Optimizations of BLIS Library for AMD ZEN Core

1 Introduction

BLIS [1] is a portable software framework for instantiating high-performance BLAS-like dense linear algebra libraries [2]. The framework was designed to isolate essential kernels of computation that, when optimized, immediately provide needed acceleration for commonly used BLAS routines. Major routines which are optimized in BLIS are level-1 and level-2 routines. In level-3 we only optimize GEMM (General Matrix-Matrix Multiply) & triangular solve matrix (trsm). In this paper, we present optimizations of few level-1, level-2 & level-3 routines. The data types currently supported are *float* and *double*. The optimization employ AVX-256 bit x86-64 SIMD intrinsics. All the algorithms are block-based and at the lowest block level, assembly coded micro-kernels are employed. Block-based technique exploits cache-based architecture systems while AVX instruction set exploits SIMD capability of the hardware.

2 BLIS Micro-kernel Optimization

This section discusses the optimization of the BLIS routines and the subsequent performance gains observed. The benchmark application is BLIS's test-suite with error checking disabled and all matrices are stored in column-major format. All performances are measured on AMD's new "ZEN"core processor. The source code is compiled using the gcc 6.3 version. BLIS routines only operate on floating point data type and hence it is important to exploit the exploit the floating-point unit of the hardware.

The floating-point unit(FPU) refer Figure 1 of "ZEN"core [3] supports 256-bit packed integer, single and double precision vector floating-point data types. There are 4 execution pipes refer Figure 2 which can execute an operation every cycle. The FPU receives 2 loads from the load/store unit

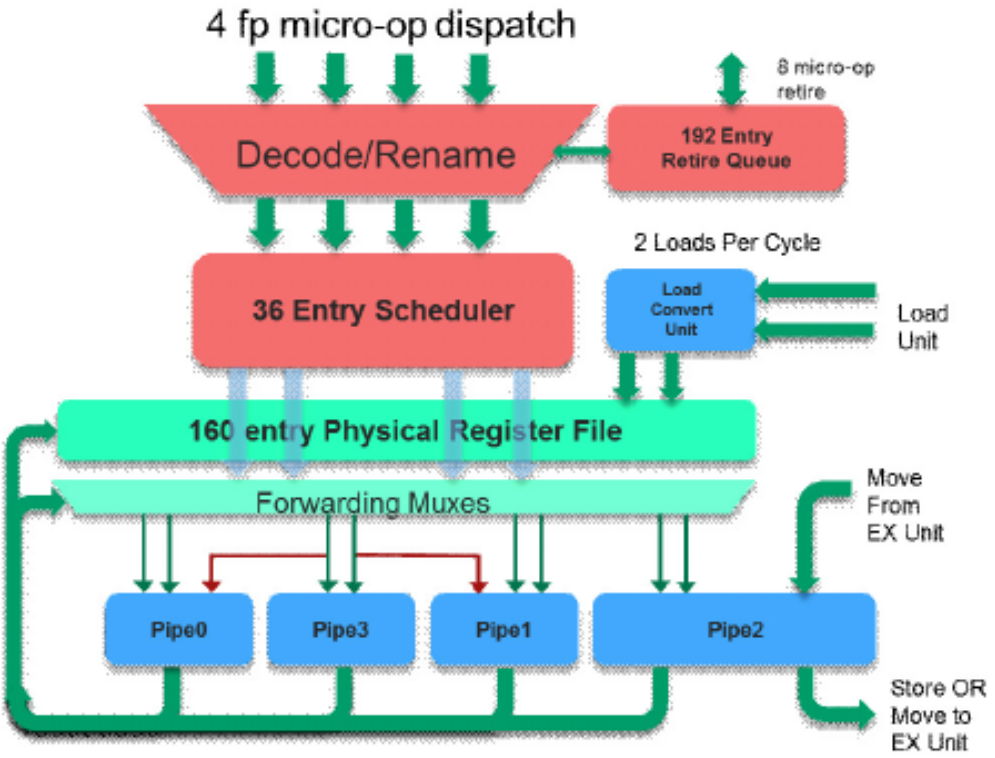

Figure 1: Floating-Point Unit Block Diagram

every cycle that are up to 128 bits each. Pipes 0 and 1 support operations that require three operands like fused-multiply add (FMA). So optimal code should operate on 128 bit (XMM register) or 256-bit (YMM registers) with every operation using the SIMD instructions which has high throughput.

2.1 TRSM

Consider the problem of computing a solution $X \in \Re^{n \times q}$ to $LX = B$ where $L \in \Re^{n \times n}$ is lower triangular and $B \in \Re^{n \times q}$. This problem is called *multiple right hand side* forward substitution problem. In a more general case its called *triangular solve matrix (trsm)* [4]. This problem can be solved by block-based algorithms that are rich in matrix-matrix multiplication assuming that q and n are large enough. To develop a block forward substitution algorithm, consider a system of equations, $LX = B$, where L is lower trian-

Unit	Pipe				Domain[2]	Ops Supported
	0	1	2	3		
FMUL	X	X			F	(v)FMUL*, (v)FMA*, Floating Point Compares, Blendv(DQ)
FADD			X	X	F	(v)FADD*
FCVT				X	F	All convert operations except pack/unpack
FDIV[1]				X	F	All Divide and Square Root except Reciprocal Approximation
FMISC	X	X	X	X	F	Moves and Logical operations on Floating Point Data Types
STORE			X		S	Stores and Move to General Register (EX) Operations
VADD[2]	X	X		X	I	Integer Adds, Subtracts, and Compares
VMUL	X				I	Integer Multiplies, SAD, Blendvb
VSHUF[3]		X	X		I	Data Shuffles, Packs, Unpacks, Permute
VSHIFT			X		I	Bit Shift Left/Right operations
VMISC	X	X	X	X	I	Moves and Logical operations on Packed Integer Data Types
AES	X	X			I	*AES*
CLM		S				*CLM*

Figure 2: Floating-Point Execution Resources

gular matrix.

$$\begin{bmatrix} L_{1,1} & 0 & \cdots & 0 \\ L_{2,1} & L_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ L_{N,1} & L_{N,2} & \cdots & L_{N,N} \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_N \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_N \end{bmatrix} \tag{1}$$

Once we solve the system $L_{11}X_1 = B_1$ for X_1, we then remove X_1 from block equation 1 through N:

$$\begin{bmatrix} L_{2,2} & 0 & \cdots & 0 \\ L_{3,2} & L_{3,3} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ L_{N,2} & L_{N,3} & \cdots & L_{N,N} \end{bmatrix} \begin{bmatrix} X_2 \\ X_3 \\ \vdots \\ X_N \end{bmatrix} = \begin{bmatrix} B_2 - L_{2,1}X_1 \\ B_3 - L_{3,1}X_1 \\ \vdots \\ B_N - L_{N,1}X_1 \end{bmatrix} \tag{2}$$

Continuing in this way we obtain the following block *axpy* forward elimination algorithm:

Algorithm 1 Block axpy Forward Elimination

for $j \leftarrow 1 : N$ **do**
  Solve $L_{jj}X_j = B_j$
  for $i \leftarrow j+1 : N$ **do**
    $B(i) \leftarrow B(i) - L_{ij}X_j$
  end for
end for

Notice that the i-loop corresponds to a single block *axpy* update of the form

$$\begin{bmatrix} B_{j+1} \\ \vdots \\ B_N \end{bmatrix} = \begin{bmatrix} B_{j+1} \\ \vdots \\ B_N \end{bmatrix} - \begin{bmatrix} L_{j+1j} \\ \vdots \\ L_{Nj} \end{bmatrix} X_j \tag{3}$$

For a given architecture this is handled as rank-k update (GEMM) for sufficiently large X_j.

2.1.1 TRSM Optimization

The equation $L_{11}X_1 = B_1$ is a trsm sub-problem. The equation can be recursively solved by breaking the sub problem into smaller sub-blocks and solving for each sub-block. In BLIS the smallest sub-block is decided based on architecture and the data type of the matrix elements. For "ZEN"core the block sizes are 6×16 (single precision) and 6×8 (double precision). At the smallest sub-block, unblocked variant of trsm is implemented as micro-kernels in X86_64 AVX2 assembly.
The kernels can be found in the file `bli_strsm_l_int_6x16.c` and the function names are `bli_strsm_l_int_6x16` for single precision and `bli_dtrsm_l_int_6x8` for double precision. To exploit cache spatial locality, there are packed routines provided by the BLIS framework to pack the data from the original matrix buffers into continuous memory blocks. These kernels directly operate on these continuous chunk of blocks. The important function arguments of these micro kernels are:

- float/double* $a11$
- float/double* $b11$
- float/double* $c11$
- int rs_c (row-stride of $c11$)

- int cs_c (column-stride of $c11$)

Here $a11$ is a packed block derived from L_{11}. During the packing process the diagonal elements α_{ii} of L_{11} are stored as $1/\alpha_{ii}$ in $a11$ to avoid costly division operation. The matrix $b11$ is temporary packed block derived from $B1$ and $c11$ is actually the solution to this trsm problem and it points to the block location of the original buffer $B1$. Hence elements of $c11$ are accessed through the strides rs_c and cs_c. Lets say the elements of matrices are represented as:

$$c_{11} = \begin{bmatrix} \gamma_{0,0} & \gamma_{0,1} & \cdots & \gamma_{0,7} \\ \gamma_{1,0} & \gamma_{1,1} & \cdots & \gamma_{1,7} \\ \vdots & \vdots & \ddots & \vdots \\ \gamma_{5,0} & \gamma_{5,1} & \cdots & \gamma_{5,7} \end{bmatrix}_{6\times 8} \quad a_{11} = \begin{bmatrix} \alpha_{0,0} & 0 & \cdots & 0 \\ \alpha_{1,0} & \alpha_{1,1} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{5,0} & \alpha_{5,1} & \cdots & \alpha_{5,5} \end{bmatrix}_{6\times 6}$$

$$b_{11} = \begin{bmatrix} \beta_{0,0} & \beta_{0,1} & \cdots & \beta_{0,7} \\ \beta_{1,0} & \beta_{1,1} & \cdots & \beta_{1,7} \\ \vdots & \vdots & \ddots & \vdots \\ \beta_{7,0} & \beta_{7,1} & \cdots & \beta_{7,7} \end{bmatrix}_{6\times 8}$$

Let $a(i)$ represent i^{th} row of $a11$, $b(i)$ represent i^{th} row of $b11$ and $c(i)$ represent i^{th} row of $c11$. Each 256-bit register can store upto 8 floats or 4 doubles. To process each row, two 256-bit YMM registers are used because

Algorithm 2 TRSM Microkernel

1: $b(0) \leftarrow b(0) * \alpha_{00}$
2: $c(0) \leftarrow b(0)$
3: **for** $i \leftarrow 1 : 5$ **do**
4: $\quad b(i) \leftarrow \left(b_i - \sum_{k=1}^{i-1} \alpha_{ik} b(k)\right) * \alpha_{i,i}$
5: $\quad c(i) \leftarrow b(i)$
6: **end for**

each register can load upto 8 floats or 4 doubles. Each $\alpha's$ broadcast into YMM registers. The full algorithm is mentioned in algorithm 2. The *for* loop in algorithm 2 is unrolled. The multiplication and addition operation mentioned in algorithm 2 is carried out by AVX2 instructions. The division operation by $\alpha_{i,i}$ is replaced by multiplication in line 4 of algorithm 2, since $\alpha_{i,i}$ is stored as $1/\alpha_{i,i}$ in $a11$ during packing, this is done to avoid costly division operation.

TRSM is further optimized by combining GEMM (rank-k update)(refer algorithm 3) and trsm microkernel algorithm 2 into a single kernel, we call

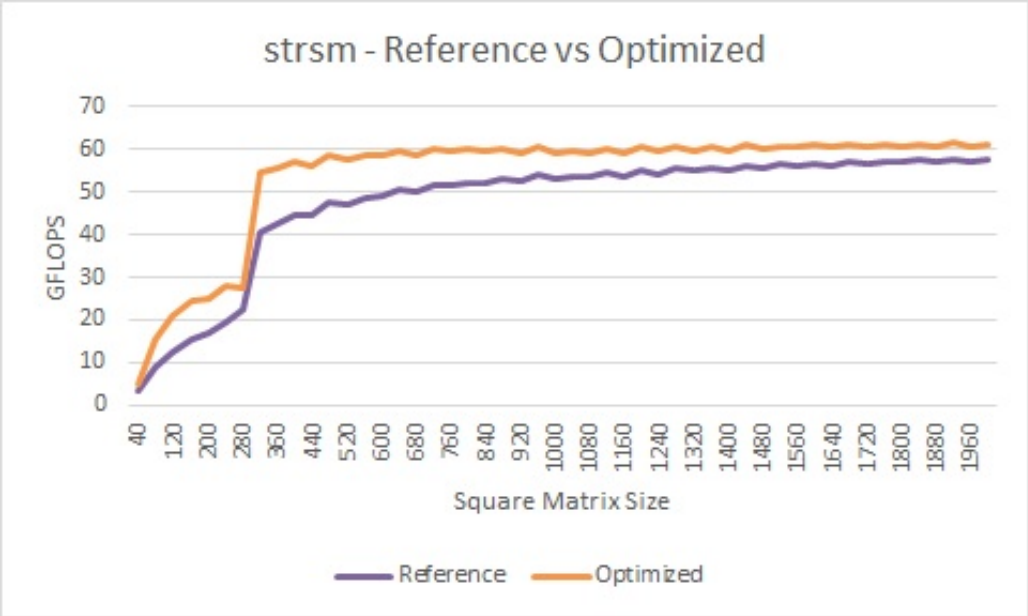

Figure 3: Single Precision Performance comparison of GEMMTRSM: Reference vs Optimized

this kernel as *gemmtrsm*. The advantage of doing this is we avoid memory read and write operations by storing the temporary result of *blockaxpy* in registers. The optimized microkernel is around 18.4% faster.

2.2 gemv

This operation performs:

$$y = \beta * y + \alpha * A * x \tag{4}$$

Where A is $m * n$ matrix, y and x are vectors. In BLIS *gemv* is implemented using fused microkernels. For column major storage, *gemv* uses *axpyf* and for row major storage it uses *dotxf* operations.

axpyf performs and fuses some implementation-dependent number of *axpyv* operations, accumulating to the same output vector. It can also be expressed as a *gemv* operation where matrix A is $m * b_{fuse}$, where b_{fuse} is the number of fused operations (fusing factor).

dotxf performs and fuses some implementation-dependent number of dot-product (*dotxv*) operations, reusing the y vector for each *dotxv*. For column-major storage, *gemv* is implemented using *axpyf*. For row-major storage, it is implemented using *dotxf*.

Algorithm 3 GEMV Column Storage

A : matrix of size $m * n$
x, y : vectors of length n
$b_{fuse} \leftarrow fusingfactor$
$n_{elem} \leftarrow m$, $n_{iter} \leftarrow n$
cs_a : column stride of A
$y \leftarrow \beta * y$ // scalv operation
for $i \leftarrow 0$; $i < n_{iter}$; $i+ = f$ **do**
  $f \leftarrow min(b_{fuse}, (n_{iter} - i))$
  $A1 \leftarrow A + i * cs\_a$
  $x1 \leftarrow x + i$
  call $axpyf(n_{elem}, f, \alpha, A1, 1, cs\_a, x1, y1)$
end for

The performance graph of *gemv* for column storage is shown in the graph 6. For *gemv* optimization refer to algorithm 3 and for *axpyf* optimization refer to algorithm 4. The optimization for *dotxf* is similar to *axpyf*. For performance graphs refer to figures 4, 5 and 6. The optimized *gemv* is around 82.7% faster.

2.3 dotv

The *dotv* operation is defined as

$$\rho = \sum_{k=0}^{n-1} x[k] * y[k] \tag{5}$$

Here, x & y are vectors of length n. The *dotv* operation can be easily vectorized using AVX intrinsic. The performance of optimized *dotv* is shown in the graph 7. The *dotv* operation is unlikely to be auto-vectorized by the compiler, hence the optimized *dotv* leads to huge performance gains as shown in figure 7, an average improvement of 196% compared to reference $C - code$ implementation.

2.4 amaxv

The *amaxv* operation is defined as

$$(\alpha, i) = \max_{1 \leq j \leq n} (|x[j]|) \tag{6}$$

Here α is element of vector x which contains the maximum absolute value and i its index in the vector. It involves two operations, computing absolute

Algorithm 4 axpyf

$axpyf(n, f, \alpha, A1, 1, csa, x1, y1)$
$A1$: Matrix with leading dimension csa
$x1$, $y1$: vectors of length n
$f \leftarrow fusingfactor$
$\alpha v[f] \leftarrow$ load f elements of $x1$
$\alpha v[0 : f-1] *= \alpha$
$xv[f] \leftarrow$ an array of $256 - bit$ YMM registers
$av[f] \leftarrow$ an array of YMM registers to load $A1$ elements
$yv \leftarrow$256-bit YMM register to load $y1$
$xv[0 : f-1] \leftarrow$ broadcast $\alpha v[0 : f-1]$
$sw \leftarrow$ SIMD Width, 8 (Single Precision), 4 (double precision)
for $i \leftarrow 0$; $(i + sw - 1) < n$; i += sw **do**
  $yv \leftarrow load\ y1$
  for $j \leftarrow 0$; $j < f$; j++ **do**
    $av[j] \leftarrow$ load YMM from $(A1 + i + j * csa)$
  end for
  for $j \leftarrow 0$; $j < f$; j++ **do**
    $yv \leftarrow yv + av[j] * xv[j]$ // y += α* x
  end for
  $y1 \leftarrow yv$ // Store the output
  $y1 \leftarrow y1$ += sw
end for

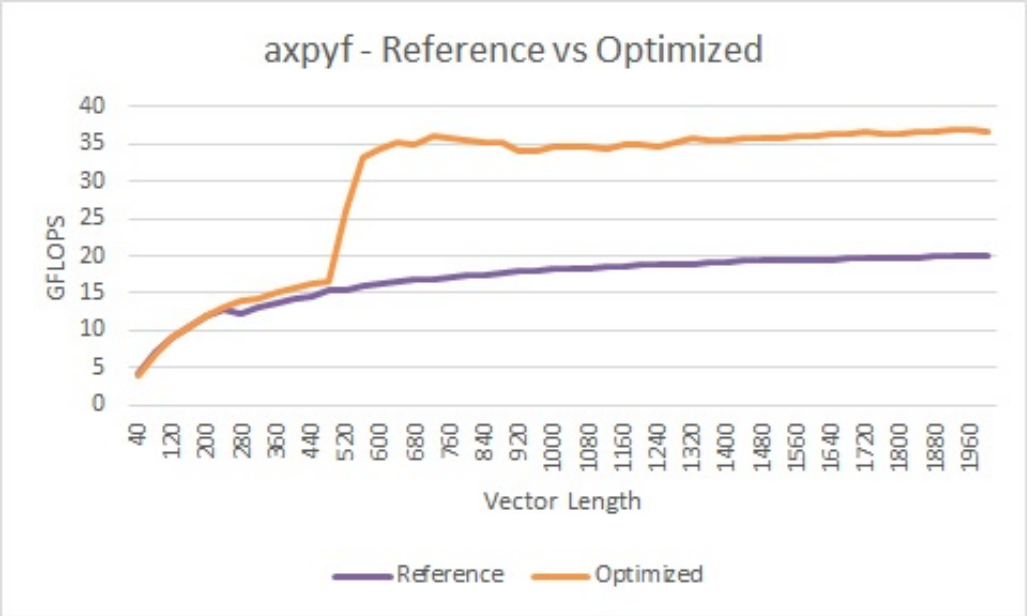

Figure 4: Single Precision Performance comparison of axpyf: Reference vs Optimized

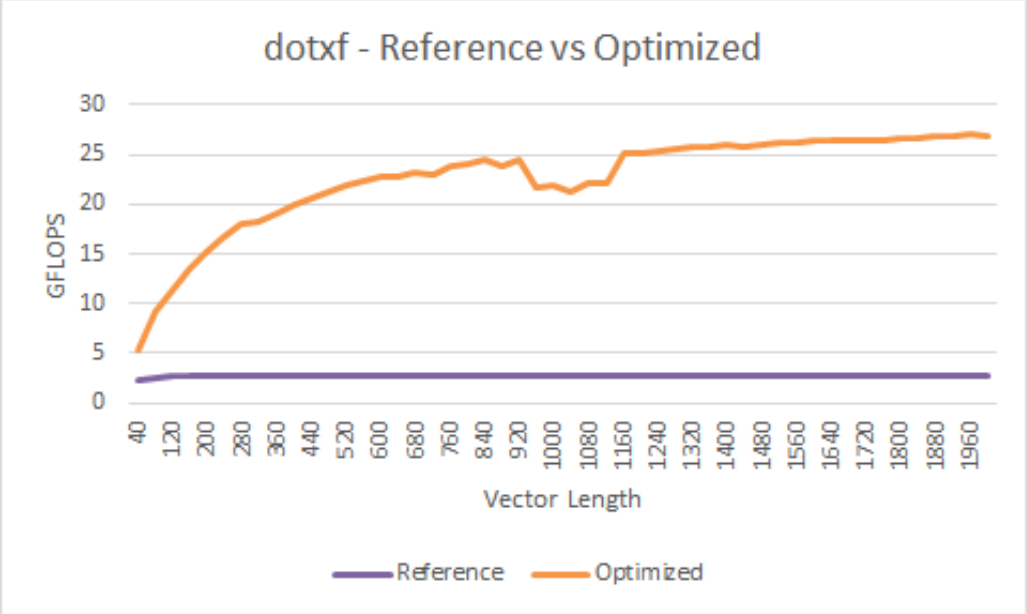

Figure 5: Single Precision Performance comparison of dotxf: Reference vs Optimized

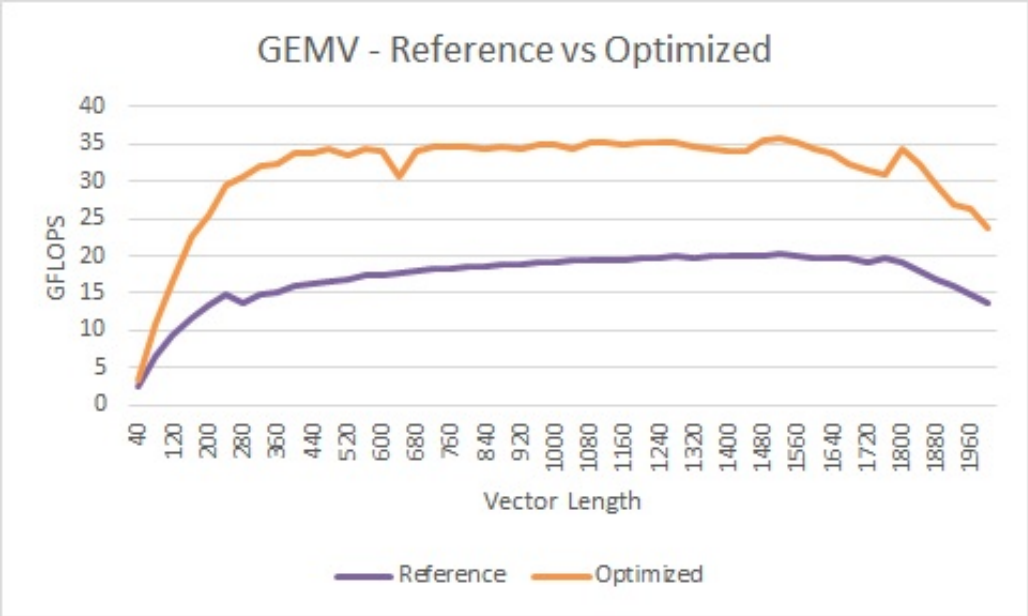

Figure 6: Single Precision Performance comparison of gemv: Reference vs Optimized

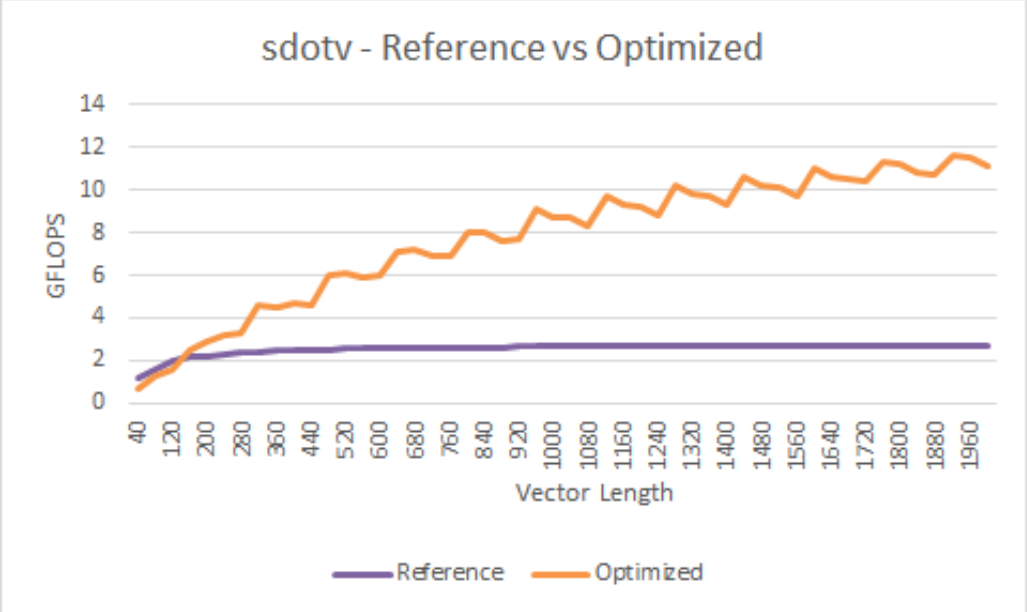

Figure 7: Single Precision Performance comparison of dotv: Reference vs Optimized

Algorithm 5 amaxv Microkernel

1: $x \leftarrow$ input vector of length n
2: $xv \leftarrow$ 256-bit YMM register to store elements of x
3: $signV \leftarrow \_mm256\_set1\_ps(-0.f)$ // to reset the sign-bit
4: $idxV \leftarrow \_mm256\_set1\_ps(7, 6, 5, 4, 3, 2, 1, 0)$ // stores the indexes of elements in a vector
5: $incV \leftarrow \_mm256\_set1\_ps(8)$ // offset to process next set of SIMD Width elements
6: $maxV \leftarrow \_mm256\_set1\_ps(-1)$ // Stores maximum absolute value of elements
7: $maxIdxV \leftarrow \_mm256\_setzero\_ps()$ // stores indexes of corresponding elements in $maxV$
8: **for** $i \leftarrow 0$; $(i + 7) < n$; $i+ = 8$ **do**
9: $xv \leftarrow$ load from x
10: $xv \leftarrow \_mm256\_andnot\_ps(signV, xv)$ // computes absolute value of elements in xv
11: $maskV \leftarrow \_mm256\_cmp\_ps(xv, maxV, \_CMP\_GT\_OS)$;
12: $maxV \leftarrow \_mm256\_blendv\_ps(maxV, xv, maskV)$ // element-wise max elements are stored in $maxV$
13: $maxIdxV \leftarrow \_mm256\_blendv\_ps(maxIdxV, idxV, maskV)$
14: $x \leftarrow (x + 8)$
15: $idxV \leftarrow (idxV + incV)$
16: **end for**
17: Find maximum element from the packed YMM register $maxV$ and corresponding max index from $maxIdxV$s
18: Repeat the same process for remaining elements

value of all elements and finding the maximum among them. The reference implementation to compute the absolute value of the element involves a branching operation, which multiplies the value of -1 with the input element when it is negative. This approach has several disadvantages like branch miss prediction overhead, multiplication of -1 with the negative floating number will utilize the floating point engine, hence more latency and the compiler cannot vectorize this operation due to branching. The performance of the optimized kernel is shown in figure 8. The optimized kernel is around 4.9X faster. Refer to algorithm 5 for optimization details of the *amaxv* kernel for single precision.

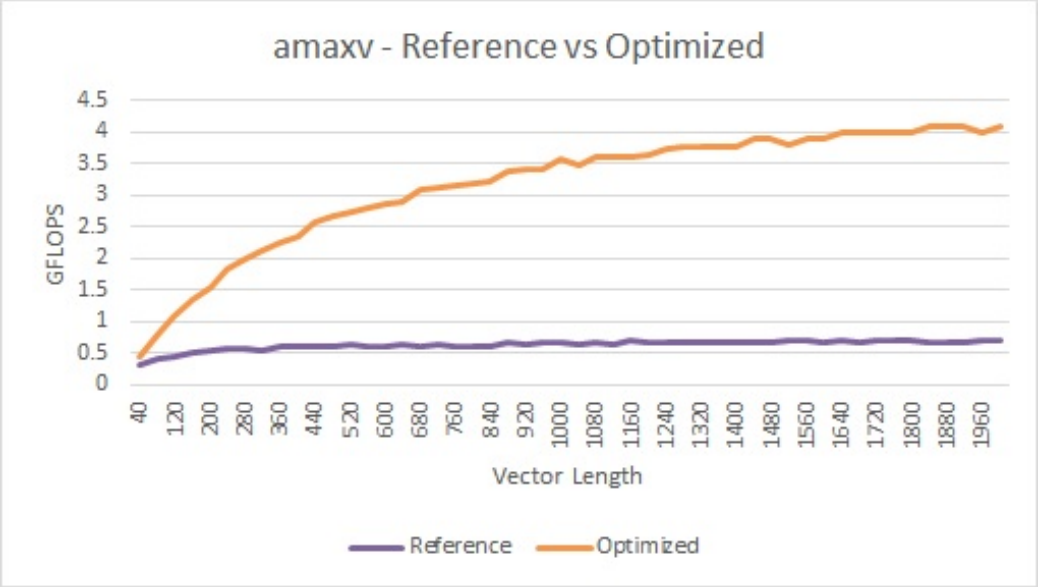

Figure 8: Single Precision Performance comparison of amaxv: Reference vs Optimized

References

[1] *https://github.com/flame/blis/wiki* .

[2] *http://www.netlib.org/blas/*.

[3] Advanced Micro Devices *Software Optimization Guide for AMD Family 17h Processors.* May 2016.

[4] Gene H. Golub and Charles F. Van Loan *Marix Computations 3rd edition* 1986: The Johns Hopkins University Press.

DISCLAIMER

The information contained herein is for informational purposes only, and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information. Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD's products are as set forth in a signed agreement between the parties or in AMD's Standard Terms and Conditions of Sale.

AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.