Homework 3, Due Thursday, Sept. 29 before class

• To submit your homework:
  - Submit a PDF file
  - Use the “handin” program on the CADE machines
  - Use the following command:
    “handin cs4961 hw3 <prob2file>”

Problem 1 (problem 5.8 in Wolfe, 1996):
(a) Find all the data dependence relations in the following loop, including dependence distances:

\[
\begin{align*}
\text{for } (i=2; i<N; i+=2) \\
\text{for } (j=i; j<N; j+=2) \\
\end{align*}
\]

(b) Draw the iteration space of the loop, labeling the iteration vectors, and include an edge for any loop-carried dependence relation.

Loop coalescing is a reordering transformation where a nested loop is combined into a single loop with a longer number of iterations. When coalescing two loops with a dependence relation with distance vector \((d_1, d_2)\), what is the distance vector for the dependence relation in the resulting loop?

Problem 3:
Provide pseudo-code for a locality-optimized version of the following code:

\[
\begin{align*}
\text{for } (j=0; j<N; j++) \\
\text{for } (i=0; i<N; i++) \\
A[i][j] &= b[j][i] + c[i][j]*5.0;
\end{align*}
\]

Problem 4: In words, describe how you would optimize the following code for locality using loop transformations.

\[
\begin{align*}
\text{for } (l=0; l<N; l++) \\
\text{for } (k=0; k<N; k++) \\
C[k][l] &= 0.0; \\
\text{for } (j=0; j<W; j++) \\
\text{for } (i=0; i<W; i++) \\
C[k][l] &= A[k+i][l+j]*B[i][j];
\end{align*}
\]
Programming Assignment 2:
Due Friday, Oct. 7
To be done on water.eng.utah.edu

In OpenMP, write a task parallel program that implements the following three tasks for a problem size and data set to be provided. For M different inputs, you will perform the following for each input:

- **TASK 1**: Scale the input data set by $2^{(i*j)}$
- **TASK 2**: Compute the sum of the data
- **TASK 3**: Compute the average, and update max avg if it is greater than previous value

Like last time, I've prepared a template

Report your results in a separate README file.
- What is the parallel speedup of your code? To compute parallel speedup, you will need to time the execution of both the sequential and parallel code, and report speedup = Time(seq) / Time (parallel)
- You will be graded strictly on correctness. Your code may not speed up, but we will refine this later.
- Report results for two different numbers of threads.

---

Today's Lecture

- Practical understanding of SIMD in the context of multimedia extensions, cont.
- Slide source:
  - Jaewook Shin, my former PhD student

---

Overview of SIMD Programming

- Vector architectures
  - Early examples of SIMD supercomputers
- TODAY Mostly
  - Multimedia extensions such as SSE and AltiVec
  - Graphics and games processors (CUDA, stay tuned)
  - Accelerators (e.g., Clear-Speed)
- Is there a dominant SIMD programming model
  - Unfortunately, NO!!!
- Why not?
  - Vector architectures were programmed by scientists
  - Multimedia extension architectures are programmed by systems programmers (almost assembly language!)
  - GPUs are programmed by games developers (domain-specific libraries)

Scalar vs. SIMD in Multimedia Extensions

Scalar: add r1,r2,r3

SIMD: vadd<sws> v1,v2,v3

Slide source: Sam Larsen
Multimedia Extension Architectures

- At the core of multimedia extensions
  - SIMD parallelism
  - Variable-sized data fields
  - Vector length = register width / type size

Exploiting SLP with SIMD Execution

- Benefit:
  - Multiple ALU ops → One SIMD op
  - Multiple ld/st ops → One wide mem op

- What are the overheads:
  - Packing and unpacking:
    - rearrange data so that it is contiguous
  - Alignment overhead
    - Accessing data from the memory system so that it is aligned to a "superword" boundary
    - Control flow (TODAY)
      - Control flow may require executing all paths

Packing/Unpacking Costs

- Packing source operands
  - Copying into contiguous memory

- Unpacking destination operands
  - Copying back to location

A = f()
B = g()
C = A + 2
D = B + 3
E = C / 5
F = D * 7

Alignment

- Data must be aligned to "superword" boundaries
  - That is, the address must be a multiple of the superword size S
  - Then address(data) % S = 0 should be true for best performance
  - Sometimes this is a requirement for correctness

- Assumption: Data structures statically declared or malloced have a starting address that is aligned

- Worst case, programmer or memory system manipulates data that is not aligned or may not be aligned by merging the result of two aligned loads

// load a[2:5]
V1 = a[0:3];
V2 = a[4:7];
Va = merge(V1, V2, 8);
Alignment Code Generation (cont.)

• Misaligned memory access
  - The address is always a non-zero constant offset away from the 16 byte boundaries.
  - Static alignment: For a misaligned load, issue two adjacent aligned loads followed by a merge.

```c
float a[64];
for (i=0; i<60; i+=4)
  Va = a[i:i+3];
```

• Statically align loop iterations

```c
float a[64];
for (i=0; i<60; i+=4)
  V1 = a[i:i+3];
  V2 = a[i+4:i+7];
  Va = merge(V1, V2, 8);
```

Alignment Code Generation (cont.)

• Unaligned memory access
  - The offset from 16 byte boundaries is varying or not enough information is available.
  - Dynamic alignment: The merging point is computed during run time.

```c
float a[64];
start = read();
for (i=start; i<60; i++)
  Va = a[i:i+3];
```

Summary of dealing with alignment issues

• Worst case is dynamic alignment based on address calculation (previous slide)
• Compiler (or programmer) can use analysis to prove data is already aligned
  - We know that data is initially aligned at its starting address by convention
  - If we are stepping through a loop with a constant starting point and accessing the data sequentially, then it preserves the alignment across the loop
• We can adjust computation to make it aligned by having a sequential portion until aligned, followed by a SIMD portion, possibly followed by a sequential cleanup
• Sometimes alignment overhead is so significant that there is no performance gain from SIMD execution
**Last SIMD issue: Control Flow**

- What if I have control flow?
  - Both control flow paths must be executed!

  ```
  for (i=0; i<16; i++)
  if (a[i] != 0)
     b[i]++;
  else
     b[i]++;
  ```

  **What happens:**
  - Compute `a[i] != 0` for all fields
  - Copy `b[i]++` for all fields in temporary `t1`
  - Merge `t1` and `t2` according to value of `a[i] != 0`

**SIMD Example**

```c
for (i=0; i<16; i++)
  if (a[i] != 0)
    b[i]++;

for (i=0; i<16; i+=4)
  pred = a[i:i+3] != (0, 0, 0, 0);
  old = b[i:i+3];
  new = old + (1, 1, 1, 1);
  b[i:i+3] = SELECT(old, new, pred);
```

**What happens:**
- Compute `a[i] != 0` for all fields
- Compute `b[i] = b[i]/a[i]` in register `t1`
- Compare `b[i]++` for all fields in `t2`
- Merge `t1` and `t2` according to value of `a[i] != 0`

**Can we improve on this?**

- Suppose that all control flow paths are not executed with the same frequency
- Code could be optimized for the one with the highest frequency
- The other would execute with the default general behavior

**What if `a` is usually 0:**
- Compute `a[i] != 0` for all fields
- Use general code when some fields are nonzero
- Otherwise do nothing

**An Optimization:**

**Branch-On-Superword-Condition-Code**

```c
for (i=0; i<16; i+=4)
  pred = a[i:i+3] != (0, 0, 0, 0);
  branch-on-none (pred) L1;
  old = b[i:i+3];
  new = old + (1, 1, 1, 1);
  b[i:i+3] = SELECT(old, new, pred);
L1: }
```

**Overhead:**
Both control flow paths are always executed!
**Control Flow**

- Not likely to be supported in today’s commercial compilers
  - Increases complexity of compiler
  - Potential for slowdown
  - Performance is dependent on input data
- Many are of the opinion that SIMD is not a good programming model when there is control flow.
- But speedups are possible!

**Nuts and Bolts**

- What does a piece of code really look like?

```plaintext
for (i=0; i<100; i+=4)
```

```plaintext
for (i=0; i<100; i+=4) {
    __m128 btmp = _mm_load_ps(float B[i]);
    __m128 ctmp = _mm_load_ps(float C[i]);
    __m128 atmp = _mm_add_ps(__m128 btmp, __m128 ctmp);
    void_mm_store_ps(float A[i], __m128 atmp);
}
```

**Wouldn’t you rather use a compiler?**

- Intel compiler is pretty good
  - icc -msse3 -vecreport3 -<file.c>
- Get feedback on why loops were not "vectorized"
- Water has its own SIMD extension,