Loop Iteration Space Splitting

Ashutosh Nema, Anupama Rasale, Venkataramanan Kumar, Raghesh Aloor

AMD Compilers Team

Loop Splitting

- Loop iteration space splitting is the process of dividing a loop into several smaller loops, each handling a portion of the original loop's iterations

- LLVM currently includes a pass called `InductiveRangeCheckElimination` that performs loop splitting to eliminate range checks

Loop Splitting

- The InductiveRangeCheckElimination pass divides a loop's iteration space into separate ranges, ensuring that the loop within the split loop segment doesn't require range checks

- However, the pass has limited applicability since its goal is solely to remove checks performed on induction variables

```
len = < known positive >
  for (i = 0; i < n; i++) {
    if (0 <= i && i < len) {
      do_something();
    } else {
      throw_out_of_bounds();
    }
  }
```

To

```
  len = < known positive >
  limit = smin(n, len)
  // no first segment
  for (i = 0; i < limit; i++) {
    if (0 <= i && i < len) {
      // Check is not required
      // This block can execute unconditionally
      do_something();
    } else {
      throw_out_of_bounds();
    }
  }
  for (i = limit; i < n; i++) {
    if (0 <= i && i < len) {
      do_something();
    } else {
      // Check is not required
      // This block can execute unconditionally
      throw_out_of_bounds();
    }
  }
```

The code highlighted in RED will get eliminated, only code highlighted in GREEN will remain.

Loop Splitting

- Beyond the current method of eliminating induction range checks, there are additional scenarios where employing loop splitting could facilitate further optimizations

- Here are some potential areas where loop splitting could be beneficial:
  - Performing Memory Alignment Checks for generating NTStores
  - Facilitating Multi-Exit Loop Vectorization
  - Eliminating fixed-point memory dependencies

- These possibilities highlight the necessity of incorporating loop splitting as a versatile utility for optimization purposes

Memory Alignment Checks For NTStores generation

MOVNTPD — Store Packed Double Precision Floating-Point Values Using Non-Temporal Hint

Description ¶

Moves the packed double precision floating-point values in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register, YMM register or ZMM register, which is assumed to contain packed double precision, floating-pointing data. The destination operand is a 128-bit, 256-bit or 512-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (EVEX.512 encoded version) boundary otherwise a general-protection exception (#GP) will be generated.

Nontemporal stores or streaming stores avoid writing to the cache line and writes directly to memory

The challenge in using the vector variant (vmovntpd), is that the store address had to be aligned to 16/32/64-byte boundary

Memory Alignment Checks For NTStores generation

- There is a necessity to partition the loops into two subloops. The first subloop will peel off a few iterations, ensuring that the memory accesses for the second loop are aligned as needed

```
double a[STREAM_ARRAY_SIZE];
double b[STREAM_ARRAY_SIZE];
double c[STREAM_ARRAY_SIZE];

for (i=0; i<STREAM_ARRAY_SIZE;i++)
  c[i] = a[i] + b[i] * scalar
```

To:

PeelFactor = Compute the steps required to go to next 32/64-byte address from the start address of array “c[i]”

```
 PeelLoopCount = MIN(PeelFactor, STREAM_ARRAY_SIZE)

for (i = 0; i< PeelLoopCount; i++)
  c[i] = a[i] + b[i] * scalar

for (i = PeelLoopCount; i<STREAM_ARRAY_SIZE; i++)
  c[i] = a[i] + b[i] * scalar
```

Peeled Loop

Loop will be vectorized with vmovntpd

Multi Exit Loop Vectorization

- At present, the LLVM loop vectorizer doesn't accommodate loops with multiple exits, potentially leading to missed optimization chances

- There have been discussions in the past about the necessity of extending the loop vectorizer to address cases with multiple exits:
  - https://lists.llvm.org/pipermail/llvm-dev/2019-September/134998.html
  - https://github.com/preames/public-notes/blob/master/multiple-exit-vectorization.rst

- This poses a challenge in scenarios where the exit condition depends on a memory access. Cases where the exit is taken between start and end of a vector iteration, the vector load instruction may access memory beyond its allocated range, potentially leading to a runtime application crash

- The AMD AOCC (AMD Optimizing C/C++ and Fortran Compilers) extends the loop vectorizer to include support for multi-exit vectorization

Multi Exit Loop Vectorization

- Having multiple exits in a vector loop presents a safety challenge for memory accesses

- Specifically, if a vector load accesses a memory location beyond its allocated point, there is a risk of crossing the page boundary at runtime, potentially leading to an application crash

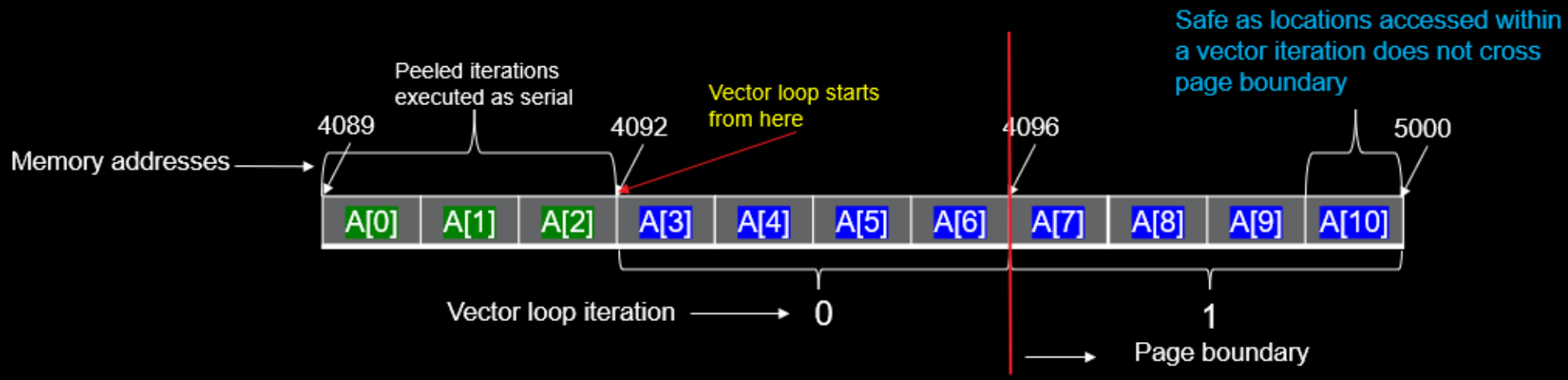
```
Example:
 for (int i = 0; i < N; i++)
   if (A[i] == 0)
     break;
```

- *Consider following assumptions:*
  - *Memory allocated for array A is 10*
  - *N = 12, Vector Factor = 4*
  - *Early exit condition becomes true when i = 8*
- *Now, in the vectorized loop, when vector loop iteration is 2, it loads (A[8], A[9], A[10] and A[11])*
- *This may result in the application crash as **10th and 11th locations of A are not allocated***

Multi Exit Loop Vectorization

- It's essential to guarantee that the start address for memory locations accessed in a vector loop is a multiple of the vector factor

- In AOCC, we introduce a runtime check to verify this condition. If necessary, we split or peel off a few iterations so that for the remaining iterations, memory accesses conform to the requirement of being multiples of the vector factor



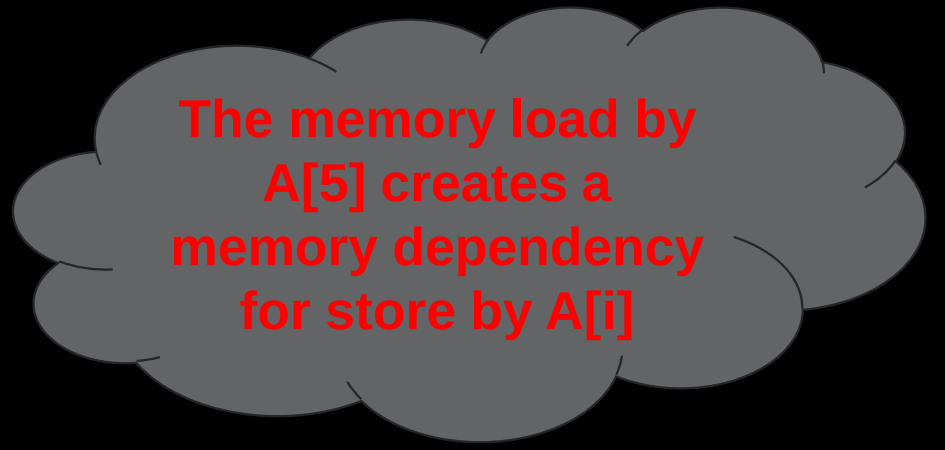
- This solution is limited to scenarios where the loop's exit condition depends on a single memory access

Fixed Point Memory Dependencies

- Fixed-point memory dependencies refer to memory dependencies that arise when working with fixed memory location access in a loop

- Example:

```
void foo(int *A, int *B, int *C, unsigned len) {
  for (int i = 0; i < len; i++)
    A[i] = A[5] + B[i] + C[i];
}
```



- Compiler optimization such as auto vectorization in presence of fixed-point memory dependencies are avoided and results in significant loss of application performance opportunities

Fixed Point Memory Dependencies

```
void foo(int *A, int *B, int *C, unsigned len) {
  for (int i = 0; i < len; i++)
    A[i] = A[5] + B[i] + C[i];
}
```

A		A		B		C
A[0]	=	A[5]	+	B[0]	+	C[0]
A[1]		A[5]		B[1]		C[1]
A[2]		A[5]		B[2]		C[2]
A[3]		A[5]		B[3]		C[3]
A[4]		A[5]		B[4]		C[4]
A[5]		A[5]		B[5]		C[5]
A[6]		A[5]		B[6]		C[6]
A[7]		A[5]		B[7]		C[7]

Please note: The vector for A[5] for the last two vector lanes has the stale values as the memory location has updated

This creates a memory dependency and prevents vectorization

Fixed Point Memory Dependencies

- The elimination of fixed-point memory dependencies is crucial to enable the key optimizations like loop vectorization
- This memory dependency is limited to a single iteration or a set of iterations as it deal with fixed point memory access
- Identify such a dependency and deploy loop splitting to separate the memory dependent iteration and nondependent iteration into separate sub loops

Fixed Point Memory Dependencies

- Loop iteration space splitting involves breaking a single loop into multiple smaller loops, each processing a subset of the original loop's iterations

```
for (int i = 0; i <= len; i++) {
  A[i] = A[5] + B[i] + C[i];
}
```

Loop Split to Eliminate Dependency:

```
MIN1 = MIN (len, 4)
for (int i = 0; i <= MIN1; i++) {
  A[i] = A[5] + B[i] + C[i]; // A[5]  is invariant and it can be hoisted
}

MIN2 = MIN (len, 5)
for (i = MIN1+1; i <= MIN2; i++) {
  A[i] = A[5] + B[i] + C[i];
}

for (i = MIN2+1; i <= len; i++) {
  A[i] = A[5] + B[i] + C[i]; // A[5]  is invariant and it can be hoisted
}
```

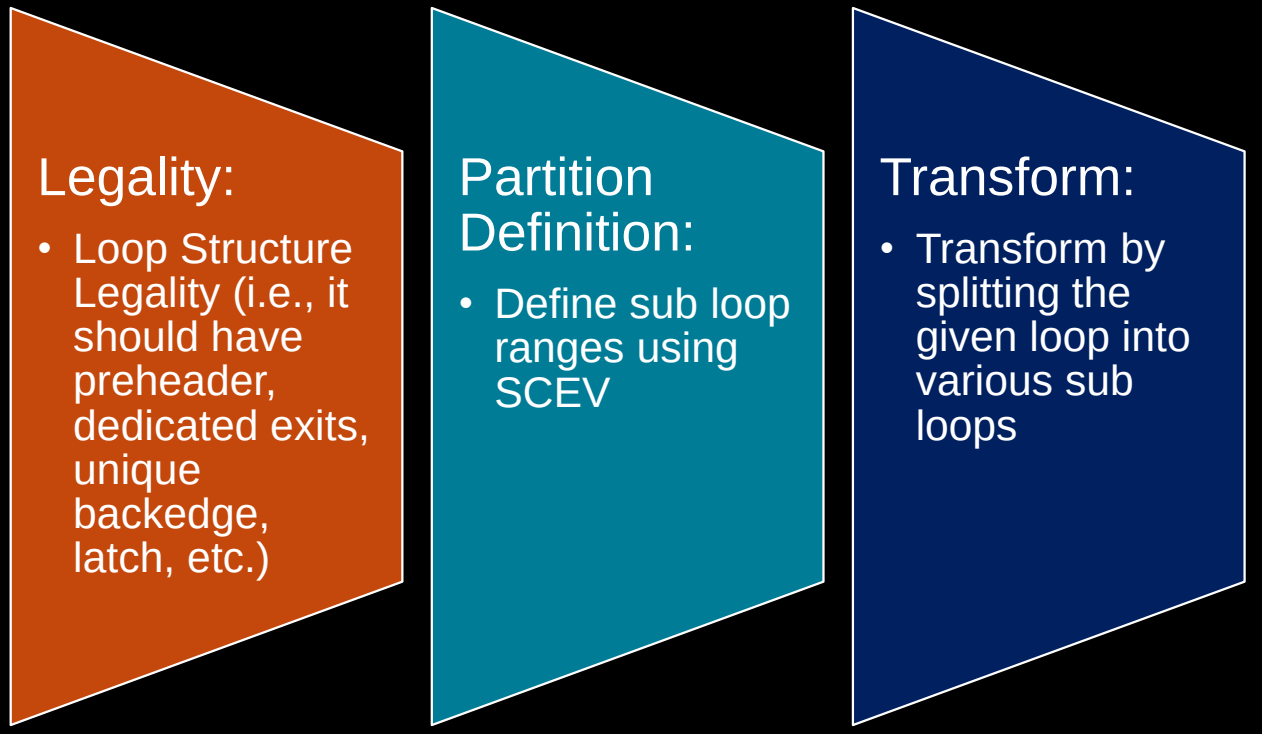
Loop without any dependency

Loop without any dependency

Loop Splitting Utility

- In AOCC, we've implemented LoopSplitting as a utility

- Its primary divided into following phases:
  - Legality
  - Partition Definition
  - Transform



Loop Splitting Utility

- Example:

How loop splitting utility is invoked

```
LoopSplitUtils *LSU = new LoopSplitUtils(CandLoop, SE, DT, DL, LI);

if (LSU->isLegalForItrSplitting()) {
   // Create Partition with different
   // SCEV bounds (S1, S2, S3, S4, S6, S6)
   // LoopRange#1 S1-S2
   // LoopRange#2 S3-S4
   // LoopRange#3 S5-S6

   Partition *P0 = new Partition(S1, S2);
   Partition *P1 = new Partition(S3, S4);
   Partition *P2 = new Partition(S5, S6);

   // Add Partitions
   LSU->addPartition(P0, 0);
   LSU->addPartition(P1, 1);
   LSU->addPartition(P2, 2);

   // Perform the splitting
   LSU->performLoopSplitting();
}
```

Split Loops

```
// Partition#0 - LoopRange#1 S1-S2
for (i = S1; i <= MIN(S2, OrigLoopInductionRangeEndPoint); i++) {
  ...
}
// Partition#1 - LoopRange#2 S3-S4
for (i = S3; i <= MIN(S4, OrigLoopInductionRangeEndPoint); i++) {
  ...
}
// Partition#3 - LoopRange#3 S5-S6
for (i = S5; i <= MIN(S6, OrigLoopInductionRangeEndPoint); i++) {
  ...
}
```

Summary

- Loop iteration space splitting entails dividing a single loop into multiple smaller loops, each handling a subset of the original loop's iterations
- Loop splitting holds potential for enabling various optimization opportunities
- In AOCC, we've implemented LoopSplitting as a utility and aim to integrate it into the community LLVM

Copyright and disclaimer


AMD