Transformations (cont.):
Exploiting Parallelism

Last time

- Loop transformations: merging, unrolling, interchange, etc.
  - Change the order in which the iteration space is traversed.
  - Can expose parallelism, increase available ILP, or improve memory behavior.
  - Dependence testing is required to check validity of transformation.

- Today:
  - Dependence analysis
  - Exploiting vector instructions – single-instruction, multiple data (SIMD) parallelism
Vector processors

- Scalar processor: operates on single data items
- Vector processor: operates on one-dimensional arrays of data called vectors

The ever-increasing vector width

- 1996: MMX (64 bits)
- 1999: SSE (128 bits)
  - Instructions only for four 32-bit single-precision floating point
- 2001: SSE2 (128 bits)
- 2004: SSE3 (128 bits)
- 2006: SSE4 (128 bits)
- 2011: AVX, starting with “Sandy Bridge” and a vector width of 256 bits
- 2011: Xeon Phi with custom vector instructions
- operating on 512-bit vectors
- 2013: AVX2 with Fused Multiplication and Addition instructions (FMA)
- Upcoming: AVX-512 on Intel “Skylake”
- microarchitecture and an increased vector width of 512 bits on x86-64
Some downsides

- Increases in vector size not always correspond to equivalent increases in efficiency
  - Not all operations parallelized in hardware
    - AVX division has around twice the latency as an SSE division...

- This increase cannot go on forever..
  - Likely to have a stop at 1024 bits – memory bandwidth becoming a limiting factor

- Autovectorization has been pursued since the 80s

Candidates for vectorization

- Not all codes suitable for vectorization, but a lot are

- Example of a non-vectorizable loop:

```c
fib[0] = 0;
fib[1] = 1;
for(i = 2; i < N; i++) {
    fib[i] = fib[i-1] + fib[i-2];
}
```

Dependence among loop iterations
Data dependences

- Data dependences: defined on memory locations / registers and not values
- Statement/instruction **b** data depends on statement/instruction **a** if there exists:
  - true or flow dependence
    - **a** writes a location/register that **b** later reads (read-after-write -- RAW conflict)
  - anti dependence
    - **a** reads location/register that **b** later writes (write-after-read -- WAR conflict)
  - output dependence
    - **a** writes location/register that **b** later writes (write-after-read -- WAW conflict)

<table>
<thead>
<tr>
<th></th>
<th>true</th>
<th>anti</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>a</strong> =</td>
<td>= a</td>
<td>a =</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a =</td>
<td>a =</td>
<td></td>
</tr>
</tbody>
</table>

Data/control dependences

- A statement **S1** control depends on statement **S2** iff
  - **S2** computes a conditional branch and
  - the execution of **S1** depends on this branch (in one case **S1** will be executed, but not in the other).

```
S1. if (a == b)
S2. a = a + b;
S3. b = a + b;
```

- Data and control dependences define **order constraints** that need to be respected in order to generate correct code.

**Fundamental Theorem of Dependence**
Any reordering transformation that preserves every dependence in a program preserves the meaning of that program.
Parallelization vs Vectorization

- Goal: find out whether a loop can be parallelized or vectorized based on data dependence analysis (assume that body of loop is a basic block)

```c
float a[100];
for (int i=3; i < 99; i++)
    a[i] = a[i+1] + 1;
```

Iteration space

- anti-dependences with dependence distance=1

```
3 4 5 . . . 98 99
```

- Loop cannot be parallelized since no order constraints among loop iterations => anti-dep. may be violated!

```
for (int i=3; i < 99; i++)
    a[i] = a[i+1] + 1;
```

- Loop can be vectorized because RHS is read before LHS is written => anti-dep. is preserved!

```
```

---

Parallelization vs Vectorization

- Goal: find out whether a loop can be parallelized or vectorized based on data dependence analysis (assume that body of loop is a basic block)

```c
float a[100];
for (int i=3; i < 99; i++)
    a[i] = a[i] + 1;
```

Iteration space

- only intra-statement antidependences, which are preserved by statement execution semantics

```
3 4 5 . . . 98 99
```

```
for (int i=3; i < 99; i++)
    a[i] = a[i] + 1;
```

- Loop can be vectorized because RHS is read before LHS is written => anti-dep. is preserved!

```
```
Parallelization vs Vectorization

- Goal: find out whether a loop can be parallelized or vectorized based on data dependence analysis (assume that body of loop is a basic block)

```c
float a[100];
for (int i=3; i < 99; i++)
a[i] = a[i-1] + 1;
```

<table>
<thead>
<tr>
<th>O</th>
<th>O</th>
<th>O</th>
<th></th>
<th>O</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>...</td>
<td>98</td>
<td>99</td>
</tr>
</tbody>
</table>

Iteration space

true dependences with dependence distance=1

```c
for (int i=3; i < 99; i++)
a[i] = a[i-1] + 1;
```

Loop cannot be parallelized since each array assignment depends on previous array assignment => inherently sequential execution.

Loop cannot be vectorized.

Data dependence general problem

- Perfectly nested loop; **A is an m-dimensional array**

```c
for (i_1 = L_1; i_1 < U_1; i_1++) {
    for (i_2 = L_2; i_2 < U_2; i_2++) {
        ...
        for (i_n = L_n; i_n < U_n; i_n++) {
            S_1 = A(f_1(i_1, ..., i_n), ..., f_m(i_1, ..., i_n)) = ...
            S_2 = A(g_1(i_1, ..., i_n), ..., g_m(i_1, ..., i_n))
        }
        ...
    }
}
```

\[ f_i(\alpha) = g_i(\beta) \] for all \( i, 1 \leq i \leq m \)

\( \alpha < \beta \)

- Finding integer solutions for this set of simultaneous equations with constraints is an integer programming problem, which is NP-complete.
Data dependence – A simpler problem

- Our assumptions
  - Singly nested loop
  - Single assignment statement
  - All arrays are one-dimensional
  - No aliasing among arrays
  - \( f(i) \) is an affine function of the induction variable \( i: a \times i + c \), with positive integer \( a > 0 \), and \( c \) is an integer
    - Anti and output dependences don’t matter here for vectorization

- Goal of dependence testing:
  1. Prove that true dependence does not exist;
  2. If cannot do that, show that dependence exists with a fixed distance vector
  3. If cannot do that, assume existence of dependence

- Dependence testing algorithm:
  - For each pair of LHS and RHS array references \( <X(f(i)), Y(g(i))> \), perform dependence test.

```plaintext
for (int i=lb; i < ub; i++)
  X(f(i)) = ... Y(g(i))...
```

Input: \( <X(f(i)), Y(g(i))> \)
Output: no dependence, dependence with distance vector \( d \), or dependence

Method: (cascading tests)
if \( X \neq Y \) then report “no dependence” else
  if \( f(i) = i + c \) and \( g(i) \) is a constant (or vice versa) then
    apply simple zero index variable (ZIV) test
  else if \( f(i) = a \times i + c_1 \) and \( g(i) = a \times i + c_2 \) then
    apply strong single index variable (SIV) test
  else
    report “dependence”
Simple Zero Induction Variable (ZIV) test

For (int i=1; i < 100; i++)
    A(e1) = A(e2) + B(j);

- e1, e2 are constants or loop invariant symbols
- if (e1 – e2) ≠ 0 no dependence exists

Examples
- <A(i),A(5)> Is (i -5) ≠ 0 for all 0 < i < 100?
- <A(i+1),A(7)> Is (i + 1 – 5) ≠ 0 for all 0 < i < 100?

Note: this is a special case of the weak-zero SIV test (next)

Weak-zero SIV test

for (i_1 = L_1; i_1 < U_1; i_1++) {
    for (i_2 = L_2; i_2 < U_2; i_2++) {
        ...
        for (i_n = L_n; i_n < U_n; i_n++) {
            S_1 A(f_1(i_1,...,i_n),...,f_m(i_1,...,i_n)) = ...
            S_2 ... = A(g_1(i_1,...,i_n),...,g_m(i_1,...,i_n))
        }
    ...
}

- Weak-zero SIV test when
  - f(...) = a × i_k + c_1 and g(...) = c_2
- Plug in α and solve for dependence
  - α = (c_1 - c_2)/a
- A dependence exists from S_1 to S_2 if:
  - α is an integer
  - L_k ≤ α ≤ U_k
Strong Single Induction Variable (SIV) test

- Strong SIV test when
  - $f(...)$ = $a \times i + c_1$ and $g(...)$ = $a \times i + c_2$
- Plug in $\alpha$, $\beta$ and solve for dependence
  - $d = \beta - \alpha = (c_1 - c_2)/a$
- A dependence exists from $S_1$ to $S_2$ if:
  - $d$ is an integer
  - $d \leq U_k - L_k$

Strong SIV test examples

- $f(i) = a \times i + c_1$ and $g(i) = a \times i + c_2$
- Examples:
  - $<A[i], A[i-1]>$, $LB = 1$, $UB = 10$
    - $d = (0 - (-1))/1 = 1$ (an integer), $d \leq 10 - 1$
  - $<A[4*i + 2], A[4*i-1]>$, $LB = 1$, $UB = 100$
    - $d = (2 - (-1)) / 4 = \frac{3}{4}$ (not an integer)
Summary: Loop dependence analysis

- To determine whether sequential code is parallelizable (MIMD or SIMD), we require memory location dependence information
- Enabling transformations
  - Constant propagation
  - Padding of arrays/structs for alignment
  - Loop transformations (unroll/jam, bump, extend, skew)
  - Common subexpression elimination
- In many cases, inexact analysis (usually alias) => compiler does not generate vectorized code from scalar implementations
  - Some compilers allow developers to “help” by providing pragma directives

Vectorization

```c
for (int i=1; i < 100; i++)
    a[i] = a[i] * b[i] + 2 * c[i];
    d[i] = b[i] - e[i-1];
```

- Assuming no true dependences, the code can be vectorized automatically
- How do we figure out what vector instructions to use in what order?
  - an NP-hard problem
- Many subtle things to take into account:
  - data layout, alignment, dependencies among loop iterations, pointer aliasing...
Barriers to vectorization and workarounds

- Pointer aliasing
- Memory misalignment

Pointer aliasing

- C99: restrict keyword
  - A promise to the compiler that memory accesses through a pointer will not alias memory accesses from any other pointer
  - Not in C++ standard but compiler specific extensions exist
    - __restrict__ in GCC
  - Compiler is then free to apply further optimizations that would be unsafe under the assumption of pointer aliasing
  - If there actually is aliasing when we promise there’s not.. undefined behavior
  - Never lie to the compiler!
Alignment

- Example:
  - when the computer's word size is 4 bytes (a byte means 8 bits on most machines, but could be different on some systems), the data to be read should be at a memory offset which is some multiple of 4

- Two categories of load/store vector instructions -- aligned and unaligned
  - An unaligned instruction can be many times slower than its aligned equivalent – especially if it spans over two cache lines..
  - If striving for performance, we need to use aligned data structures AND make sure the compiler knows they’re aligned

Alignment (cont.)

- Many options
- Aligning on the stack
  - New C++11 feature: alignas specifier with which you can declare the (minimum) desired alignment
- Example:
  - alignas(32) int array[N];
- Guarantees that array % 32 == 0
- Can also be used to align data members inside an object
Alignment (cont.)

```cpp
class A {
public:
    int N;
    alignas(32) int array[ .. ];
}
```

- Other compiler-specific extensions for the same purpose: `declspec(align(32))` and `attribute ((aligned(32)))`
- Aligning on the heap
  - Many `malloc` functions that take the alignment as an argument: `_mm_malloc`, `_aligned_malloc`.
  - `std::align`: Over-allocate – then pass a pointer and a size, will truncate and give an aligned pointer inside that container

Alignment (cont)

- After allocating the raw memory, possible to create objects on it with placement new
  ```cpp
  void *ptr = _mm_malloc(size, alignment); MyAlignedObject *obj = new (ptr) MyAlignedObject();
  ```
- Or override operators `new` and `delete` of the target object:
  ```cpp
  void * operator new(std::size_t sz) {
    void *aligned_buffer=_mm_malloc( sizeof(*this), 32 ); return ::operator new(sz, aligned_buffer);
  }
  void operator delete(void * ptr) {
    _mm_free(ptr);
  }
  ```
Alignment (cont.)

- Still need to tell the to-be-vectorized code that the data structures are aligned!
  - Just declaring them as aligned during allocation/construction is not enough..
  - If the data structures are defined in a different compilation unit than the one we hope to auto-vectorize, the compiler cannot know they’re aligned
  - An aligned instruction operating on unaligned data will crash our program.. again, the compiler has to be conservative
  - `__assume_aligned(ptr, 32)` will do the trick
    - (or `__builtin_assume_aligned ...`)

- BEWARE – non-portable code!

Alignment (cont.)

- How much alignment should there be?
  - Equal to the size of the vector registers
  - 16 bytes on 128-bit SSE
  - 32 bytes on 256-bit AVX / AVX2
  - 64 bytes on 512-bit Xeon Phi / AVX-512
- Easy to get wrong – if you’re seg. faulting, this is the first thing to check
- Never hardcode the alignment! Use a `#define`
Enabling better auto-vectorization

```c
int add(float *a, float *b, float *restrict c, int N) {
    builtin_assume_aligned(a, 32);
    builtin_assume_aligned(b, 32);
    builtin_assume_aligned(c, 32);
    int i;
    for(i = 0; i < N; i++) {
        c[i] = a[i] + b[i];
    }
}
```

Array size

- If size of the array not multiple of vector size, there are leftover elements to be processed sequentially
  - Example: \( N = 10 \), vector size of 8. Two more elements to process sequentially
- If we know array size is a multiple of the vector size, we can tell the compiler:
  ```c
  __assume(N%8==0); (ICC only!)
  ```
- Now compiler doesn’t need to generate code to handle leftover elements
for(i = 0; i < N; i++) {
    a[i].x = k * a[i].x
}

Data layout (cont.)

- Gather and scatter instructions supported by the hardware – slower, though
  - Processor exchanges memory using entire cache lines – would need to load/store whole array, even if not using all
    - More memory bandwidth, more operations
    - ... and crucially, more cache misses!
    - If having a cache miss on each access, vectorization only shortens the time between cache misses.
  - Diminished benefits from vectorization
Branches

- As mentioned last time, can be problematic, but not fatal
  - Masks: Essentially an array of booleans, one value for each element in a vector register
  - Instructions that operate only on the elements whose corresponding bit in the mask is on
  - Example: Loop can still vectorize, despite the branch

```c
for(i = 0; i < N; i++) {
    if(a[i] >= 0)
        a[i] = sqrt(a[i]);
}
```

Memory bandwidth

- The faster you process the data, the more often you need to fetch more data
  - Example: without vectorization, a cache line worth of data might take 200 cycles to process; with vectorization – only 70
- Impact of cache misses more visible
  - If your loop generates many cache misses, vectorization will just shorten the time from one cache miss to another
- Prefer to do as many calculations as possible on a set of data
  - Doing 10-20 operations with some data much better than doing 2-3 before writing back to memory
    - c[i] = a[i]*a[i] + b[i]*b[i] + b[i]*a[i] + b[i]/a[i] would likely benefit more than c[i] = a[i] + b[i]

- Memory bandwidth becoming the limiting factor
Compiler-specific extensions: ICC

- ICC (Intel’s compiler)
- #pragma ivdep
  - Put before a loop – tells the compiler to ignore assumed vector dependencies
    for (int i = 0; i < m; i++)
    a[i] = a[i + k] * c;
  - Example: Vectorization safe only if k >= number of elements of a that can fit in a vector register

    #pragma ivdep
    for (int i = 0; i < m; i++)
    a[i] = a[i + k] * c;
  - “#pragma ivdep” => If unsure about a dependency, assume it doesn’t happen

Compiler-specific extensions: ICC

- #pragma vector
  - Extra directives to control vectorization
  - Possible arguments:
    - aligned/unaligned: Use aligned/unaligned instructions, ignoring what you know about the alignment of the data structures involved
    - temporal/nontemporal: Controls use of streaming stores
    - always: “Even if you don’t think vectorization will help performance, do it anyway”
    - assert: If vectorization not possible, throw compiler error
## Compiler-specific extensions: ICC

- **#pragma simd**
  - The nuclear option – enforce vectorization if at all possible
  - Now in GCC 4.9 as part of Cilk+
  - Lots of knobs and options.. Too many to describe here
    - vectorlength: Specify safe length of vector registers (in number of elements) – useful to specify there’s a dependency between i-th and i-5-th elements
    - reduction

```c
int x = 0;
#pragma simd reduction(+:x)
for(int i = 0; i < N; i++) {
    x += a[i];
}
```

## Other options

- Source transformation tools
- Libraries
- ...
Summary

- Vectorization is hard
  - requires accurate dependence analysis
  - may be automated in some cases

- Programmers can provide extra information to allow tools/compilers to generate vectorized code