LLVM - Loop Optimizations

Why loops optimization & approaches

Loops often account for a **significant portion of program execution** time.

Optimising means:
- Reducing iteration overhead.
- Leveraging data-level and instruction-level parallelism.
- Improving computational performance.
- Making possible other optimisations.

There are **different approaches** to doing that.

Compiler-Based

Libraries

OpenMP

Compiler ‘s pragmas

Wide **support** in most mainstream compilers.

Easy to use, without modifying source code.

Introduced by the compilers or by the libraries.

Clang

```
#pragma unroll
#pragma clang loop unroll(enable)
#pragma unroll_and_jam
#pragma clang loop distribute(enable)
#pragma clang loop vectorize(enable)
#pragma clang loop interleave(enable)
```

gcc

```
#pragma GCC unroll
#pragma GCC ivdep
```

icc

```
#pragma parallel
#pragma offload
#pragma unroll_and_jam
#pragma nofusion
#pragma distribute_point
#pragma simd
#pragma vector
#pragma swp
#pragma ivdep
#pragma loop_count(n)
```

OpenMP

```
#pragma omp simd
#pragma omp for
#pragma omp target
```

Proposed optimisations

Clang
```
#pragma unroll
#pragma clang loop unroll(enable)
#pragma unro
#pragma clan
#pragma clan
#pragma clan
```

gcc
```
#pragma GCC unroll
#pragma GCC ivdep
```

msvc
```
#pragma loop(hint_parallel(0))
#pragma loop(no_vector)
#pragma loop(ivdep)
```

Cray
```
#pragma _CRI fusion
#pragma _CRI nofission
#pragma _CRI blockingsize
#pragma _CRI interchange
#pragma _CRI collapse
```

OpenMP
```
#pragma omp simd
#pragma omp for
#pragma nodepchk
```

xlc
```
#pragma unrollandfuse
#pragma stream_unroll
#pragma block_loop
#pragma fission
#pragma blocking size
#pragma altcode
#pragma noinvarif
#pragma mem prefetch
#pragma interchange
#pragma ivdep
```

OpenACC
```
#pragma acc kernels
```

icc
```
#pragma simd
#pragma vector
#pragma swp
#pragma ivdep
#pragma loop_count(n)
```

HP
```
#pragma UNROLL_FACTOR
#pragma IF_CONVERT
#pragma IVDEP
#pragma NODEPCHK
```

```
#pragma unroll
#pragma clang loop unroll(enable)
```

```
#pragma clang loop vectorize(enable)
```

Proposed code

The proposed code focuses on a loop with a simple task, the sum of two arrays in a third one.

For simplicity, we focus on the **pseudocode**.

```
void
sumArray(
    int* arrA, int* arrB,
    int* arrC, size_t size
) {

    for (size_t i = 0; i < size; ++i) {
        arrC[i] = arrA[i] + arrB[i];
    }
}
```

```
function sumArray:
    i = 0
    for i to size:
        C[i] = A[i] + B[i]
        i = i + 1
```

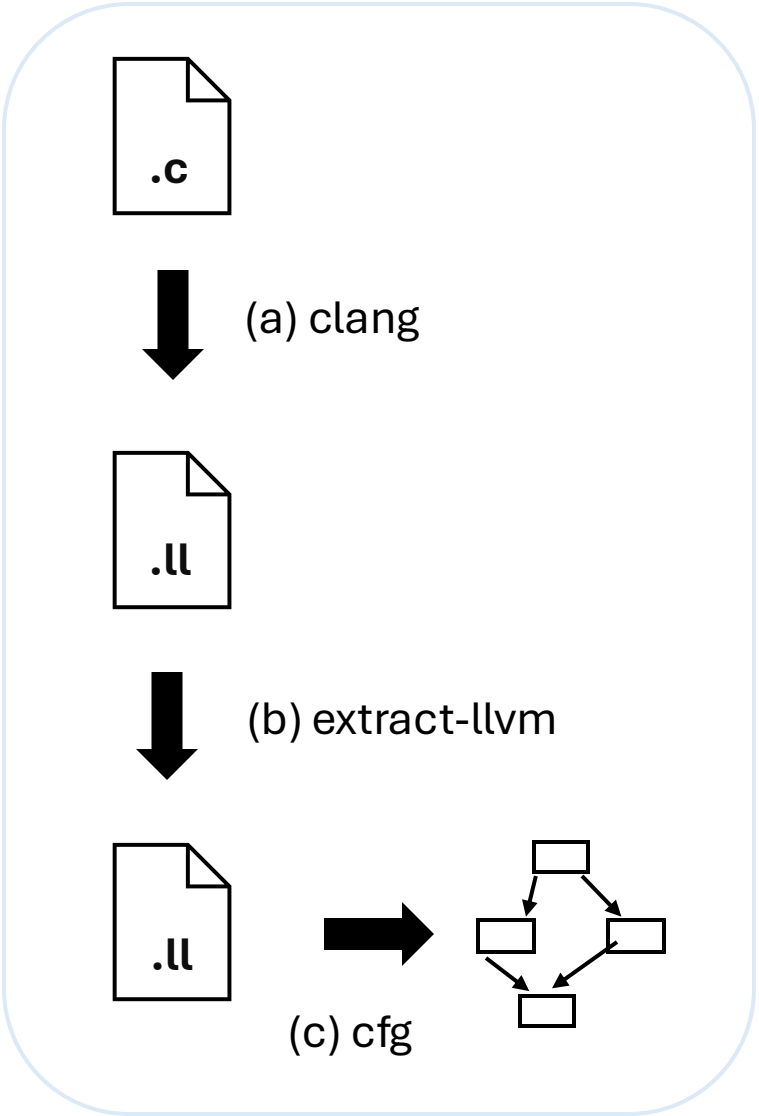
Generation of LLVM code & CFG

Use of **clang compiler** to generate **LLVM-IR** of the entire file (a).

Filter of a specific part LLVM-IR(b).

Estraction of cfg (c).

Source files written in .c.



Generation of LLVM code & CFG

```
# (a)
clang -O1 -funroll-loops -fno-discard-value-names -S -emit-llvm -o main.ll main.c
# (b)
llvm-extract -S -func='sumArray' -o sumArray.ll main.ll
opt -passes=dot-cfg sumArray.ll > /dev/null
# (c)
dot -Tpdf '.sumArray.dot' -o cfg.pdf
```

#pragma unroll N

This **pragma suggests** the compiler unroll the loop by N (4 in our case).

N must be noted at compile time.

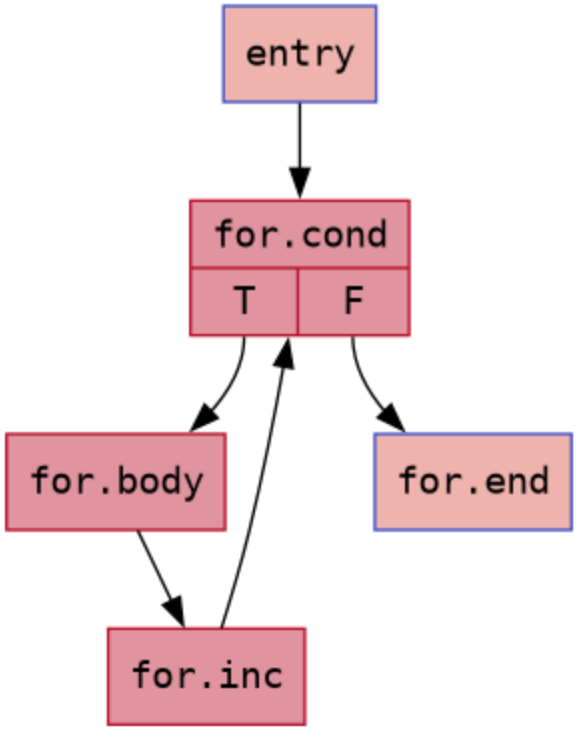
Remaining unrolled loop. (a)

```
i = 0
#pragma unroll 4
for i to size:
    C[i] = A[i] + B[i]
```

a)

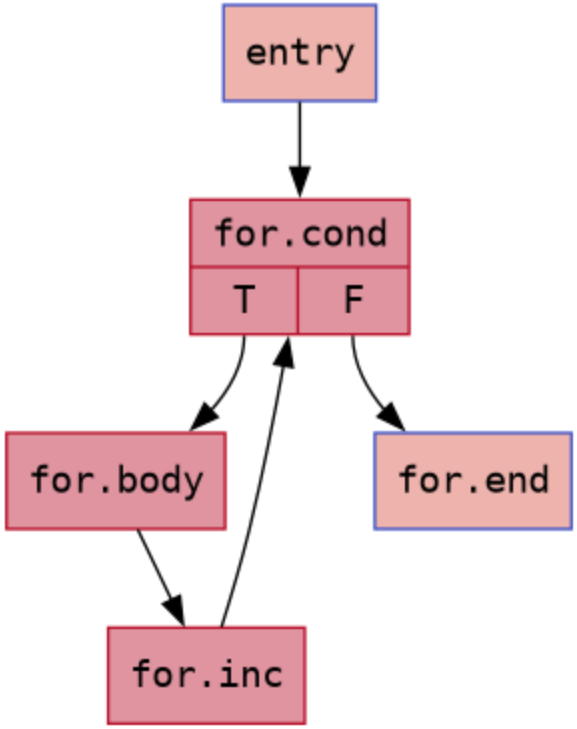
```
i = 0
for i to size:
    if i + 3 >= size:
        break
    C[i] = A[i] + B[i];
    C[i + 1] = A[i + 1]
                + B[i + 1]
    C[i + 2] = A[i + 2]
                + B[i + 2]
    C[i + 3] = A[i + 3]
                + B[i + 3]
    i = i + 4
for i to size:
    C[i] = A[i] + B[i];
    i = i + 1
```

#pragma unroll N - CFG



CFG for 'sumArray' function

Unrolled by 4



CFG for 'sumArray' function

Not unrolled

#pragma unroll N - LLVM

```
for.body:
%i.08 = phi i64 [ %inc, %for.body ], [ 0, %entry ]
%arrayidx = getelementptr inbounds
            i32, ptr %arrA, i64 %i.08
%0 = load i32, ptr %arrayidx, align 4, !tbaa !5
%arrayidx1 = getelementptr inbounds
            i32, ptr %arrB, i64 %i.08
%1 = load i32, ptr %arrayidx1, align 4, !tbaa !5
%add = add nsw i32 %1, %0
%arrayidx2 = getelementptr inbounds
            i32, ptr %arrC, i64 %i.08
store i32 %add, ptr %arrayidx2, align 4, !tbaa !5
%inc = add nuw i64 %i.08, 1
%exitcond.not = icmp eq i64 %inc, %size
br i1 %exitcond.not, label %for.cond.cleanup,
        label %for.body, !llvm.loop !9
}
```

Not unrolled

```
for.body:   ; preds = %for.body, %for.body.preheader.new
%i.08 = phi i64 [ 0, %for.body.preheader.new ],
                [ %inc.3, %for.body ]
%niter = phi i64 [ 0, %for.body.preheader.new ],
                [ %niter.next.3, %for.body ]
%arrayidx = getelementptr inbounds
            i32, ptr %arrA, i64 %i.08
%3 = load i32, ptr %arrayidx, align 4, !tbaa !5
%arrayidx1 = getelementptr inbounds
            i32, ptr %arrB, i64 %i.08
%4 = load i32, ptr %arrayidx1, align 4, !tbaa !5
%add = add nsw i32 %4, %3
%arrayidx2 = getelementptr inbounds
            i32, ptr %arrC, i64 %i.08
store i32 %add, ptr %arrayidx2, align 4, !tbaa !5
%inc = or disjoint i64 %i.08, 1
...
%arrayidx.3 = getelementptr inbounds
            i32, ptr %arrA, i64 %inc.2
%9 = load i32, ptr %arrayidx.3, align 4, !tbaa !5
%arrayidx1.3 = getelementptr inbounds
            i32, ptr %arrB, i64 %inc.2
%10 = load i32, ptr %arrayidx1.3, align 4, !tbaa !5
%add.3 = add nsw i32 %10, %9
%arrayidx2.3 = getelementptr inbounds
            i32, ptr %arrC, i64 %inc.2
store i32 %add.3, ptr %arrayidx2.3, align 4, !tbaa !5
%inc.3 = add nuw i64 %i.08, 4

%niter.next.3 = add i64 %niter, 4
%niter.ncmp.3.not = icmp eq i64 %niter.next.3, %unroll_iter
br i1 %niter.ncmp.3.not, label
             %for.cond.cleanup.loopexit.unr-lcssa,
             label %for.body, !llvm.loop !11
}
```

Unrolled by 4

#pragma unroll N - LLVM

The sum of the elements on the array **occurs 4 times**, during the same iteration of the loop. (a)

Increments within the loop. (b)

Annotations in the loop. (c)

```
for.body:   ; preds = %for.body, %for.body.preheader.new
%i.08 = phi i64 [ 0, %for.body.preheader.new ],
                [ %inc.3, %for.body ]
%niter = phi i64 [ 0, %for.body.preheader.new ],
                 [ %niter.next.3, %for.body ]                    c)
a) %arrayidx = getelementptr inbounds
              i32, ptr %arrA, i64 %i.08
   %3 = load i32, ptr %arrayidx, align 4, !tbaa !5
   %arrayidx1 = getelementptr inbounds
              i32, ptr %arrB, i64 %i.08
   %4 = load i32, ptr %arrayidx1, align 4, !tbaa !5
   %add = add nsw i32 %4, %3
   %arrayidx2 = getelementptr inbounds
              i32, ptr %arrC, i64 %i.08
   store i32 %add, ptr %arrayidx2, align 4, !tbaa !5
   %inc = or disjoint i64 %i.08, 1                               b)
   ...
a) %arrayidx.3 = getelementptr inbounds
              i32, ptr %arrA, i64 %inc.2
   %9 = load i32, ptr %arrayidx.3, align 4, !tbaa !5
   %arrayidx1.3 = getelementptr inbounds
              i32, ptr %arrB, i64 %inc.2
   %10 = load i32, ptr %arrayidx1.3, align 4, !tbaa !5
   %add.3 = add nsw i32 %10, %9
   %arrayidx2.3 = getelementptr inbounds
              i32, ptr %arrC, i64 %inc.2
   store i32 %add.3, ptr %arrayidx2.3, align 4, !tbaa !5
   %inc.3 = add nuw i64 %i.08, 4                                 b)
c) %niter.next.3 = add i64 %niter, 4
   %niter.ncmp.3.not = icmp eq i64 %niter.next.3, %unroll_iter
   br i1 %niter.ncmp.3.not, label
                %for.cond.cleanup.loopexit.unr-lcssa,
                label %for.body, !llvm.loop !11
}
```

#pragma clang loop unroll(enable/disable)/unroll_count(N)

Force the compiler to unroll or to not unroll a specific loop. (a, b)

The compiler choose the unrolling strategy based on various optimization factors.

To **explicit the unroll factor** is necessary use 'unroll_count(N)' which force an unrolling with a factor of N. (b)

a)

```
i = 0
#pragma clang loop unroll(enable)
for i to size:
    C[i] = A[i] + B[i]
```

b)

```
i = 0
#pragma clang loop unroll(disable)
for i to size:
    C[i] = A[i] + B[i]
```

c)

```
i = 0
#pragma clang unroll_count(4)
for i to size:
    C[i] = A[i] + B[i]
```

unroll N *vs* clang loop unroll(enable)/unroll_count(N)

#pragma unroll N ‘suggests’ the compiler to unroll. (a)

#pragma clang loop unroll(enable) ‘forces’ the compiler to unroll loop. (b)

#pragma clang loop unroll_count(N) ‘forces’ the compiler to unroll loop on a factor of N. (c)

a)

```
i = 0
#pragma unroll 4
for i to size:
    C[i] = A[i] + B[i]
```

b)

```
i = 0
#pragma clang loop unroll(enable)
for i to size:
    C[i] = A[i] + B[i]
```

c)

```
i = 0
#pragma clang unroll_count(4)
for i to size:
    C[i] = A[i] + B[i]
```

#pragma clang loop vectorize(enable)

Similar to '#pragma omp simd' proposed by the OpenMP library.

Operations are done on 'intervals'. (a)

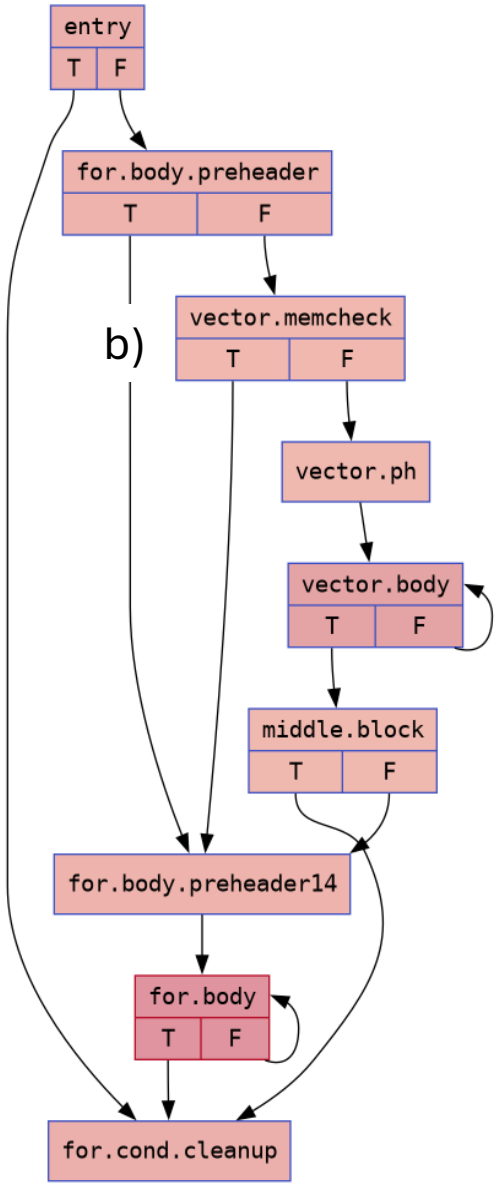
Check on memory. (b)

```
i = 0
#pragma clang loop vectorize(enable)
for i to size:
    C[i] = A[i] + B[i]
```

a)

```
i = 0
for i to size:
    C[i :  i +3] =
      A[i :  i +3] + B[i :  i +3]
```

b)



CFG for 'sumArray' function

#pragma clang loop vectorize(enable) - LLVM

LLVM bitcode generated has a new 'part'. (a)

Load of 4 elements. (b)

Sum using **SIMD register**. (c)

Store of 4 elements. (d)

a)
```
vector.body:      ; preds = %vector.body, %vector.ph
%index = phi i64 [ 0, %vector.ph ],
         [ %index.next, %vector.body ]
%2 = getelementptr inbounds i32,
         ptr %arrA, i64 %index
```
b)
```
%wide.load = load <4 x i32>,
         ptr %2, align 4, !tbaa !5
%3 = getelementptr inbounds i32,
         ptr %arrB, i64 %index
```
b)
```
%wide.load13 = load <4 x i32>,
         ptr %3, align 4, !tbaa !5
%4 = add nsw <4 x i32> %wide.load13,
         %wide.load
```
c)
```
%5 = getelementptr inbounds i32,
         ptr %arrC, i64 %index
store <4 x i32> %4, ptr %5, align 4,
         !tbaa !5
```
d)
```
%index.next = add nuw i64 %index, 4
%6 = icmp eq i64 %index.next, %n.vec
br i1 %6, label %middle.block,
         label %vector.body, !llvm.loop !9
```

Execution test

2 test cases, one heavyer (a) than the other (b).

Unexpected results.

Different **#pragma combined**. (c)

a)

```
for (size_t i = 0; i < size; ++i) {
    arrC[i] = sin(arrA[i]) * cos(arrB[i])
            + exp(arrA[i]) * log(arrB[i]);
}
```

b)

```
for (size_t i = 0; i < size; ++i) {
    arrC[i] = sin(arrA[i]) + sin(arrB[i]);
}
```

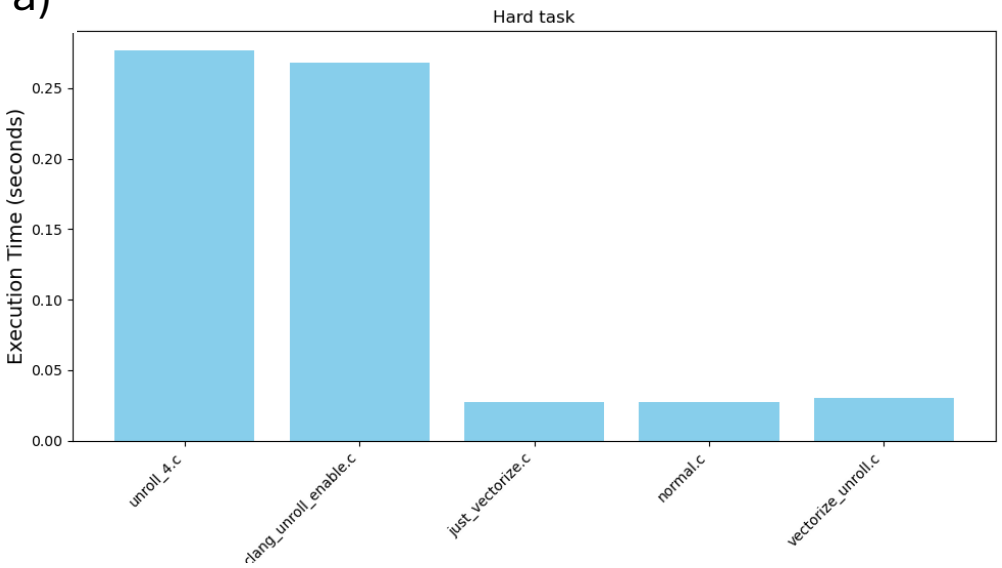
c)

```
i = 0
  #pragma clang loop unroll(enable)
  #pragma clang loop vectorize(enable)
for i to size:
    C[i] = A[i] + B[i]
```

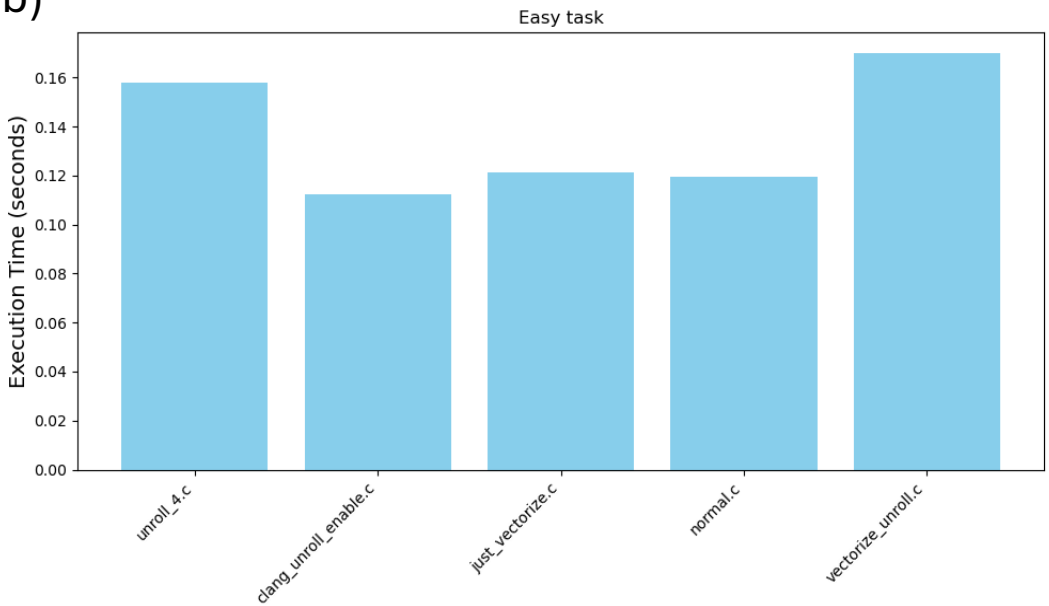
Execution test

Loop unrolling makes execution faster only if the cost of the task inside the loop is lighter than the loop itself.

a)



b)



References

llvm.org: https://llvm.org

llvm.org metadata: https://llvm.org/docs/TransformMetadata.html

Kruse-LoopTransforms: https://llvm.org/devmtg/2018-10/slides/Kruse-LoopTransforms.pdf

Thanks for the attention