Autovectorisation

L25: Modern Compiler Design

SIMD

- Single Instruction, Multiple Data
- Single Register Multiple Data
- 2-8 values are loaded at once, operated on, stored.
- Operations must be grouped
- Modern SIMD units support scatter-gather, but slower than contiguous data

Characteristics of Modern Vector Units

- Multiple pipelines for different kinds of operation
- Independent operations dispatched in parallel
- Usually one (or more) instruction per pipeline dispatched per cycle
- Multi-cycle (2-20) latency before results are available

Explicit Language Support

- Fortran, APL, GNU C and OpenCL C provide vector types
- Compiles to scalar operations or vector operations if available
- Lots of work for the programmer

Autovectorisation

- Take scalar source code
- ???
- ~~Profit!~~
- Run high-performance vector code

Aside: Vector Types in LLVM

- LLVM IR supports arbitrary-sized vectors
- All scalar arithmetic operations are defined for vectors
- Type legalisation (before code generation) splits them into smaller vectors for the target
- Autovectorisation algorithms can be target independent, converting scalar IR into vector IR
- Target-specific cost model is important for deciding which transforms make sense

Prerequisites for Vectorisation:

Example:

```
a = b+c;
d = e+f;
```

- Can this be vectorised?

Prerequisites for Vectorisation: Alias Analysis

Example:

```
a = b+c;
d = e+f;
```

- Can this be vectorised?
- Only if `a` doesn't alias `e` or `f` (e.g. C++ `int &a = e`)
- `restrict` keyword is helpful in this context
- Why might the resulting code be slower?

Prerequisites for Vectorisation: Alignment

- Many vector units depend on vectors having natural alignment for loads and stores
- Unaligned loads and stores can be done by loading as scalar and copying to vector register
- Alternatively by two vector loads and a permute
- This is very slow
- For on-stack allocations, we can modify the alignment
- For loops, we can special-case the unaligned first / last elements

Pattern-Based Loop Vectorisation

- Recognise common loop patterns
- Transform to vector equivalents
- Used by GCC, XLC
- Works well for specific cases that match patterns
- Not general - no benefit for near misses (pattern must match exactly)

Example Loop Pattern

```
for (int i=0 ; i<x; i++)
    a[i] = b[i] + c[i];
```

Transforms to:

```
int i=0;
while (insufficiently_aligned(&a[i]))
    a[i] = b[i] + c[i];
for (; i+4<x; i+=4)
    vector4_add(&a[i], &b[i], &c[i]);
for (; i<x; i++)
    a[i] = b[i] + c[i];
```

General Loop Vectorisation

- Unroll the loop (a multiple of n times for n-way vectors)
- Perform *if conversion* to eliminate branches
- Canonicalise induction variables / pointers
- Vectorise instructions within the resulting basic block
- Re-roll the loop

Superword Level Parallelism (SLP)

- Identify pairs / tuples of the same instruction
- Combine into vector operations
- Inspect operands, try to perform the same combination
- Bottom-up, works across basic blocks

Basic block vectorisation

- Inspect pairs of instructions
- Identify pairs of the same instruction
- Discard pairs where the result can not be used by another pair
- Build a tree of possible pairs with dependencies
- Prune the tree so there is only one possible pairing for each dependency
- Lots of heuristics!

Loop Nest Optimisation (LNO)

- Generic family of optimisations
- Transform nested loops into canonical forms
- Expose many future optimisation opportunities
- Most autovectorisation works on loops and depends on loops being in a comprehensible form
- Heuristic: 90% of all program execution is spent in relatively tight loops

Canonicise induction variables

- Canonical form for induction loops (‘for loops’) has value that is incremented on each iteration
- Transform loop induction variables such that:
  - Induction variable starts at 0
  - Is incremented by 1 each iteration
- Followed by Loop Strength Reduction
  - Turns all array accesses into GEPs on array base for first iteration and loop increment

Aside: Do-Loop Transform

- Some targets (especially DSPs) have very simple loop branch predictors or ‘zero cost’ loops
- Loop induction variable should count down to 0, decrementing by 1 each time
- Loop branch always predicted taken when induction variable is non-zero
- Loop branch always predicted not-taken when induction variable is zero
- No branch predictor misses for loop in this form

Loop Invariant Code Motion (LICM)

- Hoist values that don't depend on any ϕ nodes inside the loop to the start
- Avoids redundant computations within loop
- Reduces the amount of code that loop optimisations need to look at

Before:

```
for (i=0 ; i<j ; i++){
  x = a + b;
  bar(y[i] + x);
}
```

After:

```
x = a + b;
for (i=0 ; i<j ; i++){
  bar(y[i] + x);
}
```

Loop Unswitching

- Transform loops containing conditionals into conditionals containing loops
- Dramatically reduces number of conditional branches executed
- Exposes parallelism between iterations more cleanly
- Dual of LICM

Before:

```
for (i=0 ; i<j ; i++){
  if (x)
    foo(y[i]);
  else
    bar(y[i]);
}
```

After:

```
if (x) {
  for (i=0 ; i<j ; i++)
    foo(y[i]);
} else {
  for (i=0 ; i<j ; i++)
    bar(y[i]);
}
```

Loop Unrolling

- Expands loops to be a smaller number of loops with multiple copies of the body
- Less useful when loop branch predictors are competent
- Increases instruction cache usage
- ...but exposes more optimisation opportunities

Before:

```
for (i=0 ; i<32 ; i++)
  bar(y[i]);
```

After:

```
for (i=0 ; i<32 ; i++){
  bar(y[i++]);
  bar(y[i++]);
  bar(y[i++]);
  bar(y[i]);
}
```

Polyhedral Optimisation (Polytope Model)

- Create dependency graph of array elements for array iterations
- Perform affine transform on graph
- Rewrite loop

Polyhedral Example

Dithering:

```
for (int j = 0; j < h; ++j) {
    for (int i = 0; i < w; ++i) {
        int v = src[i][j];
        v -= (dst[i-1][j] - src[i-1][j]) / 2;
        v -= (dst[i][j-1] - src[i][j-1]) / 4;
        v -= (dst[i+1][j-1] - src[i+1][j-1]) / 2;
        dst[i][j] = (v < 128) ? 0 : 255;
        src[i][j] = (v < 0) ? 0 : (v < 255) ? v :
            255;
    }
}
```

Loop Data Dependencies

Each iteration reads:

```
src[i][j]
dst[i-1][j], src[i-1][j]
dst[i][j-1], src[i][j-1]
dst[i+1][j-1], src[i+1][j-1];
```

Each iteration writes:

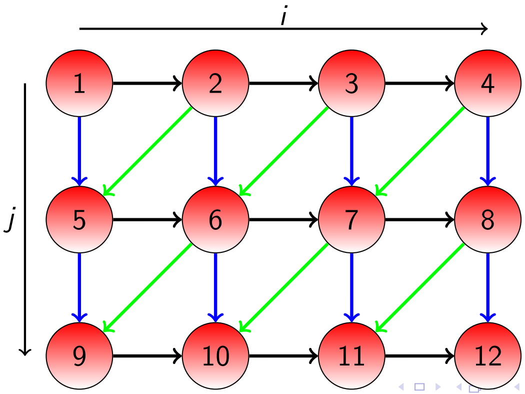
```
dst[i][j]
src[i][j]
```

Loop Iteration Dependencies

Each iteration depends on the results from:

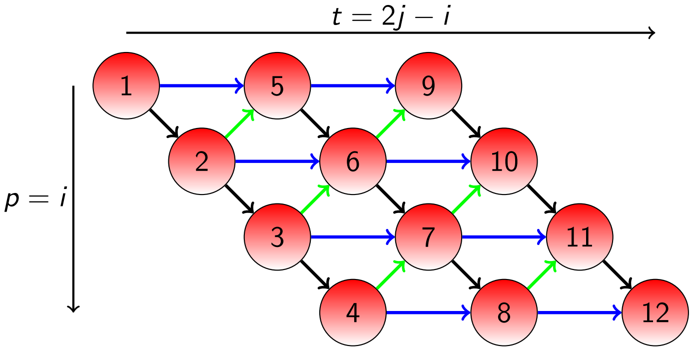
```
(i-1,j)
(i,j-1)
(i+1,j-1)
```

As a polyhedron:



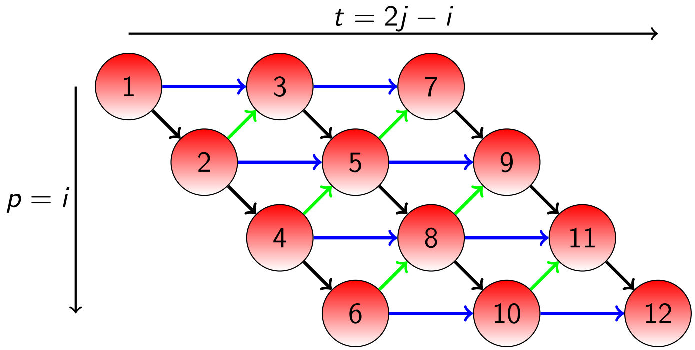
Applying an Affine Transform

- Affine transforms are matrices that change coordinate spaces
- Can skew, rotate, scale (not relevant in this context)
- Skew and rotate applied here to the dependencies:
- $(p, t) = (i, 2j + i)$



Changing the Execution Order

- t becomes the outer loop
- p becomes the inner loop

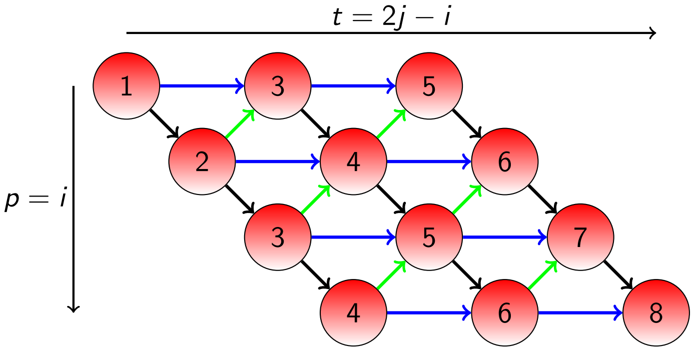


Why?

- Polyhedral transformations allow various reorderings of the loop
- Dependencies between iterations are preserved
- May expose better parallelism opportunities
- May expose better locality of reference
- Factor of $10\times$ speedup or more for some algorithms

Parallel Execution

- First and last two iterations are scalar
- All of the rest are 2-element vectors



Questions?