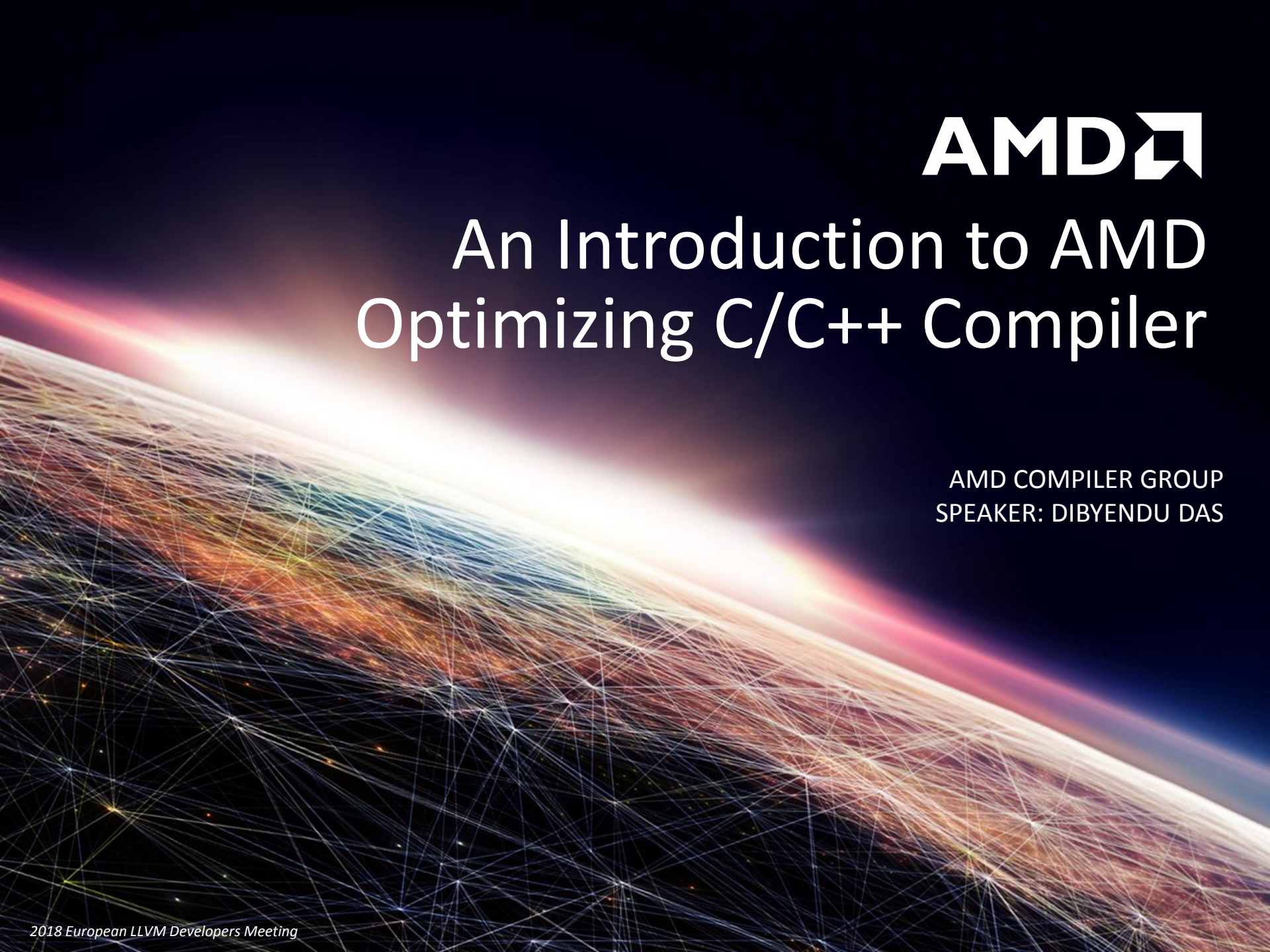


An Introduction to AMD Optimizing C/C++ Compiler

AMD COMPILER GROUP
SPEAKER: DIBYENDU DAS

2018 European LLVM Developers Meeting

OVERVIEW OF TALK

AMD

- AMD Optimizing C/C++ Compiler (“AOCC”) overview
- Optimizations
- Results
  - SPEC CPU® 2017 Rate
    - More information about SPEC CPU2017 available at www.spec.org
- Conclusion

AOCC Overview

WHAT IS AOCC ?

AMD

- AOCC is AMD's Optimizing C/C++ (and Fortran using DragonEgg/Flang) compiler based on LLVM
- First version released mid-2017
- Targeted for AMD's Zen and future processors
- Multiple release every year based on latest LLVM releases

OPTIMIZATIONS IN AOCC

AMD

- Many optimizations – in this talk, we cover the following
  - Vectorization
    - Strided
    - Epilog
    - SAD, AVG
    - SLP (jumbled memory)
  - Data layout optimization
    - Array remapping
    - AOS -> SOA
  - Loop optimizations
    - Loop-versioned LICM
    - Path-invariance based loop un-switching
    - Improved loop strength reduction
  - Generic scalar optimizations
    - Recursion inlining
    - Dynamic cast removal
  - LLC optimizations
    - znver1 scheduler model

AOCC

Vectorization

LOOP VECTORIZATION – SAD GENERATION

AMD

- Generation of SAD (Sum of Absolute Difference) instruction
- Modified both the loop vectorizer and the SLP vectorizer

Inner loop of x264_pixel_sad_8x8() in pixel.c

```
for( int x = 0; x < lx; x++ )
{
    i_sum += abs( pix1[x] - pix2[x] );
}
```

(lx=8, pix1 and pix2 are uint8 pointers and i_sum is of type 'int')

```
movq   (%rdi),%xmm3
movq   (%rdx),%xmm1
psadbw %xmm1,%xmm3
```

- http://lists.llvm.org/pipermail/llvm-dev/2015-February/081561.html

EPILOG LOOP VECTORIZATION

AMD

- Currently Loop Vectorizer inserts an epilogue loop for handling loops that are not multiples of the ‘vector factor(VF)’
  - Executed as scalar code
- Epilog vectorization aims to vectorize epilog loop where original loop is vectorized with large vector factor
  - Ex: for VF=16, you may have up to 15 iterations in the epilog
  - Try to vectorize that using a lower VF=8 or VF=4
- http://llvm.1065342.n5.nabble.com/llvm-dev-Proposal-RFC-Epilog-loop-vectorization-td106322.html

STRIDED VECTORIZATION

AMD

- Compilers may fail to vectorize loops with strided accesses
- Vectorization of strided data may incur
  - An overhead of 'consolidating' data into an operable vector, refer Figure (a)
  - An overhead of 'distributing' the data elements after the operations - refer Figure (b).
- Designed improved strided vectorization
  - Uses 'skip factor'

```
for (int i = 0; i < len; i++)
    a[i*2] = b[i*2] + c[i*3];
```

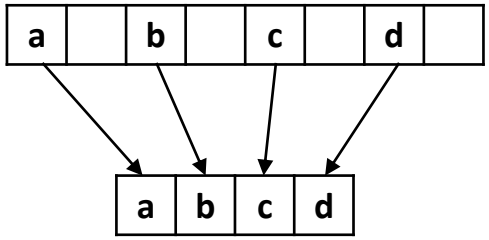


Figure (a) : Example with stride 2 - loading data into an operable vector

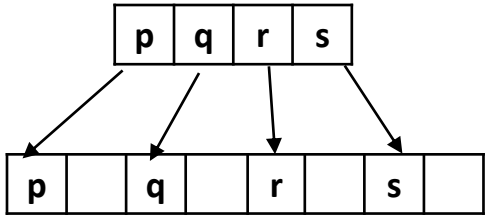
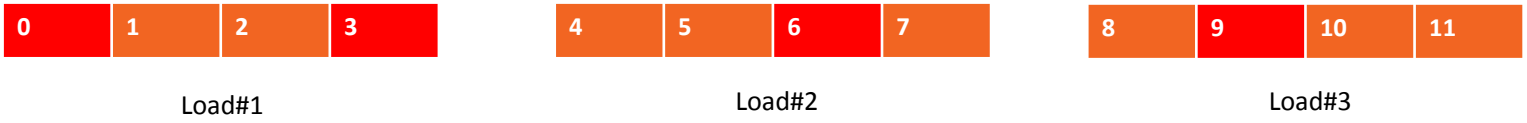


Figure (b) : Example with stride 2 - storing data to strided memory

STRIDED VECTORIZATION - MEMORY SKIPPING

- Skip factor helps to minimize the number of loads and stores
  - Example: for stride = 3 and VF = 4, generally 3 loads are required
  - But by skipping memory locations we can do with 2 loads



Load with stride 3 (i.e. load for b [3 * i])

```
%5 = getelementptr inbounds i32, i32* %b, i64 %.induction
%6 = bitcast i32* %5 to <4 x i32>*
%stride.load27 = load <4 x i32>, <4 x i32>* %6, align 1
%7 = getelementptr i32, i32* %5, i64 6
%8 = bitcast i32* %7 to <4 x i32>*
%stride.load28 = load <4 x i32>, <4 x i32>* %8, align 1
%strided.vec29 = shufflevector <4 x i32> %stride.load27, <4 x i32> %stride.load28, <4 x i32> <i32 0, i32 3, i32 4, i32 7>
```

Note: Next GEP offset by 6, from previous load

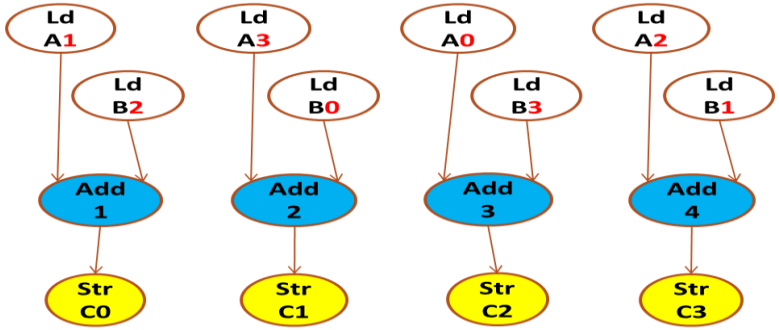
http://llvm.1065342.n5.nabble.com/llvm-dev-Proposal-RFC-Strided-Memory-Access-Vectorization-td96860.html

SLP VECTORIZATION

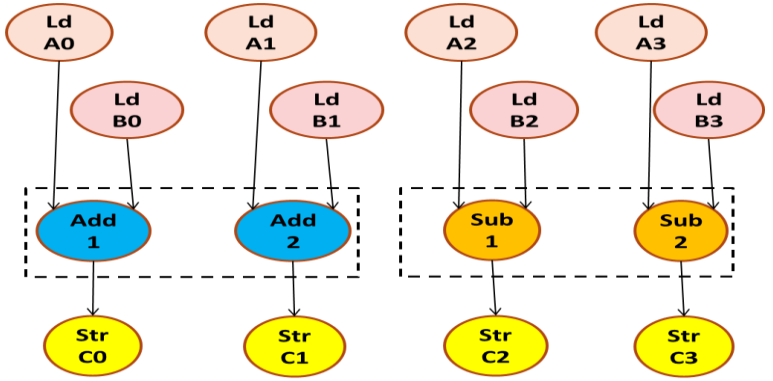
AMD

- Non-consecutive memory accesses add overheads in vectorization
  - Memory accesses may be jumbled
  - As a group they access consecutive memory locations though
- Non-isomorphic operations such as ADD-SUB, MUL-SHIFT prevent exploitation of SIMD ALU ops
- We cater to these scenarios
  - Memory accesses are made consecutive
  - Increasing isomorphism
- Submitted as a patch - https://reviews.llvm.org/D36130

Non-Consecutive Accesses



Non-Isomorphic ALU Ops



LOOP VECTORIZATION - VPAVGB

AMD

- Generation of AVG (average) instruction

```
static void pixel_avg( uint8_t *dst,  int i_dst_stride,
                       uint8_t *src1, int i_src1_stride,
                       uint8_t *src2, int i_src2_stride,
                       int i_width, int i_height )
{
    for( int y = 0; y < i_height; y++ )
    {
        for( int x = 0; x < i_width; x++ )
            dst[x] = ( src1[x] + src2[x] + 1 ) >> 1;
        dst  += i_dst_strid
        src1 += i_src1_stri
        src2 += i_src2_stri
    }
}
```

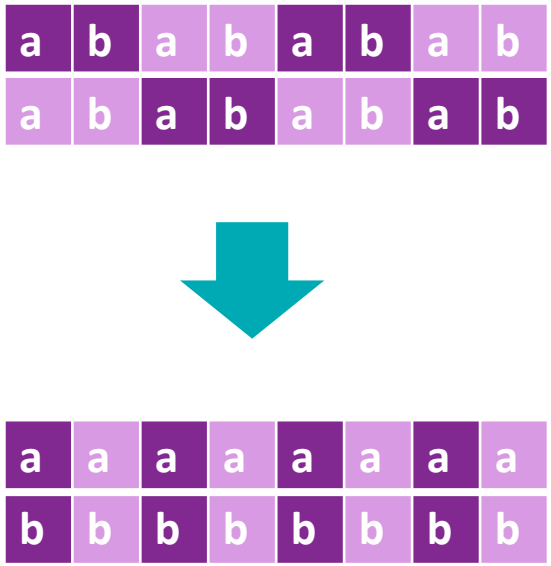
VPAVGB (VEX.128 encoded version)

AOCC

Data Layout

AOS -> SOA

```
struct {                    struct {
    long a;                     arr_a [N];
    float b;                    arr_b[N];
 } arr[N];                    } Ns;
main() {
        ...
       for (i=0; i < N; i++)
              ... = Ns.arr_a[i];
         ...
        for (i=0; i < N; i++)
              ... = Ns.arr_b[i];
        …..
 }
```



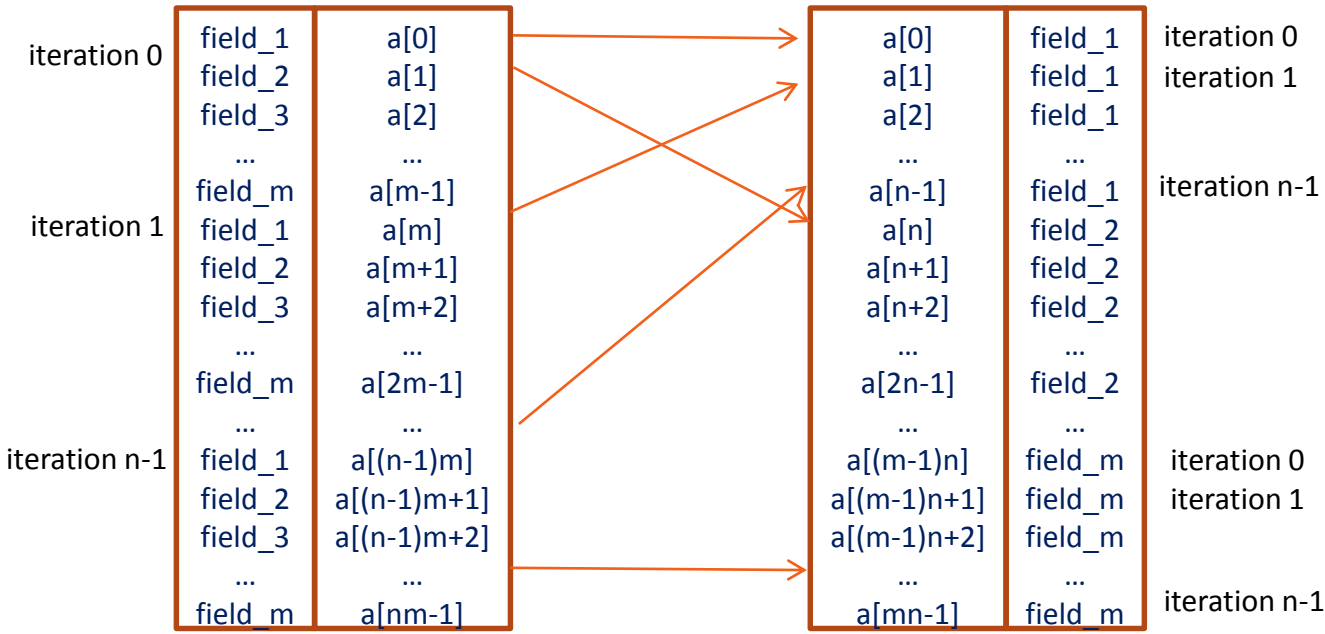
Less cache misses

ARRAY REMAPPING

- Transforms array accesses in a single dimensional array for better cache utilization
- Better cache utilization

```
void LBM_performStreamCollide(
      LBM_Grid srcGrid, LBM_Grid dstGrid ) {
      int i;
      double ux, uy, uz, u2, rho;

      for( i = 0; i < 20*1300000; i += N_CELL_ENTRIES ) {
            if( TEST_FLAG_SWEEP( srcGrid, OBSTACLE )) {
                  dstGrid[0 + i]          = srcGrid[0 + i];
                  dstGrid[-1998 + i]    = srcGrid[1 + i];
                  dstGrid[2001 + i]     = srcGrid[2 + i];
                  dstGrid[-16 + i]       = srcGrid[3 + i];
                  dstGrid[23 + i]        = srcGrid[4 + i];
                  dstGrid[-199994 + i] = srcGrid[5 + i];
                  dstGrid[200005 + i]  = srcGrid[6 + i];
                  dstGrid[-2010 + i]    = srcGrid[7 + i];
                  dstGrid[-1971 + i]    = srcGrid[8 + i];
                  dstGrid[1988 + i]     = srcGrid[9 + i];
                  dstGrid[2027 + i]     = srcGrid[10 + i];
                  dstGrid[-201986 + i] = srcGrid[11 + i];
                  dstGrid[198013 + i]  = srcGrid[12 + i];
                  dstGrid[-197988 + i] = srcGrid[13 + i];
                  dstGrid[202011 + i]  = srcGrid[14 + i];
                  dstGrid[-200002 + i] = srcGrid[15 + i];
                  dstGrid[199997 + i]  = srcGrid[16 + i];
                  dstGrid[-199964 + i] = srcGrid[17 + i];
                  dstGrid[200035 + i]  = srcGrid[18 + i];
                  continue;
            }
      ...
```



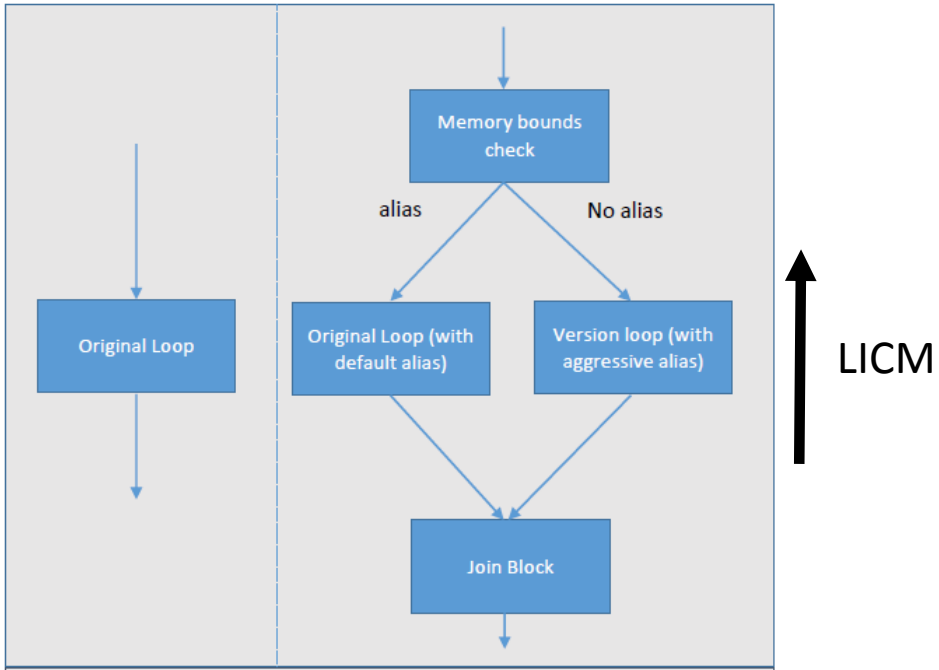
a[i] becomes a[(i%m)*n+(i/m)]

AOCC

Loop Optimization

LOOP VERSIONING LICM

- Aliasing decisions are made at runtime
- Creates two versions of the loop
  - One with aggressive aliasing assumptions
  - The original loop with conservative aliasing assumptions
- These two loops are preceded by a memory runtime check [upstreamed]



PARTIAL LOOP UNSWITCH

AMD

- Identifies partial-invariant condition for a path
- Moves the conditional from inside the loop to outside of it by duplicating the loop's body
- Places a loop version inside each of the if and else clauses of the conditional
- The variant path has the full loop with all conditions
- The partially invariant path has the improved version

Original Loop	Loop with partial un-switched version

for (i = 0; i < N; i++)
  if (X)
  a[i] = 0;
  else {
  X = b[i]; // X is modified
  b[i] = <...>
  }
	LoopExecutionAssurance = (0 < N);
	if (LoopExecutionAssurance) {
	  if (X) {
	  for (i = 0; i < N; i++)
	  a[i] = 0;
	  } else {
	  // Original loop version
	  for (i = 0; i < N; i++)
	  if (X)
	  a[i] = 0;
	  else {
	  X = b[i]; // X is modified
	  b[i] = <...>
	  }
	  }
	}

OTHER LOOP OPTIMIZATIONS

AMD

- Improved induction variable life time splitting
- Improved loop strength reduction (LSR) in nested loop

Scalar Optimizations

DYNAMIC CAST OPTIMIZATION

AMD

- A dynamic cast test in C++ is converted into a typeid comparison when the cast involves a leaf class in the inheritance graph

```
if (dynamic_cast<EtherPauseFrame*>(frame)!=NULL)
{
   …
}
```

This is transformed into:

```
If (typeid(*frame) == typeid(EtherPauseFrame *))
{
   …
 }
```

RECURSION INLINING

AMD

- Enables the inlining of recursive function
- Works up to a certain depth by generating function clones

AMD

LLC Optimizations

LLC OPTS

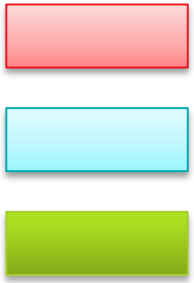
znver1 Scheduler Model

Promote constant to register:
Replace ADD R1 ← R2, k (where k is a constant) with
MOV R3 ← k and ADD R1 ← R2, R3

Redundant Load/Store and MOV Elimination

Branch Fusion: Re-order code to place CMP and TEST instructions immediately preceding BRANCH instructions

Register Pressure-aware LICM



Shorter instruction encoding

Reduced instruction path length

Enable hardware micro-op fusion

ZNVER1 SCHEDULER MODEL

AMD

- Zen scheduler model is added for sub-target named "znver1"

  File: .../lib/Target/X86/X86ScheduleZnver1.td

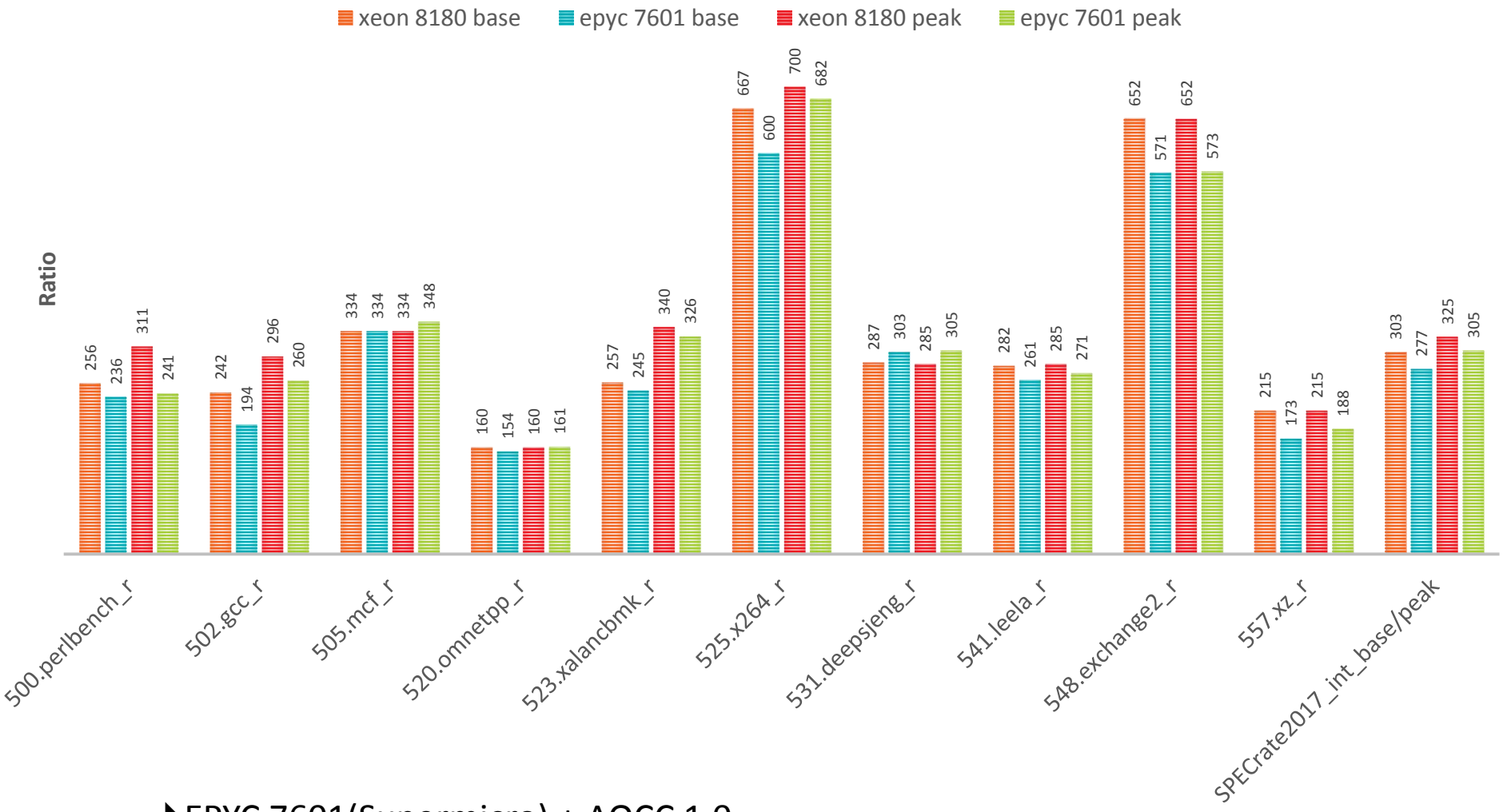
- Covers all Zen supported ISAs. Instructions are grouped as per their nature(Integer, FP, Move, Arithmetic, Logic, Control Transfer)
- Exhaustive model that covers both integer and floating point execution units
  - Covers latencies and micro-op details of all modeled instructions
- Microcoded instructions are marked as WriteMicrocoded with high latency
- Upstreamed

AOCC

Results

SPEC CPU® 2017 Rate (INT)

EPYC™ 7601 VS XEON PLATINUM 8180

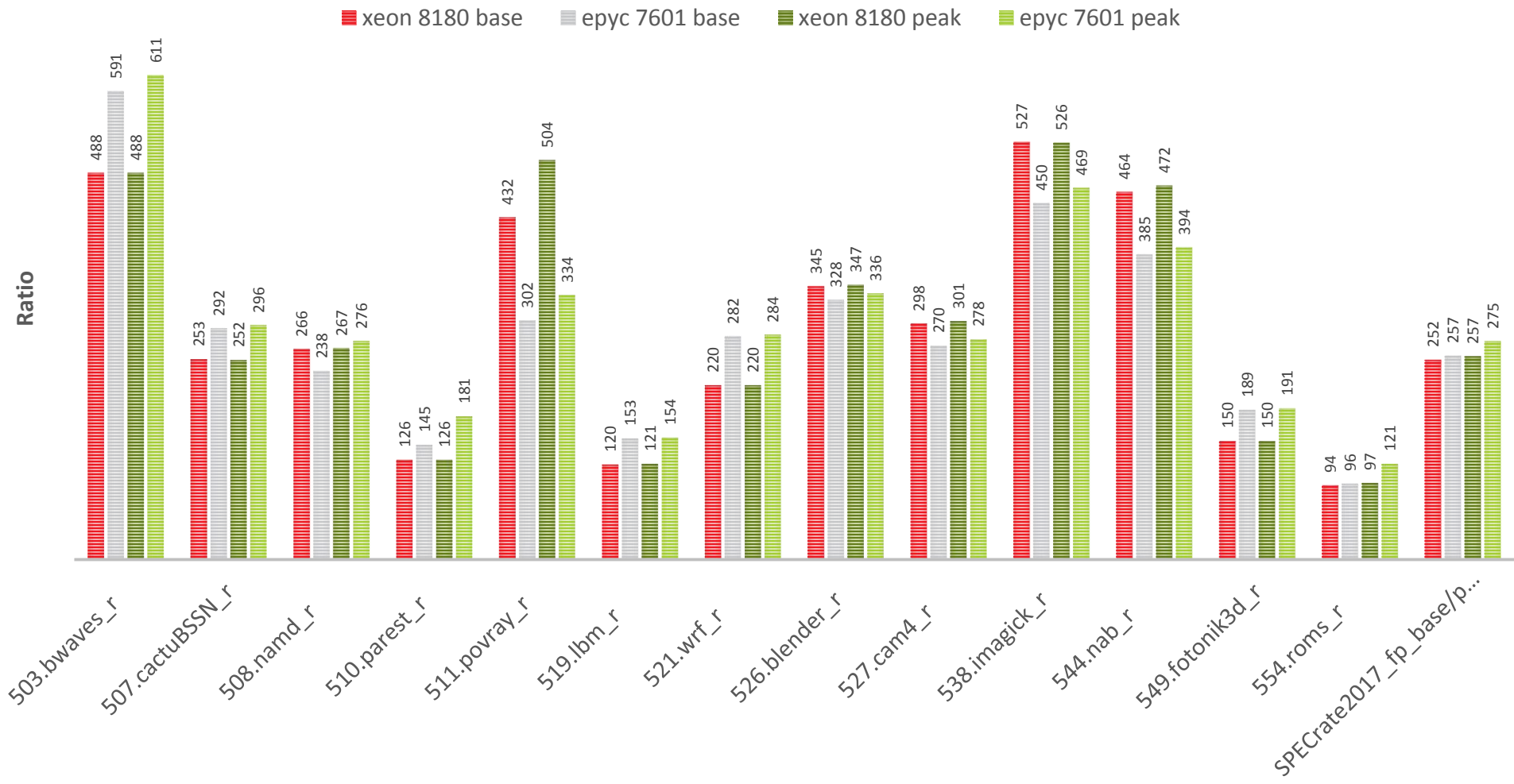


- EPYC 7601(Supermicro) + AOCC 1.0
- XEON 8180(Asus) + icc 18.0.0.128

- https://www.spec.org/cpu2017/results/res2018q1/cpu2017-20180108-02536.html
- https://www.spec.org/cpu2017/results/res2018q1/cpu2017-20180121-02623.html

SPEC CPU® 2017 Rate (FP)

EPYC™ 7601 VS XEON PLATINUM 8180



- EPYC 7601 (Supermicro) + AOCC 1.0
- XEON 8180 (Asus) + icc 18.0.0.128

- https://www.spec.org/cpu2017/results/res2018q1/cpu2017-20180108-02537.html
- https://www.spec.org/cpu2017/results/res2018q1/cpu2017-20180121-02625.html

PHORONIX

AMD

https://www.phoronix.com/scan.php?page=article&item=amd-aocc-11&num=1

AMD AOCC 1.1 Shows Compiler Improvements vs. GCC vs. Clang (Jan, 2018)

AOCC

Resources

AOCC WEB PAGE (developer.amd.com)

AMD

https://developer.amd.com/amd-aocc/

We have released AOCC 1.1 and will release AOCC 1.2 aligned with LLVM 6.0 very soon

Conclusion

CONCLUSION

AMD

- We have demonstrated a powerful optimizing compiler built on top of the latest LLVM
- Introduced many optimizations in opt and llc
  - Some of them upstreamed already
- We want to upstream more aggressively
- A BIG THANK YOU to the entire community for making this possible

ACKNOWLEDGEMENTS

- Abhilash Bhandari
- Anupama Rasale
- Ashutosh Nema
- Bala Rishi Bhogadi
- Bhargav Reddy Godala
- Deepak Porwal
- Deepali Rai
- Ganesh Gopalasubramanian
- Ganesh Prasad
- Md Asghar Ahmad Shahid
- Muthu Kumar Raj Nagarajan
- Nagajyothi Eggone
- Pradeep Rao
- Pratap Gadi
- Prathiba Kumar
- Pratik Dayanand Bhatu
- Rajasekhar Venkata Bhetala
- Ravindra Venkata Durgi
- Santosh Zanjurne
- Satish Kumar Narayanaswamy
- Shivarama Rao
- Suresh Mani
- Venkataramanan Kumar
- Venugopal Raghavan
- Vishwanath Prasad
- Sunil Anthony
- Jay Hiremath

DISCLAIMER & ATTRIBUTION

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

ATTRIBUTION