThreads =omp_get_num_threads();
    }
    vector<string> messages(nThreads);
    #pragma omp parallel
    {
        const int tid = omp_get_thread_num();
        int cpuid, nodeid;
        tacc_rdtscp(&cpuid, &nodeid);
        string hostname(1024, '\0');
        gethostname(&hostname.front(), hostname.size());
        ostringstream mystream;
        mystream << "[thread=" << tid << "]\t"
                  << "Running on host " << hostname.c_str() << "\nCPU " << cpuid << "\tNUMA NODE " << nodeid;
        #pragma omp critical
        messages[tid] = mystream.str();
    }
    for (const string& message : messages)
        cout << message << endl;
    return 0;
}
```

Returns the processor ID and NUMA node (not portable)

```cpp
unsigned long tacc_rdtscp(int *core, int *node)
{
    unsigned long a,d,c;
    __asm__ _volatile("rdtscp" : =a" (a), =d" (d), =c" (c));
    *node = (c & 0xFFF000) >> 12;
    *core = c & 0xFFF;
    return ((unsigned long)a) | (((unsigned long)d) << 32);;
}
```
Let’s test this:

No processor affinity:

1. `export OMP_PROC_BIND=false`
2. `export OMP_NUM_THREADS=4`

3. [thread=0/4] Running on host eu-c7-061-02 CPU 37 NUMA NODE 1
4. [thread=1/4] Running on host eu-c7-061-02 CPU 25 NUMA NODE 0
5. [thread=2/4] Running on host eu-c7-061-02 CPU 26 NUMA NODE 0
6. [thread=3/4] Running on host eu-c7-061-02 CPU 38 NUMA NODE 1

OpenMP will schedule threads according to available resources (default for GCC)
OpenMP Processor Binding and Affinity

Let's test this:

No processor affinity:

```
1 export OMP_PROC_BIND='false'
2 export OMP_NUM_THREADS=4
3 [thread=0/4] Running on host eu-c7-061-02 CPU 37 NUMA NODE 1
4 [thread=1/4] Running on host eu-c7-061-02 CPU 25 NUMA NODE 0
5 [thread=2/4] Running on host eu-c7-061-02 CPU 26 NUMA NODE 0
6 [thread=3/4] Running on host eu-c7-061-02 CPU 38 NUMA NODE 1
```

OpenMP will schedule threads according to available resources (default for GCC)

With processor affinity enabled:

```
1 export OMP_PROC_BIND='true'
2 export OMP_NUM_THREADS=4
3 [thread=0/4] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0
4 [thread=1/4] Running on host eu-c7-061-02 CPU 1 NUMA NODE 0
5 [thread=2/4] Running on host eu-c7-061-02 CPU 2 NUMA NODE 0
6 [thread=3/4] Running on host eu-c7-061-02 CPU 3 NUMA NODE 0
```

With processor affinity enabled, the default binding order is given by increasing physical core ID's (see numactl and/or hwloc-1s)
OpenMP Processor Binding and Affinity

With `#pragma omp parallel proc_bind(master)`:

```
1. export OMP_PROC_BIND='true'
2. export OMP_NUM_THREADS=4
3. [thread=0/4] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0
4. [thread=1/4] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0
5. [thread=2/4] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0
6. [thread=3/4] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0
```

All threads map to the place partition of the master thread.
OpenMP Processor Binding and Affinity

With `#pragma omp parallel proc_bind(master)`: All threads map to the place partition of the master thread

1. `export OMP_PROC_BIND='true'`
2. `export OMP_NUM_THREADS=4`
3. `[thread=0/4] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0`
4. `[thread=1/4] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0`
5. `[thread=2/4] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0`
6. `[thread=3/4] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0`

With `#pragma omp parallel proc_bind(close)`: All threads map to a place partition close to the parent thread (here `master`)

1. `export OMP_PROC_BIND='true'`
2. `export OMP_NUM_THREADS=4`
3. `[thread=0/4] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0`
4. `[thread=1/4] Running on host eu-c7-061-02 CPU 1 NUMA NODE 0`
5. `[thread=2/4] Running on host eu-c7-061-02 CPU 2 NUMA NODE 0`
6. `[thread=3/4] Running on host eu-c7-061-02 CPU 3 NUMA NODE 0`
OpenMP Processor Binding and Affinity

With `#pragma omp parallel proc_bind(master)`:  

1. `export OMP_PROC_BIND='true'`
2. `export OMP_NUM_THREADS=4`
3. [thread=0/4] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0
4. [thread=1/4] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0
5. [thread=2/4] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0
6. [thread=3/4] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0

All threads map to the place partition of the master thread

With `#pragma omp parallel proc_bind(close)`:  

1. `export OMP_PROC_BIND='true'`
2. `export OMP_NUM_THREADS=4`
3. [thread=0/4] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0
4. [thread=1/4] Running on host eu-c7-061-02 CPU 1 NUMA NODE 0
5. [thread=2/4] Running on host eu-c7-061-02 CPU 2 NUMA NODE 0
6. [thread=3/4] Running on host eu-c7-061-02 CPU 3 NUMA NODE 0

All threads map to a place partition close to the parent thread (here `master`)

With `#pragma omp parallel proc_bind(spread)`:  

1. `export OMP_PROC_BIND='true'`
2. `export OMP_NUM_THREADS=4`
3. [thread=0/4] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0
4. [thread=1/4] Running on host eu-c7-061-02 CPU 12 NUMA NODE 1
5. [thread=2/4] Running on host eu-c7-061-02 CPU 24 NUMA NODE 0
6. [thread=3/4] Running on host eu-c7-061-02 CPU 36 NUMA NODE 1

Threads are sparsely mapped over the set of all possible places. We have 48 threads total and 4 threads to map, the stride will be 12 in this case.
OpenMP Processor Binding and Affinity

What happens when we request more threads than the hardware can deliver?

**Oversubscription**

```
1 export OMP_PROC_BIND='true'
2 export OMP_NUM_THREADS=50
3 [thread=0/50] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0
4 [thread=1/50] Running on host eu-c7-061-02 CPU 1 NUMA NODE 0
... 49 [thread=46/50] Running on host eu-c7-061-02 CPU 46 NUMA NODE 1
50 [thread=47/50] Running on host eu-c7-061-02 CPU 47 NUMA NODE 1
51 [thread=48/50] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0
52 [thread=49/50] Running on host eu-c7-061-02 CPU 1 NUMA NODE 0
```

With an arbitrary number of threads, OpenMP continues the affinity pattern at the end. In this example, CPU 0 and 1 are assigned two threads.
OpenMP Processor Binding and Affinity

What happens when we request more threads than the hardware can deliver?

Oversubscription

With an arbitrary number of threads, OpenMP continues the affinity pattern at the end. In this example, CPU 0 and 1 are assigned two threads.

If the number of threads is evenly divisible by the number of available places, OpenMP will distribute the threads evenly with this pattern.

Note: The same pattern is observed if we set OMP_PLACES='threads(48)'
OpenMP Processor Binding and Affinity

What happens when we request more threads than the hardware can deliver?

With \texttt{#pragma omp parallel proc_bind(close)}:

1. \texttt{export OMP_PROC_BIND='true'}
2. \texttt{export OMP_PLACES='cores(24)'}
3. \texttt{export OMP_NUM_THREADS=96}
4. \texttt{[thread=0/96] Running on host eu-c7-061-02 CPU 0 NUMA NODE 0}
5. \texttt{[thread=1/96] Running on host eu-c7-061-02 CPU 24 NUMA NODE 0}
6. \texttt{[thread=2/96] Running on host eu-c7-061-02 CPU 24 NUMA NODE 0}
7. \texttt{[thread=3/96] Running on host eu-c7-061-02 CPU 24 NUMA NODE 0}
8. \texttt{[thread=4/96] Running on host eu-c7-061-02 CPU 1 NUMA NODE 0}
9. \texttt{[thread=5/96] Running on host eu-c7-061-02 CPU 1 NUMA NODE 0}
10. \texttt{[thread=6/96] Running on host eu-c7-061-02 CPU 1 NUMA NODE 0}
11. \texttt{[thread=7/96] Running on host eu-c7-061-02 CPU 1 NUMA NODE 0}
   ...
92. \texttt{[thread=88/96] Running on host eu-c7-061-02 CPU 22 NUMA NODE 1}
93. \texttt{[thread=89/96] Running on host eu-c7-061-02 CPU 22 NUMA NODE 1}
94. \texttt{[thread=90/96] Running on host eu-c7-061-02 CPU 22 NUMA NODE 1}
95. \texttt{[thread=91/96] Running on host eu-c7-061-02 CPU 22 NUMA NODE 1}
96. \texttt{[thread=92/96] Running on host eu-c7-061-02 CPU 47 NUMA NODE 1}
97. \texttt{[thread=93/96] Running on host eu-c7-061-02 CPU 23 NUMA NODE 1}
98. \texttt{[thread=94/96] Running on host eu-c7-061-02 CPU 23 NUMA NODE 1}
99. \texttt{[thread=95/96] Running on host eu-c7-061-02 CPU 23 NUMA NODE 1}

If we specify \texttt{OMP_PLACES='cores(24)'} , then 4 threads are mapped to each core. The same happens for \texttt{proc_bind(spread)}.

Note: CPU 0 and 24 are on the same core.
OpenMP Overheads

What is the cost of spawning a parallel region?

• A useful benchmark for measuring overhead can be found here in this link: https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-openmp-micro-benchmark-suite
• You can measure the overhead of various OpenMP constructs
  ‣ Synchronization (parallel, parallel for, for, barrier, atomic, reduction, …)
  ‣ Loop scheduling (static, dynamic, guided and different chunk sizes)
  ‣ Tasking (not covered in this course)
OpenMP Overheads

Measurements on Euler with gcc:

OMP_PROC_BIND='false'

OMP_PROC_BIND='true'

gcc does benefit slightly from processor binding.
OpenMP Overheads

Measurements on Euler with clang:

**OMP_PROC_BIND=‘false’**

clang reductions do slightly improve with processor binding. Overall performance is superior to gcc for this platform.

**OMP_PROC_BIND=‘true’**

Roughly constant overhead for large number of threads.
OpenMP Overheads

Measurements on Euler with Intel (icc):

OMP_PROC_BIND='false'

icc and clang have similar overhead for this platform
Spawning and joining threads is expensive
  ‣ They are system calls to the OS
The OpenMP runtime library spawns threads only once
  ‣ When it encounters the first parallel region
  ‣ The threads are re-used at the next parallel region
This means that after the end of a parallel region
  ‣ Only the master thread continues
  ‣ The other threads become idle

What happens after OpenMP has spawned the first parallel region?

```c
int main(int argc, char* argv[])
{
    #pragma omp parallel
    {
        // first parallel region:
        // threads are spawned here
    }

    #pragma omp parallel
    {
        // second parallel region:
        // threads are re-used here
    }
    return 0;
}
```
OpenMP Performance

How should idle threads spend their time?

- You can specify a hint to OpenMP how threads should wait with the `OMP_WAIT_POLICY` environment variable.
- The variable cannot be changed during runtime.
- It can take two possible values (default is implementation defined):
  - `OMP_WAIT_POLICY='active'`: Waiting threads should mostly be active, consuming processor cycles while waiting (busy-wait, spinning).
  - `OMP_WAIT_POLICY='passive'`: Waiting threads should mostly be passive, not consuming processor cycles while waiting (threads yield the processor to other threads or go to sleep).
OUTLINE

• Lecture 1
  • Uniform and Non-Uniform Memory Access
  • OpenMP Processor Binding and Performance
  • Amdahl’s Law and Strong Scaling Analysis

• Lecture 2
  • Instruction Set Architecture (ISA)
  • Pipelining
  • ISA Extensions and Data Alignment
  • Intel SPMD Program Compiler
Recall Amdahl’s Law from the first lecture:

\[ S_p = \frac{1}{f + \frac{1-f}{p}} \]

I wrote a shared memory code. How well does my code run in parallel?
Recall Amdahl’s Law from the first lecture:

\[ S_p = \frac{1}{f + \frac{1-f}{p}} \]

I wrote a shared memory code. How well does my code run in parallel?

In a picture:

- **Serial execution**: \(f\)
- **Parallel execution** (\(p\) processors): \(\frac{1-f}{p}\)
- **Serial fraction of the code**: \(f\)
- **Parallel fraction of the code**: \(\frac{1-f}{p}\)

**Speedup** \(S\) with \(p\) processors
**Strong Scaling Analysis**

**Implicit assumptions in Amdahl’s Law:**

- **Fixed problem size**
  - Makes sense if $p$ is relatively small
  - Often we want to keep the execution time constant and increase the problem size (weak scaling)

- **Negligible communication cost**
  - The number of processors $p$ should be small

- **All-or-None parallelism**
  - A more realistic model would be:

\[
S_p = \frac{1}{f_1 + \sum_{i=2}^{p} \frac{f_i}{i}} \quad \text{with} \quad \left( \sum_{i=1}^{p} f_i = 1 \right)
\]

Problems for which those assumptions are reasonable can use this model for performance analysis. Such analysis is called **Strong Scaling**.

**Recall:** Shared memory architectures cannot implement a large number of processors due to limitations on the memory bus as well as related cost issues. Communication cost using shared memory is still relatively low compared to distributed memory models.
**Strong Scaling Analysis**

**Implication of fixed problem size:**

- Speed of a certain task: \( \frac{W}{t} \)
  - associated work (problem size)
  - time needed to complete the work

- Speed for serial task: \( \frac{w}{t_1} \)
- Speed for parallel task: \( \frac{w}{t_p} \)

**Strong scaling speedup:**

\[
S_p = \frac{w/t_p}{w/t_1} = \frac{t_1}{t_p}
\]
Implication of serial fraction $f$:

Strong Scaling Analysis

The serial fraction implies a performance upper-bound:

$$\lim_{p \to \infty} S_p = \frac{1}{f}$$

Even with an infinite amount of processors, this is the best we could do. Strong scaling analysis is very sensitive towards the serial fraction. Communication overhead (e.g. synchronization) further degrades performance.
Strong Scaling Analysis

Recall last week’s false sharing problem (approximation of $\pi$):

$S_p = \frac{t_1}{t_p}$  
(Speedup is computed by benchmarking execution time)

- We would expect a large parallel fraction in that code given the algorithm to approximate $\pi$
- We used strong scaling analysis to test our implementation
- Strong scaling will tell you whether there is an issue in the implementation!
OUTLINE

• Lecture 1
  • Uniform and Non-Uniform Memory Access
  • OpenMP Processor Binding and Performance
  • Amdahl’s Law and Strong Scaling Analysis

• Lecture 2
  • Instruction Set Architecture (ISA)
  • Pipelining
  • ISA Extensions and Data Alignment
  • Intel SPMD Program Compiler
Instruction Set Architecture (ISA)

Instruction Set Architecture:

- Defines a basic set of ‘tools’ the CPU can work with
  - 32bit or 64bit instructions (e.g. x86 or x86_64)
- Specifies memory addressing
  - Little Endian or Big Endian byte order
- Reduced Instruction Set Computer (RISC) — e.g. MIPS, PowerPC or Sun SPARC
  - Instruction set is limited to simple instructions, fast (can do many instructions per second), ideally latency of one instruction is one clock cycle, easier to implement
- Complex Instruction Set Computer (CISC)
  - Can do more complex instructions, thus slower, chip circuitry is much more complex
  - Different instructions take different amount of time to execute

Some basic instructions:
- Load / Store
- Jumps and branches (address jumps)
- Add / Multiply numbers (ALU instructions)
- Input / Output devices
All operands are registers. They must be fetched with load/store instructions.

Operands may also be memory references.

Recent ISA implement these formats.
Consider the expression:

\[(a \times b) - (b \times c) - (a \times d)\]

- On a register-register / register-memory computer, the multiplications may be evaluated in any order
  - More efficient because of the location of the operands
  - Allows pipelining concerns (discussed next)
Instruction Set Architecture Classes

Consider the expression:

\[(a \times b) - (b \times c) - (a \times d)\]

- On a register-register / register-memory computer, the multiplications may be evaluated in any order
  - More efficient because of the location of the operands
  - Allows pipelining concerns (discussed next)

<table>
<thead>
<tr>
<th>Type</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register-Register</td>
<td>Simple, fixed-length instruction encoding, simple code generation model, instructions take similar numbers of clocks to execute</td>
<td>Higher instruction count than architectures with memory references in instructions, more instructions lead to larger programs (machine code)</td>
</tr>
<tr>
<td>(0,3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARM, MIPS, PowerPC, Sun SPARC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register-Memory</td>
<td>Data can be accessed without a separate preceding load instruction, instruction format tends to be easy to encode and yields good density</td>
<td>Clocks per instruction vary by operand location, requires encoding a register ID and a memory address in each instruction</td>
</tr>
<tr>
<td>(1,2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBM 360/370, Motorola 68000, Intel x86</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory-Memory</td>
<td>Most compact, does not waste registers for temporaries</td>
<td>Memory accesses create memory bottleneck, large variation in instruction size and work per instruction. (Not used today)</td>
</tr>
<tr>
<td>(2,2) or (3,3) VAX</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
OUTLINE

• Lecture 1
  • Uniform and Non-Uniform Memory Access
  • OpenMP Processor Binding and Performance
  • Amdahl’s Law and Strong Scaling Analysis

• Lecture 2
  • Instruction Set Architecture (ISA)
    • Pipelining
  • ISA Extensions and Data Alignment
  • Intel SPMD Program Compiler
Pipelining is an implementation technique (on the processor level) that takes advantage of the parallelism that exists among the actions required to execute an instruction.

- Pipelining is the key technique to make today's CPUs fast.
- It is based on the fact that to execute one instruction, multiple clock cycles are required.
- Therefore, the pipelining technique reduces the number of clock cycles per instruction (CPI).
- A pipeline is similar to an assembly line in the automobile industry, for example.
Pipelining

We will study pipelining based on a simple register-register RISC architecture (MIPS):

- All instructions work on data stored in registers
- The only operations that affect memory are load and store operations (move data from memory to a register or move data from a register to memory, respectively)
- The instruction format consists of a few elements, with all instructions being the same size
Pipelining

We will study pipelining based on a simple register-register RISC architecture (MIPS):

• All instructions work on data stored in registers
• The only operations that affect memory are load and store operations (move data from memory to a register or move data from a register to memory, respectively)
• The instruction format consists of a few elements, with all instructions being the same size

With these guidelines we can construct a simple instruction set, where we assume that every instruction can be implemented in at most 5 clock cycles:

1. Instruction fetch cycle (IF): Fetch the current instruction from memory.
2. Instruction decode / register fetch cycle (ID): Decode and read register source specifiers. Performs a test to detect a possible branch instruction.
3. Execution / effective address cycle (EX): The ALU performs the operation depending on the fetched instruction.
4. Memory access (MEM): In case of a load, the memory is read using the effective address. In case of a store, the register read in step 2 is written using the effective address.
5. Write-back cycle (WB): Write the result of step 3 (ALU) or step 4 (in case of a load) back to the register file.
Using our simple instruction set, we can construct a 5-stage pipeline:

<table>
<thead>
<tr>
<th>Instruction number</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction i</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction i+1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction i+2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction i+3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction i+4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5-Stage Pipeline

- **Note**: The fastest instruction in our ISA is 2 clock cycles (branch), 4 clock cycles for stores and every other instruction is 5 clock cycles. The pipeline throughput is bound to the instruction (in the set) with the highest latency (5 cycles in this case).

- **Note**: The pipeline is full in the 5th clock cycle and has full throughput for the following instructions.

A full pipeline has a throughput of one instruction every clock cycle!
Pipeline Reality Check

- Ideally you get a 5x speedup
- In reality, there is **overhead** associated with pipelining
- Amdahl’s law: Speedup is limited by overhead

**Example:**

We use our pipeline implementation from above. Assume that a clock cycle takes 1 ns and we need 4 cycles for ALU operations and branches, and 5 cycles for memory operations. Assume that these operations have a relative frequency of 40%, 20% and 40%, respectively. Furthermore, due to clock skew and setup the pipelined processor adds 0.2 ns of overhead to the clock. What is the speedup of the pipelined processor?
Pipeline Reality Check

• Ideally you get a 5x speedup
• In reality, there is overhead associated with pipelining
• Amdahl’s law: Speedup is limited by overhead

Example:
We use our pipeline implementation from above. Assume that a clock cycle takes 1 ns and we need 4 cycles for ALU operations and branches, and 5 cycles for memory operations. Assume that these operations have a relative frequency of 40%, 20% and 40%, respectively. Furthermore, due to clock skew and setup the pipelined processor adds 0.2 ns of overhead to the clock. What is the speedup of the pipelined processor?

Average instruction execution time = clock cycle x average CPI = 1 ns x [(0.4 + 0.2)x4 + 0.4x5] = 4.4 ns

Speedup from pipelining = \( \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}} \) = \( \frac{4.4 \text{ ns}}{1.2 \text{ ns}} \) = 3.7x speedup (Amdahl’s Law)

Full pipeline: One instruction every clock + overhead
Pipeline Hazards

The pipelining concept seems easy, but it is not!

Pipeline **hazards**: 

- **Structural hazards**: Arise from resource conflicts when the hardware cannot support all possible combinations simultaneously
- **Data hazards**: The next instruction may depend on the result of the previous instruction
- **Control hazards**: Arise from branching and other instructions that change the program counter

- In each cycle, it must be ensured that all data paths are different to avoid structural hazards
- Registers allow simultaneous read and write in the same clock cycle (**Recall**: Register-register/register-memory computers allow pipelining designs for that reason!)
- The ALU can only execute one operation each cycle

**Note**: To ensure different data paths when we access memory for instructions (IF) and data (MEM) is exactly the reason why we must use two L1 caches on a pipelined processor (one for instructions and one for data). Otherwise the two yellow blocks would conflict!
Instruction Level Parallelism

Pipelining enables Instruction Level Parallelism (ILP)

• On the hardware level, ILP is invisible to us (pipelined processor)
  ‣ Hardware speculation
• The compiler tries its best to increase ILP
  ‣ Basic pipeline scheduling
  ‣ Speculation, branch prediction
• On the software level, there are techniques to increase ILP with the goal in mind to decrease data and control hazards in the pipeline
  ‣ Loop-level parallelism: Loop unrolling
  ‣ Code fusion: Break/reduce dependency chains among instructions by increasing instruction count for small function bodies
Loop Unrolling

Consider the addition of two vectors: \( y = x + y \) \( x, y \in \mathbb{R}^n \)

```c
#define N 1024
int main(int argc, char* argv[]) {
    float* const x = new float[N];
    float* const y = new float[N];
    // initialize data
    // vector addition
    for (int i = 0; i < N; ++i) {
        y[i] = x[i] + y[i];
    }
    delete [] x;
    delete [] y;
    return 0;
}
```

No data dependence, we can unroll the loop easily

```c
#define N 1024
int main(int argc, char* argv[]) {
    float* const x = new float[N];
    float* const y = new float[N];
    // initialize data
    // vector addition, manual unroll
    for (int i = 0; i < N; i+=4) {
        y[i+0] = x[i+0] + y[i+0];
        y[i+1] = x[i+1] + y[i+1];
        y[i+2] = x[i+2] + y[i+2];
        y[i+3] = x[i+3] + y[i+3];
    }
    delete [] x;
    delete [] y;
    return 0;
}
```
Consider the addition of two vectors: \[ y = x + y \quad x, y \in \mathbb{R}^n \]

**Assembly:** `g++ -O2`

```
# define N 1024

int main(int argc, char* argv[])
{
    float* const x = new float[N];
    float* const y = new float[N];

    // initialize data

    // vector addition, manual unroll
    for (int i = 0; i < N; i+=4)
    {
        y[i+0] = x[i+0] + y[i+0];
        y[i+1] = x[i+1] + y[i+1];
        y[i+2] = x[i+2] + y[i+2];
        y[i+3] = x[i+3] + y[i+3];
    }

    delete [] x;
    delete [] y;
    return 0;
}
```

**Note:** For GCC, you can use the flag `-funroll-loops` to globally unroll loops. Alternatively, you can tag specific functions:

```
__attribute__((optimize("unroll-loops")))
void myfunc(void) { /* body */ }
```
Functions `func_A` and `func_B` perform different tasks using the same data. The function body of `func_A` is small with data dependence and the one of `func_B` is large.

Fusing `func_A` and `func_B` into one bigger function increases ILP and possibly adds other instructions that the compiler can use to generate reordering to hide the latency of the data dependent instructions in `func_A`.
OUTLINE

• Lecture 1
  • Uniform and Non-Uniform Memory Access
  • OpenMP Processor Binding and Performance
  • Amdahl’s Law and Strong Scaling Analysis

• Lecture 2
  • Instruction Set Architecture (ISA)
  • Pipelining
  • ISA Extensions and Data Alignment
  • Intel SPMD Program Compiler
Almost all of today's architectures introduce extensions to the basic ISA. These additional instructions mainly address integer and floating point arithmetic. For Intel, several extensions have evolved over time (in chronological order):

- Multimedia Extensions (MMX)
- Streaming SIMD Extensions (SSE)
- Advanced Vector Extensions (AVX)

Why are they there?

- They offer additional (data level) parallelism by using short vectors (DLP)
- Easy to realize on the hardware level
- Old Cray vector machines already operated on data vectors to perform computation in parallel (used for numerical simulation back then), which results in considerable performance boost
## Instruction Set Architecture Extensions

### Intel x86 ISA

| Vector register width | x86_32 | 386  
486  
Pentium  
| MMX | Pentium MMX  
| SSE | Pentium III  
| SSE2 | Pentium 4  
| SSE3 | Pentium 4E  
| x86_64 | Pentium 4F  
| SSE4 | Core 2 Duo  
Core i7 (Nehalem)  
| AVX | Sandy Bridge  
Haswell  
| AVX2 | Skylake  
Coffee Lake  
Cannon Lake  

### Processors

- 1985
- 2018

- 64 bit (integers)
- 128 bit
- 256 bit
- 512 bit
Extended ISA: Vector Registers

Architectures with support for a certain ISA extension have access to wider (in terms of bits) registers (vectors)

**Example SSE:** 128 bit registers (xmm):

- Cache line 512 bit:
  - Register %xmm0 128 bit:
    - 16-way
    - 8-way
    - 4-way
    - 2-way

The vector register can hold a different number of elements depending on the element type:

- 16x char (8 bit = 1 byte)
- 8x short int (16 bit = 2 byte)
- 4x int or float (32 bit = 4 byte)
- 2x long long int or double (64 bit = 8 byte)
Extended ISA: Vector Registers

How are vector registers (SIMD) useful?

• The extended ISA supplies new fully pipelined instructions that operate on vector registers (instead of scalars)

• This allows to operate on multiple data simultaneously with a single instruction (SIMD — Single Instruction Multiple Data)

• This enables data level parallelism (DLP). Up to now we were looking at thread level parallelism (TLP) using shared memory

• We need both of these forms of parallelism and ILP to get close to the roofline ceiling!
Example: Loop Vectorization

Consider the addition of two vectors:
\[ y = x + y \quad x, y \in \mathbb{R}^n \]

```c
#include <stdlib.h> // posix_memalign
#include <xmmintrin.h> // vector instructions
#define N 1024
int main(int argc, char* argv[])
{
    float *x, *y;
   就够usalign((void**)&x, 16, N*sizeof(float));
    posix_memalign((void**)&y, 16, N*sizeof(float));
    // initialize data
    for (int i = 0; i < N; ++i)
        y[i] = x[i] + y[i];
    delete [] x;
    delete [] y;
    return 0;
}
```

**Note:** The Intel intrinsics guide is a helpful resource for instruction look-up. It also lists associated latencies. [https://software.intel.com/sites/landingpage/IntrinsicsGuide/](https://software.intel.com/sites/landingpage/IntrinsicsGuide/)
Consider the addition of two vectors:
\[ y = x + y \quad x, y \in \mathbb{R}^n \]

Assembly: `g++ -02`
```assembly
1  movss (%rbx,%rdx), %xmm0
2  addss 0(%rbp,%rdx), %xmm0
3  movss %xmm0, (%rbx,%rdx)
4  addq $4,%rdx
5  cmpq $4096, %rdx
```

Scalar add

Vector add

Next address

4 byte offset

Next address

16 byte offset

4x faster!

SSE

Assembly: `g++ -O2 -msse`
```assembly
1  movaps (%rbx,%rdx), %xmm0
2  addps 0(%rbp,%rdx), %xmm0
3  movaps %xmm0, (%rbx,%rdx)
4  addq $16,%rdx
5  cmpq $4096, %rdx
```

Need to increment one register width!

Note: The Intel intrinsics guide is a helpful resource for instruction look-up. It also lists associated latencies.

Consider the addition of two vectors: \( y = x + y \) \( x, y \in \mathbb{R}^n \)

```c
#include <stdlib.h>    // posix_memalign
#include <xmmintrin.h> // vector instructions
#define N 1024
int main(int argc, char* argv[])
{
    float *x, *y;
    // posix_memalign: returns 16byte aligned addresses
    posix_memalign((void**)&x, 16, N*sizeof(float));
    posix_memalign((void**)&y, 16, N*sizeof(float));
    // initialize data
    // vector addition
    const int simd_width = 16/sizeof(float);
    for (int i = 0; i < N; i += simd_width)
    {
        const __m128 x4 = _mm_load_ps(x+i);
        const __m128 y4 = _mm_load_ps(y+i);
        _mm_store_ps(y+i, _mm_add_ps(x4, y4));
    }
    free(x);
    free(y);
    return 0;
}
```

**Two observations:**
1. Memory alignment is important
2. We have to take care of \textbf{loads and stores}

Efficient loads and stores require that memory is \textit{aligned} at register boundaries!
Example: Loop Vectorization

Consider the addition of two vectors: \( y = x + y \), \( x, y \in \mathbb{R}^n \)

Updated slide:
Use `posix_memalign` to allocate aligned memory instead of C++11 `alignas(16)` specifier

Two observations:
1. Memory alignment is important
2. We have to take care of loads and stores

Efficient loads and stores require that memory is aligned at register boundaries!

Consider the addition of two vectors:
\[
\begin{align*}
y &= x + y \\
x, y &\in \mathbb{R}^n
\end{align*}
\]

Updated slide:
Use `posix_memalign` to allocate aligned memory instead of C++11 `alignas(16)` specifier

Two observations:
1. Memory alignment is important
2. We have to take care of loads and stores

Efficient loads and stores require that memory is aligned at register boundaries!

Updated slide:
Use `posix_memalign` to allocate aligned memory instead of C++11 `alignas(16)` specifier

Two observations:
1. Memory alignment is important
2. We have to take care of loads and stores

Efficient loads and stores require that memory is aligned at register boundaries!

Updated slide:
Use `posix_memalign` to allocate aligned memory instead of C++11 `alignas(16)` specifier

Two observations:
1. Memory alignment is important
2. We have to take care of loads and stores

Efficient loads and stores require that memory is aligned at register boundaries!

Updated slide:
Use `posix_memalign` to allocate aligned memory instead of C++11 `alignas(16)` specifier

Two observations:
1. Memory alignment is important
2. We have to take care of loads and stores

Efficient loads and stores require that memory is aligned at register boundaries!

Updated slide:
Use `posix_memalign` to allocate aligned memory instead of C++11 `alignas(16)` specifier

Two observations:
1. Memory alignment is important
2. We have to take care of loads and stores

Efficient loads and stores require that memory is aligned at register boundaries!

Updated slide:
Use `posix_memalign` to allocate aligned memory instead of C++11 `alignas(16)` specifier

Two observations:
1. Memory alignment is important
2. We have to take care of loads and stores

Efficient loads and stores require that memory is aligned at register boundaries!
Example: Loop Vectorization

Consider the addition of two vectors: $\mathbf{y} = \mathbf{x} + \mathbf{y}$, $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$

Understand the intrinsics syntax:

```c
#include <stdlib.h>    // posix_memalign
#include <xmmintrin.h> // vector instructions
#define N 1024
int main(int argc, char* argv[])
{
    float *x, *y;
    // posix_memalign: returns 16byte aligned addresses
    posix_memalign((void**)&x, 16, N*sizeof(float));
    posix_memalign((void**)&y, 16, N*sizeof(float));
    // initialize data
    // vector addition
    const int simd_width = 16/sizeof(float);
    for (int i = 0; i < N; i += simd_width)
    {
        const __m128 x4 = _mm_load_ps(x+i);
        const __m128 y4 = _mm_load_ps(y+i);
        _mm_store_ps(y+i, _mm_add_ps(x4, y4));
    }
    free(x);
    free(y);
    return 0;
}
```

%xmm register

Scalar

- SIMD lane
- SIMD lane
- SIMD lane
- SIMD lane

Packed

- SIMD lane
- SIMD lane
- SIMD lane
- SIMD lane

Check out the Intel intrinsics guide for more:
Extended Instruction Set Architecture

What is the best way to use the instruction set extensions?

• Performance critical kernel:
  ‣ Write the SIMD code explicitly using intrinsics (time consuming, best results)
• Rely on the compiler’s skills for auto-vectorization
  ‣ Difficult task for compiler, often not satisfying (clang does a better job here than gcc, personal experience)
  ‣ For our simple example above (vector addition), we can turn on auto-vectorization by using the `-ftree-vectorize` flag or the `-O3` optimization flag. The result is (almost) the same as the manual vectorization.
• Use a library that supports vectorized kernels (e.g. Eigen)
• Use a specific programming language/model that is designed to exploit SIMD
  ‣ Examples are: Intel ISPC, OpenCL, IVL or VecImp
OUTLINE

• Lecture 1
  • Uniform and Non-Uniform Memory Access
  • OpenMP Processor Binding and Performance
  • Amdahl’s Law and Strong Scaling Analysis

• Lecture 2
  • Instruction Set Architecture (ISA)
  • Pipelining
  • ISA Extensions and Data Alignment
  • Intel SPMD Program Compiler
• Issues with explicit vectorization
  ‣ Requires a good understanding of the particular hardware you are targeting
  ‣ Time consuming task — You need to determine whether the performance gain (if you can achieve it) is worth the time spend on optimization
  ‣ The optimized code is usually not portable to other architectures

• What is the Intel SPMD Program Compiler (ISPC):
  ‣ Programming model (extension of C with some C++ features) with focus on CPU performance gain through SIMD (you cannot rely on auto-vectorization!)
  ‣ SPMD: **Single Program Multiple Data** — When you program in ISPC, you have to put yourself into the position of one SIMD lane
  ‣ ISPC can compile optimized code for various architectures
  ‣ Time to software is shorter than for explicit vectorization

ISPC on GitHub: https://ispc.github.io/
Intel SPMD Program Compiler

**ISPC work flow:**

1. You start with an unoptimized code
2. Identify the performance critical function(s) in your code with the help of a profiler
3. Rewrite the performance critical code using ISPC
4. Compile object file(s) of the performance aware code using ISPC and link the object file(s) to your application code (2 step process)
Example: Loop Vectorization

Let's look at the vector addition again: \( y = x + y \) \( x, y \in \mathbb{R}^n \)

Our code will use the kernel we specify in this header. This is useful to test different variants.

Version 1: Base C++

```c
#include <stdlib.h> // posix_memalign
#include "vector_add.h"

#define N 1024

int main(int argc, char* argv[])
{
    float *x, *y;
    posix_memalign((void**)&x, 16, N*sizeof(float));
    posix_memalign((void**)&y, 16, N*sizeof(float));
    // initialize data

    // performance critical kernel (outsourced)
    vector_add(x, y, N);

    delete [] x;
    delete [] y;
    return 0;
}
```

Version 2: ISPC a

```c
export void vector_add(
    uniform float* const uniform x,
    uniform float* const uniform y,
    const uniform int N)
{
    for (int i = 0; i < N; ++i)
        y[i] = x[i] + y[i];
}
```

Version 3: ISPC b

```c
export void vector_add(
    uniform float* const uniform x,
    uniform float* const uniform y,
    const uniform int N)
{
    foreach (i = 0 ... N)
        y[i] = x[i] + y[i];
}
```
ISPC Execution Model

```
export void vector_add(
  uniform float* const uniform x,
  uniform float* const uniform y,
  const uniform int N)
{
  for (int i = programIndex; i < N; i+=programCount)
    y[i] = x[i] + y[i];
}
```

- Think SIMD:
  - ISPC runs multiple instances of your program (SPMD)
  - Each instance of your program maps to a SIMD lane
  - A team of SIMD lanes (program instances) is called a gang (similar to a CUDA warp on the GPU)
  - When you code using ISPC think as if you were a SIMD lane: How can branches (an if-statement, for example) influence the control flow of a gang? How is performance affected?
  - Efficiently managing branching execution flow is difficult in SIMD programming and can significantly degrade performance if not done well. ISPC offers a lot of support for this issue.

<table>
<thead>
<tr>
<th>%xmm register</th>
<th>programCount: Size of the gang</th>
<th>programIndex: Index of the program instance within the gang</th>
</tr>
</thead>
<tbody>
<tr>
<td>programCount = 4</td>
<td>Time</td>
<td></td>
</tr>
</tbody>
</table>

- Because all 4 SIMD lanes are operated on in the same instruction, how is performance affected if, say, the control flow of one lane diverges over time?
Diverging control flow within a gang (example 4-way SIMD):

```
int f(int a, int b) {
    if (a < 0)
        a = 0;
    else
        a += b;
    return a;
}
```

**ISPC guarantees maximal convergence:**

Maximal convergence means that if two program instances follow the same control path, they are guaranteed to execute each program statement concurrently. If two program instances follow diverging control paths, it is guaranteed that they will reconverge as soon as possible in the function. In the above example, the gang diverges on line 3 and is maximally converged after line 6.
Maximally Converged:
All concurrent program instances that are running before the `if` are guaranteed to run concurrently after the `else` block. This also means that the `print` function in the following code will never be called (the code hangs in the loop):
```c
1 if (programIndex == 0) {
2   while (true); // infinite loop
3 } // infinite loop
4 print("hello, world\n");
```

**ISPC guarantees maximal convergence:**
Maximal convergence means that if two program instances follow the same control path, they are guaranteed to execute each program statement concurrently. If two program instances follow diverging control paths, it is guaranteed that they will reconverge as soon as possible in the function. In the above example, the gang diverges on line 3 and is maximally converged after line 6.
ISPC Rate Specifiers

- ISPC rate (storage class) qualifiers:
  - **uniform**: The variable is shared among all program instances in a gang. The variable value is the same for all program instances, they have a uniform view of the memory.
  - **varying**: Each program instance has its own unique variable, which is allowed to take different values in each program instance (non-uniform view of memory)

- The default for unqualified variables is **varying**
- **varying** allows for more complex control paths (gang divergence), at the cost of performance
- Qualifying a variable as **uniform** helps the compiler to generate better optimized code
- A **uniform** variable is not subject to masking (all lanes follow the same control path)
- **General advise**: Whenever possible you should use uniform variables

**Special cases:**
- **Pointers**:
  - By default, if not qualified, the pointer value (its **address**) is **varying**, while the data it points to is **uniform**.
- **References**:
  - In ISPC, references **can not** be bound to **varying** lvalues (compiler error)
    
    ```
    1 uniform float * uniform uptr = // ...;
    2 float &a = *uptr; // ok
    3 uniform float * varying vptr = // ...;
    4 float &b = *vptr; // ERROR: *ptr is a varying lvalue type
    ```
**Manual vectorization**

```c
const int simd_width = 16/sizeof(float);
for (int i = 0; i < N; i += simd_width) {
    const __m128 x4 = _mm_load_ps(x+i);
    const __m128 y4 = _mm_load_ps(y+i);
    _mm_store_ps(y+i, _mm_add_ps(x4, y4));
}
```

*Note:* programCount == simd_width

**ISPC equivalent**

```c
export void vector_add(
    uniform float* const uniform x,
    uniform float* const uniform y,
    const uniform int N)
{
    for (int i = programIndex; i < N; i+=programCount)
        y[i] = x[i] + y[i];
}
```

**ISPC more general version**

```c
export void vector_add(
    uniform float* const uniform x,
    uniform float* const uniform y,
    const uniform int N)
{
    foreach (i = 0 ... N)
        y[i] = x[i] + y[i];
}
```
**ISPC Storage Class Specifiers**

### Manual vectorization

10 constant int simd_width = 16 / sizeof(float);
11 for (int i = 0; i < N; i += simd_width) {
12     const __m128 x4 = _mm_load_ps(x+i);
13     const __m128 y4 = _mm_load_ps(y+i);
14     _mm_store_ps(y+i, _mm_add_ps(x4, y4));
15 }

**Note:** programCount == simd_width

Only works for N % simd_width = 0

Works with any N. ISPC takes care of remainders. `foreach` also works with multiple dimensions, for example:

```ispc
foreach (ix = 0 ... Nx, iy = 0 ... Ny, iz = 0 ... Nz)
```

**Note:** `foreach` is similar to `#pragma omp for`.

ISPC supports more loop-constructors that we do not cover here. See the documentation: [https://ispc.github.io/documentation.html](https://ispc.github.io/documentation.html)

### ISPC equivalent

1 export void vector_add(
2     uniform float* const uniform x,
3     uniform float* const uniform y,
4     const uniform int N)
5 {
6     for (int i = programIndex; i < N; i += programCount) {
7         y[i] = x[i] + y[i];
8     }
9 }

### ISPC more general version

1 export void vector_add(
2     uniform float* const uniform x,
3     uniform float* const uniform y,
4     const uniform int N)
5 {
6     foreach (i = 0 ... N) {
7         y[i] = x[i] + y[i];
8     }
9 }

**Note:** `foreach` is similar to `#pragma omp for`.

ISPC supports more loop-constructors that we do not cover here. See the documentation: [https://ispc.github.io/documentation.html](https://ispc.github.io/documentation.html)
The efficiency to exploit data level parallelism depends on the memory layout.

**Array of structures:**

1. `struct Foo { float x, y, z; };`
2. `Foo AoS[10]; // Array of Foo structures`
3. `float x2 = AoS[2].x; // memory address 0x00`
4. `float y2 = AoS[2].y; // memory address 0x04`
5. `float z2 = AoS[2].z; // memory address 0x08`

**Structure of arrays:**

1. `struct Foo { float *x, *y, *z; };`
2. `Foo SoA; // Structure Foo of arrays`
3. `SoA.x = (float*)malloc(10*sizeof(float));`
4. `SoA.y = (float*)malloc(10*sizeof(float));`
5. `SoA.z = (float*)malloc(10*sizeof(float));`
6. `float x2 = SoA.x[2]; // memory address 0x00`
7. `float y2 = SoA.y[2]; // memory address 0x28`
8. `float z2 = SoA.z[2]; // memory address 0x50`

**Layout in memory:**

**AoS:** Coordinates x, y and z are next to each other in memory. Often not desired when vectorizing code (expensive gather instructions).

**SoA:** Coordinates x, y and z are stored in separate arrays. Desired layout for vectorization (efficient vector loads).

**Bad padding** for vectorization. Can be fixed with

```
struct Foo { float x, y, z, dummy; }
```

**Bad alignment. May happen with malloc**
Layout of Data in Memory

ISPC supports SoA layouts

Conventional AoS:

\[
x_0 \ y_0 \ z_0 \ x_1 \ y_1 \ z_1 \ x_2 \ y_2 \ z_2 \ x_3 \ y_3 \ z_3 \ x_4 \ y_4 \ z_4
\]

Gather loads are expensive

Index of ISPC program instance

Efficient vector loads

ISPC short SoA:

\[
x_0 \ x_1 \ x_2 \ y_0 \ y_1 \ y_2 \ y_3 \ z_0 \ z_1 \ z_2 \ z_3 \ x_4 \ x_5 \ x_6
\]

16 bytes apart

\[
\text{struct } Foo \{
\text{ float } x, y, z;
\};
\]

1

2

3

4

5

6

7

\[
\text{uniform } \text{Foo AoS[10];} \quad // \text{Results in gathers}
\]

\[
\text{float } x2 = \text{AoS[2].x;} \quad // \text{memory address 0x00}
\]

\[
\text{float } y2 = \text{AoS[2].y;} \quad // \text{memory address 0x04}
\]

\[
\text{float } z2 = \text{AoS[2].z;} \quad // \text{memory address 0x08}
\]

\[
\text{struct } Foo \{
\text{ float } x, y, z;
\};
\]

1

2

3

4

5

6

7

\[
\text{soa<4> Foo AoS[10];} \quad // \text{converts to ISPC short SoA}
\]

\[
\text{uniform } \text{float } x2 = \text{AoS[2].x;} \quad // \text{memory address 0x00}
\]

\[
\text{uniform } \text{float } y2 = \text{AoS[2].y;} \quad // \text{memory address 0x10}
\]

\[
\text{uniform } \text{float } z2 = \text{AoS[2].z;} \quad // \text{memory address 0x20}
\]

\[
\text{struct }
\{\text{ uniform float } x[4], y[4], z[4]; \}
\]

Converts to:

\[
\text{struct }
\{\text{ uniform float } x[4], y[4], z[4]; \}
\]
### Layout of Data in Memory

**ISPC supports SoA layouts**

**Conventional AoS:**

```plaintext
struct Foo { float x, y, z; }
uniform Foo AoS[10]; // Results in gathers
float x2 = AoS[2].x; // memory address 0x00
float y2 = AoS[2].y; // memory address 0x04
float z2 = AoS[2].z; // memory address 0x08
```

**Gather loads are expensive**

**ISPC short SoA:**

```plaintext
struct Foo { float x, y, z; }
soa<4> Foo AoS[10]; // converts to ISPC short SoA
uniform float x2 = AoS[2].x; // memory address 0x00
uniform float y2 = AoS[2].y; // memory address 0x10
uniform float z2 = AoS[2].z; // memory address 0x20
```

**Efficient vector loads**
 Compile ISPC code

How to compile with ISPC:

1. Generate ISPC object file and header:
   ‣ ispc -O2 --arch=x86-64 --target=sse2 vector_add.ispc -o vector_add.o -h vector_add.h
2. Compile application code and link to ISPC object file:
   ‣ g++ -m64 -O2 vector_add.o main.cpp -o my_ispc_app

**ISPC kernel: vector_add.ispc**

```ispc
export void vector_add(
    uniform float* const uniform x,
    uniform float* const uniform y,
    const uniform int N)
{
    foreach (i = 0 ... N)
        y[i] = x[i] + y[i];
}
```

**Application code: main.cpp**

```cpp
#include "vector_add.h"

#define N 1024

int main(int argc, char* argv[])
{
    // performance critical kernel (outsourced)
    vector_add(x, y, N);
    return 0;
}
```

Export indicates that the function should be made available to be called from application code.
Compile ISPC code

How to compile with ISPC:

1. Generate ISPC object file and header:
   - `ispc -O2 --arch=x86-64 --target=sse2 vector_add.ispc -o vector_add.o -h vector_add.h`

2. Compile application code and link to ISPC object file:
   - `g++ -m64 -O2 vector_add.o main.cpp -o my_ispc_app`

---

**ISPC kernel: vector_add.ispc**

```ispc
1 export void vector_add( 
2    uniform float* const uniform x, 
3    uniform float* const uniform y, 
4    const uniform int N) 
5 {
6    foreach (i = 0 ... N) 
7    y[i] = x[i] + y[i]; 
8 }
```

**Application code: main.cpp**

```cpp
1 #include "vector_add.h" 
2 3 #define N 1024 
4 int main(int argc, char* argv[]) 
5 {
10   // performance critical kernel (outsourced)
11   vector_add(x, y, N); 
15   return 0; 
16 }
```
ISPC Tips

• Try to use uniform whenever possible
• Try to inline functions aggressively
• Declare variables in the scope where they are used
• Try to avoid 64bit addressing calculations

Useful links:

Home:  https://ispc.github.io/index.html
https://ispc.github.io/documentation.html
https://ispc.github.io/perfguide.html