Compiler auto-vectorization: techniques and challenges

Amara Emerson
Senior HPC Compiler Engineer
Advanced Compiler Team
Development Solutions Group

University of Manchester
April 26th 2016

©ARM 2016
Lecture agenda

Part 1
- ARM overview
- Manchester Design Centre:
  - Who we are & what we do
- State of compiler technology
- Why work in compilers?

Part 2
- General loop optimizations
- Auto-vectorization:
  - Aims
  - Challenges
  - Techniques
Compilers in the modern software ecosystem

- Software compilers have attracted major investment & reached maturity from silicon designers

- Exploiting the maximum performance from CPU & GPU requires intelligent analysis & optimization

- Two major open source compilers dominate most platforms:
  - GNU GCC
  - Clang/LLVM
Why work in compilers?

- Compiler engineering is a close marriage of computer science theory & engineering.

- Extremely complex pieces of software, with many challenges in for various interests.

- A good foundation for work & research into other areas, requires an broad understanding of computer architecture, theory & software engineering to be effective in the industry.
Loop Optimizations

Optimizing loops is one of the most important functions of modern compilers.

Runtime performance is dominated by loop execution in many workloads, and loop analysis and transformation is one of most complex tasks in the compilation pipeline.

If we optimize loops well, then we can extract more parallelism. For many HPC workloads, this is critical for achieving performance closer to the theoretical maximum of the system.
Loop Optimizations

Many kinds of loop optimizations:

- Loop invariant code motion
- Loop unrolling
- Loop distribution/fission
- Loop interchange
- Loop tiling
- Loop peeling
- Loop vectorization
- …and more!
Loop Invariant Code Motion

- Commonly known as LICM, this is a fundamental transformation, sometimes applied multiple times throughout compilation.

- Goal is to remove code from loops which do not have a dependence on any loop variant value.

- Code removed from loops is vital not only for performance, but for other optimizations to work effectively.

- Code lifted out of the loop and placed before it is hoisted, code pushed down below a loop is sunk.
Loop Invariant Code Motion - Example

Before:

```c
int *c = ...
int foo = ...
int bar = ...
for (int i = 0; i < n; ++i) {
    a[i] = b[i] + foo * bar;
    *c = foo;
}
```

After:

```c
int foo = ...
int bar = ...
int tmp = foo * bar;
for (int i = 0; i < n; ++i) {
    a[i] = b[i] + tmp;
}
* c = foo;
```

The computation of `foo * bar` has been hoisted up out of the loop and replaced with a pre-computed value.

The loop invariant store to the address `c` has been sunk below the loop.
Loop Unrolling

- Unrolling is a technique for exposing more instruction-level parallelism by replicating loop bodies multiple times, and decreasing the loop trip count by the unroll factor.

- Exposing more independent instructions to the CPU core, superscalar execution can improve performance by utilizing multiple functional units.
  - Therefore unrolling is generally more effective on out-of-order cores.

- It amortizes the loop control flow overhead over a number of logical iterations.
Loop Distribution/Fission

- The loop distribution transformation \textit{splits a loop into multiple loops}, each executing a portion of the original loop body’s code, generally over the same iteration space.

- Useful for:
  - \textit{Improving cache behaviour}, due to accessing less memory during a single iteration of the loop
  - Isolating parts of the loop which inhibit other optimizations, e.g. loop-carried data dependences
  - Reducing register pressure by potentially lowering the number variables live across a loop iteration
Loop Distribution/Fission - Example

Before:

```c
for (int i = 0; i < n; ++i) {
    a[i] = b[i];
    c[i] = c[i-1] + x;
}
```

After:

```c
for (int i = 0; i < n; ++i) {
    a[i] = b[i];
    for (int i = 0; i < n; ++i)
        c[i] = c[i-1] + x;
}
```

The statement $S_2$ in the loop inhibits parallelization due to the loop-carried data dependence, if we distribute the original loop then we can parallelize $S_1'$ safely.
Loop Interchange

- Interchange inverts the execution of an outer loop and an inner loop.
- For performance, loop interchange can result in a more efficient memory access pattern for cache behaviour.
- It can also allow more optimizations by moving a potentially parallelization to the innermost position in the loop nest.

Before:

```c
for (int i = 0; i < n; ++i)
    for (int j = 0; j < m; ++j)
        a[j][i] = b[j][i] * K;
```

After:

```c
for (int j = 0; j < n; ++j)
    for (int i = 0; i < m; ++i)
        a[j][i] = b[j][i] * K;
```
Loop Tiling

- Also known as loop “blocking”, tiling a loop modifies its execution order through the iteration space for improving temporal locality.
- Data is re-used in the blocks, reducing cache misses as the working set of each block of computation should ideally fit within the CPU data cache.
- Requires sophisticated heuristics to choose a good tiling block size.
Loop Tiling - Example

Before:

```c
for (int i = 0; i < n; ++i) {
    // computation...
}
```

After:

```c
for (int i = 0; i < n; i+= K) {
    for (int i2 = i; i2 < min(n, i+K); ++i2) {
        // computation
    }
}
```
Loop Peeling

- Shave off iterations either before or after a scalar loop in order to align the loop iteration on some boundary.

- When the loop trip count is not statically known, runtime checks are needed in order to compute the number iterations to peel, if any.

- Some transformations, like vectorization, may require this to be done in order to guarantee data access alignment for best performance.
Loop Vectorization

- Modern high performance CPU architectures have specialized instructions to exploit data parallelism in loops.

- Commonly referred to as **SIMD** (Single-Instruction-Multiple-Data) or **vector extensions**.

- Operate on multiple elements of a wide vector register simultaneously.
  - Therefore reducing the runtime trip-count of a loop by the **vectorization factor (VF)**.

- Requires sophisticated analyses and heuristics in order to make good decisions about vectorization safety and profitability.
Loop Vectorization

- For simple loops, vectorization is essentially a simple transform that widens each operation in the loop from a scalar type to a vector type.

- Let's think of transform operating on a simple IR. We will use:
  - `dest = loadN address [, offset]` – Load N bytes from address + scaled offset
  - `storeN value, address [, offset]` – Store N bytes of value to address + scaled offset
  - `dest = dataproc_op source1, source2...` – Perform an data processing operation
  - `br condition target` – Conditional branch to a block “target”
  - `broadcast value` – Broadcast some scalar value into a vector
  - `label name` – block label “name”
Simple vectorization example

For the simple loop:

```c
for (int i = 0; i < 256; ++i)
  a[i] = b[i] * c[i] + k;
```

We will have the scalar IR:

```c
indvar = 0
label loop_header:
  bval = load4 b, indvar
  cval = load4 c, indvar
  t1 = mul bval, cval
  t2 = add t1, k
  store4 t2, a, indvar
  indvar = add indvar, 1
  br (indvar < 256), @loop_header
```
Simple vectorization example

For the simple loop:

```c
for (int i = 0; i < 256; ++i)
    a[i] = b[i] * c[i] + k;
```

We will have the scalar IR:

```c
indvar = 0
label loop_header:
    bval = load4 b, indvar
    cval = load4 c, indvar
    t1 = mul bval, cval
    t2 = add t1, k
    store4 t2, a, indvar
    indvar = add indvar, 1
    br (indvar < 256), @loop_header
```

Set up an IV (induction variable) that forms arithmetic progression throughout the loop execution.

The IV is used as the basis for iteration-dependent computations in the loop body.
Simple vectorization example

How do we vectorize this simple scalar loop? A general technique could be:
1. Analyze loop iteration behaviour, find trip count either statically or find symbolic expression to be computed at runtime.
2. Create a vector loop body and create a modified induction variable which ranges from the start value to end / VF (vector factor).
3. If loop requires peeling or iteration count doesn’t divide by VF, maintain the scalar loop in the CFG (not necessary for our loop).
4. Generate vector statements for the scalar statements in the original loop body.
Simple vectorization example

Scalar IR:

    indvar = 0
label loop_header:
    bval = load4 b, indvar
    cval = load4 c, indvar
    t1 = mul bval, cval
    t2 = add t1, k
store4 t2, a, indvar
    indvar = add indvar, 1
br (indvar < 256), @loop_header

Vectorized IR:

    indvar = 0
label loop_header:
    vec_k = broadcast k
vec_bval = vload4 b, indvar
vec_cval = vload4 c, indvar
vec_bval = vmul vec_bval, vec_cval
vec_cval = vadd vec_bval, vec_cval
vstore4 t2, a, indvar
    indvar = add indvar, 1
br (indvar < 64), @loop_header
Simple vectorization example

Vectorized IR:

\[
\begin{align*}
\text{indvar} &= 0 \\
\text{vec}_k &= \text{broadcast } k \\
\text{vstore4 } t2, a, \text{indvar} \\
\text{br } (\text{indvar} < 64), \text{loop_header}
\end{align*}
\]

Loads and stores are replaced with vector counterparts. We assume here that the induction variable offset will be scaled appropriately for the vector width.
Simple vectorization example

Vectorized IR:

```assembly
indvar = 0
vec_k = broadcast k
label loop_header:
    vec_bval = vload4 b, indvar
    vec_cval = vload4 c, indvar
    t1 = vmul vec_bval, vec_cval
    t2 = vadd t1, vec_k
    vstore4 t2, a, indvar
    indvar = add indvar, 1
    br (indvar < 64), @loop_header
```

Computations are converted to vector form. Scalars can simply be broadcasted into a vector value. LICM should hoist these out if the scalar value is loop invariant.

Arithmetic operations have a like for like replacement.
Simple vectorization example

Vectorized IR:

\[
\text{indvar} = 0 \\
\text{vec}_k = \text{broadcast } k
\]

\text{label } \text{loop}_\text{header}:

\text{vec}_b\text{val} = \text{vload4 } b, \text{indvar} \\
\text{vec}_c\text{val} = \text{vload4 } c, \text{indvar} \\
\text{t1} = \text{vmul } \text{vec}_b\text{val}, \text{vec}_c\text{val} \\
\text{t2} = \text{vadd } \text{t1}, \text{vec}_k \\
\text{vstore4 } \text{t2}, a, \text{indvar} \\
\text{indvar} = \text{add } \text{indvar}, 1 \\
\text{br } (\text{indvar} < 64), \text{@loop}_\text{header}

Loop control flow is modified to have the original tripcount \((256) / \text{VF}\) as the terminating condition.

If the trip count was not known statically, a scalar tail loop would be required to execute any \((N \text{ mod } \text{VF})\) iterations.
When can we vectorize?

Before a loop transformation can be applied, the compiler must be certain it’s legal to perform. This means:

- The runtime side-effects of the optimized code should be equivalent to the original code generated.
- There are techniques the compiler can use if the pure source code isn’t enough to guarantee safety:
  - Emitting runtime checking code to ensure a safety invariant holds.
  - Using additional programmer added pragmas to make additional assumptions about source code behaviour.
  - Speculative transformation with some fall-back mechanism.
Dependence Analysis

- Vectorization by its nature is a re-ordering of the logical execution of the original loop’s iterations.

- It's only safe to do so if data dependences between statements across iterations are preserved.
  - Note: only flow (aka true) dependences are problems in this respect.

- If a backward data dependence exists but the dependence distance is greater than the number of logical iterations being executed as a single vector iteration, then the dependence is preserved.
Dependence Analysis

for (int i = 0; i < N; ++i) {
    a[i] = a[i-1] + b[i];
}

This loop exhibits a flow dependence on a[i-1]. With a vectorization factor of 4 and a dependence distance of 1, this loop is not legal to vectorize.

for (int i = 0; i < N; ++i) {
    a[i] = a[i-6] + b[i];
}

This loop exhibits a flow dependence on a[i-6]. With a dependence distance of 6 and a vectorization factor of 4, this would be legal to vectorize.
Pointer Aliasing

Pointer aliasing is the name given to different pointers in C pointing to non-disjoint memory regions.

- Either overlapping memory regions for arrays, or pointing to the same memory.

Aliasing gives C & C++ great flexibility in the way that a programmer can use pointers and manage memory manually, but it comes at a cost for the compiler & for runtime performance.

All transformations must respect potentially legal aliasing of pointers in the original code. In practice aliasing is not used in most cases, resulting in conservative optimization decisions.
Pointer Aliasing

Modern compilers have multiple forms of alias analyses in order to disambiguate pointer values from each other. The knowledge the analyses can build up must be conservative, and comes from various sources, e.g.

- Data flow analysis of the function
- Type semantics of the source language

Lack of aliasing knowledge can prevent our loop optimizations from being legally performed. Let’s look at our C code again as a whole function...
void func(int *a,
     int *b,
     int *c,
     int k) {
    for (int i = 0; i < 256; ++i)
        a[i] = b[i] * c[i] + k;
    return;
}

If this code is all the compiler can see, then vectorizing this function with no special techniques is not legal.

Why?
Aliasing – Vectorization example

```c
void func(int *a,
          int *b,
          int *c,
          int k) {
    for (int i = 0; i < 256; ++i)
        a[i] = b[i] * c[i] + k;
    return;
}
```

If this code is all the compiler can see, then vectorizing this function with no special techniques is not legal.

Why?

The write to pointer `a` could alias with the read from `b` & `c`. Sequential execution would provide different results than a parallelized execution of this loop.
Aliasing – Vectorization example

```c
void func(int * restrict a,
            int * restrict b,
            int * restrict c,
            int k) {
    for (int i = 0; i < 256; ++i)
        a[i] = b[i] * c[i] + k;
    return;
}
```

From C99, a new keyword was added to the standard to allow programmers to tell the compiler about aliasing of pointers.

restrict can be used on pointer declarations to express that accesses to memory from the pointer will not alias.
void func(int * restrict a,
        int * restrict b,
        int * restrict c,
        int k) {
    for (int i = 0; i < 256; ++i)
        a[i] = b[i] * c[i] + k;
    return;
}

If restrict is not provided by the programmer, then it’s still possible to vectorize some loops. ..

The compiler must emit runtime alias checks of the pointers, branching to a vectorized loop if they don’t alias.
Language semantics & optimizations

- Modern, sophisticated compilers make extensive use of the C & C++ language semantics to do optimization.

- The C language specification defines a program’s behaviour with respect to certain constructs. There are however some situations where the specification either omits to define behaviour or explicitly states that some behaviour is undefined.

- “Undefined behaviour” (aka UB) in the language specification is the optimizer’s best friend!
  - Allows optimization due to extra assumptions the compiler can make.
Undefined behaviour

If, during execution, a C program exhibits undefined behaviour, anything can happen. No, really, *anything*:

- The program can terminate or crash.
- It can compute the “wrong answers”.
- It can set the computer on fire.
- It can do nothing at all.
- It can do what the programmer expects.

So if you write a program that has UB, keep a fire extinguisher near by.
UB example: signed integer overflow

Example:
The C language does not define signed integer overflow. That is, you cannot safely assume that INT_MAX + 1 wraps around to INT_MIN. How does this help the compiler?

- It assumes that such an event like wrapping cannot occur. If it does, then the program behaviour is undefined.
- Compiler does not have to generate code to produce the expected behaviour for the wrapping case. Helps loop optimizations as int’s are often used as loop counters.

However unsigned integer overflow is defined as modulo the unsigned range.
Profitability

While vectorization can provide a significant performance speedup, in some cases almost linear with increasing VF, it has costs to be considered.

- Vector loop bodies can be larger than their scalar forms. More complex operations may be needed, increasing code size.
  - Especially true if a scalar loop body must be maintained as well.
- Vector loop may have increased startup costs to prepare data for vectorized execution.
- Runtime check overhead for memory aliasing.
- Vector instructions on architectures can take more cycles.
Summary

Auto-vectorization is an important compiler optimization, part of a set of loop transformations modern compilers will employ to extract the best performance from serially written code.

Many challenges in determining the legality & profitability of loop optimizations.
- Strong dependence & alias analysis is key for performant loop vectorization.

Programming language semantics play a large role in the assumptions compilers can make, and scope of optimizations available.
Further reading

- C++ Undefined Behavior: What is it, and why should I care?”
  [link](http://accu.org/content/conf2014/MarshallClowUndefined%20Behavior-ACCU2014.pdf)
- [link](http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html)
- “Optimizing compilers for modern architectures: a dependence-based approach” – Kennedy & Allen