# An Optimized Matrix Multiplication on ARMv7 Architecture

[*1]Kun Feng, [2]Cheng Xu, [3]Wei Wang, [4]Zhibang Yang, [5]Zheng Tian
*[* 1, 2, 4, 5] Embedded System and Network Laboratory*
*College of Information Science and Engineering, Hunan University,*
*Changsha 410082, China,*
*[3]Faculty of Transportation and Information, Hainan College of Vocation and Technique,*
*Haikou 570216, China, E-mail: fkengun@gmail.com*

***Abstract***

*A sufficiently optimized matrix multiplication on embedded systems can facilitate data processing in high performance mobile measuring equipment since plenty of the kernel mathematical algorithms are based on matrix multiplication. In this paper, we propose a matrix multiplication specially optimized for ARMv7 architecture. The performance-critical differences between ARMv7 and conventional desktop/server architecture are considered to block the simple implementation. The Advanced-SIMD (Single Instruction Multiple Data) engine NEON is additionally exploited to increase the arithmetic computing performance and decrease the memory access latency. Experimental results demonstrate that the proposed scheme is 7-20 times faster than the simple implementation and superior to popular algorithm and open source libraries.*

**Keywords***: Embedded system, Optimization, Matrix Multiplication, ARMv7, NEON*

## 1. Introduction

Matrix multiplication is the most complex and time consuming task in basic matrix operations which is widely used in computer vision[1], scientific computing[2] and industrial measurement[3]. In processing of chemical component analysis, data matrices are processed by MATLAB on PC. Technicians copy the data from fluorescence spectrometer to PC and input them into matrix processing program. Such a routine, however, is not efficient and convenient enough for portable use. With the help of modern embedded technology, it's becoming both possible solution and industry requirement to build mobile and portable equipment to take the place of those ponderous PC based facilities in those areas. But present methods mainly emphasize particularly on optimizing its performance on desktop or server processors. Taking account of the significant differences between embedded and desktop or server processors as well as the current poor hardware utilization of embedded processors, a need arises to find an optimized matrix multiplication on embedded processors.

Because of the importance of the matrix operations in both industrial and academic areas such as linear algebra, pioneers started the optimization work long time ago. The optimization can be implemented both in the format of separate optimization or a matrix operation library with matrix multiplication inside. BLAS (Basic Linear Algebra Subprograms)[4] is a standard linear algebra library which is composed of three levels of matrix operations: vector-vector operation, matrix-vector operations and matrix-matrix operations. It was designed to abstract and normalize the basic matrix operations so that the portability and efficiency can be well reached on different architectures. After that, several optimized libraries were created following the prototypes defined by BLAS. Whaley proposed the open source project ATLAS (Automatically Tuned Linear Algebra Software) to auto-detect the architecture of the running environment and tried as many combinations of a variety of parameters as possible to find the best configuration of his algorithm[5][6]. Goto created the library GotoBLAS[7][8] in which he presented a new high-performance of the matrix-matrix multiplication by refining a model of architectures which are capable of multilevel memories[9]. Eigen[10] is a versatile, fast, reliable and elegant C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms. Meanwhile separate optimization can be found in [11]. Recently, researchers become more curious about using FPGA to accelerate the matrix multiplication[12][13]. But that will cost much more time and money to buy new hardware and learn to use new developing tools.

Aside from those scholars, the processor vendors themself may release a copy of library that optimized for linear algebra computation on its own products. We call it vendor-BLAS. Intel MKL and AMD APPML are such mathematic mathematical libraries optimized for Intel's and AMD's processors respectively. STI (Sony, Toshiba and IBM) launched the CELL/BE SDK to optimize numerical computations, which include matrix multiplication too, on its CELL BE processor. Different from general x86 architecture, CELL BE combines a general-purpose Power architecture core of modest performance with streamlined coprocessing elements which greatly accelerate multimedia and vector processing applications. Optimization work on it becomes popular these years[14][15].

So far, however, there are several problems those literatures share. First, none of the literature was found to be an optimization on embedded system. The development on those platforms, mainly ARM based, usually requires cross-compiling which makes most optimized libraries impossible to work. In all the mentioned work, only Eigen supports ARM platform. Second, except vendor-BLAS, most of those works focused on the general core side and ignored the specialized processing core. But there is no vendor-BLAS for ARM devices. Third, the characteristics of memory access must be considered, which can be different over platforms and optimized specifically. For all these reasons, we propose an optimized matrix multiplication on ARMv7 platform taking account of architecture feature of both general and specific cores and characteristics of memory access.

The rest of this paper is organized as follow. In Section 2, the original optimization scheme on desktop/server architecture is reviewed first. Then a series of evolutionary optimizing approach is introduced. The experimental results of the proposed optimization method and other related publications are included in Section 4. At last we give a conclusion of this work in the last section.

## 2. Original optimized matrix multiplication

The prototype of general matrix multiplication in BLAS can be described as follow:

$$C = alpha * A * B + beta * C \qquad (1)$$

The size of $A$, $B$ and $C$ is $m$ by $k$, $k$ by $n$ and $m$ by $n$, respectively. To simplify the evaluation, for the rest of this paper, $alpha$ and $beta$ are set to 1.

As we all know, the memory hierarchy inside a computer system can be briefly described as a pyramid shown in Figure 1(a). Traditional implementation in BLAS of matrix multiplication naively loads and stores data from and into RAM, the relatively slowest memory in the hierarchy. But it would be the ideal performance if we can accomplish the computation in the way of only accessing data from registers. However, considering the very few number of registers, typically 8~32, that is almost impossible. Unfortunately again, the cache is not large enough to hold the entire matrix when the matrix becomes huge. Accordingly, blocking is required to keep data that will be used in near future in cache to speed up the calculation. For the same reason, the scheme of blocking becomes the most significant component in the algorithm.

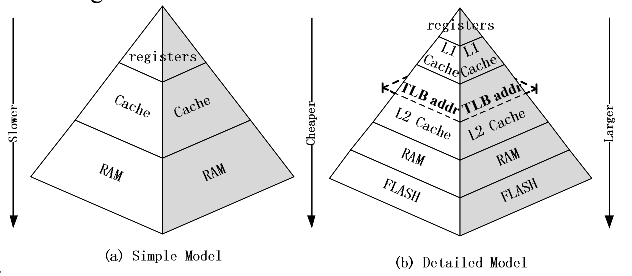


**Figure 1.** Memory hierarchy of a typical embedded system

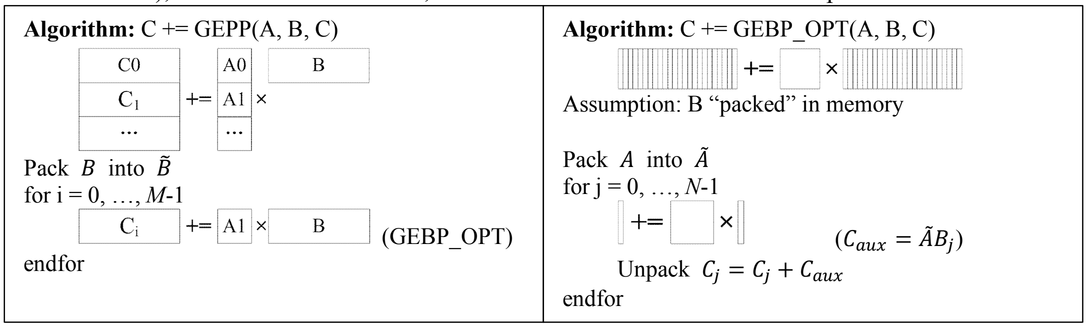
In GotoBLAS, Goto et al. extended work of Gunnels et al.. They have proposed six layered blocking schemes and, a step further, proved that one of them would be inherently superior over others and obtained the best performance. They called it a general matrix multiplication based on an optimized GEPP via calls to an optimized GEPB. The GEPP and GEBP parts of the scheme can be seen in Figure 2. The blocking was carried out at $k$, $m$, $n$ and $m$ dimension successively.

The most significant contribution of GotoBLAS is based on two experimental observations:

1) The elements of A should reside in L2 cache because of the relatively smaller ratio between the rate of floating point operations (flops) that can be performed by the floating point unit and the

rate of floating point numbers that can be streamed from the L2 cache.

2) Frequently, it is the size of the data that can be addressed by the TLB (Translation Lookaside Buffer), not the size of L2 cache, that limits the size of A that can be kept in L2 cache.



```
Algorithm: C += GEPP(A, B, C)
    C0   |  A0  |  B
    C1   += A1  ×
    ...  |  ...
Pack $B$ into $\tilde{B}$
for i = 0, …, $M$-1
    Ci += A1 × B    (GEBP_OPT)
endfor
```

```
Algorithm: C += GEBP_OPT(A, B, C)
    += ×
Assumption: B "packed" in memory

Pack $A$ into $\tilde{A}$
for j = 0, …, $N$-1
    += ×    ($C_{aux} = \tilde{A}B_j$)
    Unpack $C_j = C_j + C_{aux}$
endfor
```

**Figure 2.** Blocking scheme

To Goto's opinion, TLB has to be considered in the memory hierarchy which makes the pyramid like Figure 1(b). TLB miss will definitely stall CPU and then result in poor performance. Thus in addition to basic blocking scheme and cache layer selection, several considerations were proposed to improve the performance, like packing matrices into memory continuous layout to minimize TLB miss and arranging the packing carefully so that the memory access time can be hided in arithmetic computing. Afterwards, it is the key step to transform theoretical scheme to practical blocking size: $m_c$, $k_c$, $m_r$ and $n_r$, which represent the blocking size for cache and register at $m$, $k$ and $n$ dimension respectively. These parameters were decided according to constraints of all considerations.

## 3. Optimized matrix multiplication on ARMv7 architecture

Referring to mature studies on desktop/server platforms, three optimization schemes are proposed progressively to resolve deficiencies of previous researches on embedded system. At first inspired by GotoBLAS, optimization on general core side is optimized to be more suitable for ARMv7 architecture. After that, NEON is enabled to boost the data processing. An original optimization on memory access is added at last to enhance the performance a step further.

### 3.1 General core optimization

Although matrix multiplication optimization on desktop/server architecture is going to reach the peak, many things change when it comes to ARMv7. Since those two types of architectures are originally designed for completely different application scenario, desktop/server one for high-performance and ARM for low-power embedded application, there are several important factors in architecture that affect the final algorithm.

**Table 1.** Differences of two types of architectures (general core part)

| **Differences** | **Desktop/server architectures** | **ARMv7 architecture** |
|---|---|---|
| Execution of instructions | Out-of-order | In-order/Out-of-order |
| Replacement policy of TLB | LRU | Round-robin |
| Replacement policy of Cache | LRU | Random |

As we can see from Table 1, there are at least three functionalities that involve in the performance of the general core. We will discuss them one by one.

#### Execution of instructions

Almost all modern architectures for desktop/server application execute instructions out of order. Compared with that, most embedded architecture choose to implement the instruction scheduler in order to avoid the high complexity of circuit and high power consumption that out-of-order execution brings. The most popular processor in middle and high-end embedded application, the Cortex-A8 which implements the ARMv7 architecture, is a typical in-order-execution processor. Although the next generation of ARMv7 based processor, the Cortex-A9, is an out-of-order processor, Cortex-A8 is still the mainstream of market and worthy of paying special attention to improve its performance.

In out-of-order processor, instruction scheduler will not blindly issue the first instruction in instruction queue into pipeline or keep on waiting for unsatisfied instruction or data dependence as in-order processor. On the contrary, it will issue the instruction that has met all requirements, which is probably not the first instruction in queue. Consequently, the order of instructions is much more vital for performance of in-order processors. Instructions sequence should be carefully adjusted by hand. We will show more about it later.

### Replacement policy of TLB

TLB plays an influential role in GotoBLAS. The size of data of $A$ is limited by the size of space that can be addressed by TLB entries. The original constraint is as follow:

$$T_A + 2\left(T_{B_j} + T_{C_j}\right) \leq T \tag{2}$$

$T_A$, $T_{B_j}$ and $T_{C_j}$ are the number of TLB entries for data of $A$, $B_j$, $C_j$ respectively. $T$ is the total number of all entries in TLB. The factor two is the result of the LRU (Least Recently Used) replacement policy of TLB. In GotoBLAS, the size of $A$ in L2 cache is maximized to avoid TLB miss. In order to keep the entries for data of $A$ $(T_A)$ in TLB, two columns of B and $C$, $B_j$, $C_j$ and $B_{j+1}$, $C_{j+1}$are cached to make $T_{B_j}$ and $T_{C_j}$ the least recently used entries and be swapped out instead of $T_A$.

However, the replacement policy in ARMv7 architecture is not LRU but Round-robin, which uses a counter to indicate the index of the next TLB entry added in. Therefore, we cannot control which entry is replaced when TLB is full. The factor two can be omitted so that our constraint on size of $A$ is:

$$T_A + \left(T_{B_j} + T_{C_j}\right) \leq T \tag{3}$$

Without the factor two, $T_A$ becomes larger and size of $A$ that can reside in L2 cache increases which benefits the overall performance. And $k_c$ increases consequently since it's decided by size of $A$.

### Replacement policy of cache

Similar to Goto's consideration of replacement policy of TLB, Whaley et al added a factor two in their algorithm to keep $B_j$ and $C_j$ the least recently used cache line then replaced by $B_{j+2}$ and $C_{j+2}$ so that the cache line for $A$ is always kept in cache. ARM chooses to implement the random replacement policy to simplify the circuit design for its approximate performance as LRU but much smaller die size. So we can increase the size of $A$ again by ignoring the effect of replacement policy of cache just like what we have done in TLB consideration.

## 3.2 NEON optimized data processing

Besides the differences in general core side, NEON was added into ARM architecture in ARMv7 as a SIMD engine just as other commercial product like SSE and AVX from Intel, AltiVec from IBM and SPE in CellBE etc. It provides standardized acceleration for media and signal processing applications.

In addition to its original purpose, NEON can be used to accelerate the matrix multiplication to meet the requirement of high performance embedded computing. NEON is capable of pipelined integer and floating point arithmetical operations. With the help of separate floating point pipeline, NEON can accomplish four single precision or two double precision floating point multiplication or addition in one instruction.

There are 32 64bit registers in standard Cortex-A8 and Cortex-A9 processor. Each of them can store two single precision floating point operators or one double precision floating point operator. Instruction can be written in 128bit and executed in 64bit width (128bit in enhanced architecture Scorpion from Qualcomm). Since the double precision operations have a poor performance in ARMv7 and cannot be pipelined, we focus our attention mainly in matrix multiplication in single precision level.

By default, all single precision operations will be complied into VFP (Vector Floating Point) instructions in ARMv7 architecture, which is not pipelined in Cortex-A8. To take advantage of NEON, programmers can add the compiling options "--with-fpu=neon --with-float=softfp" to enable auto-vectorization, which can, to some extent, compile some operations into NEON instructions. However, when it comes to a complex program like code statements with more than two nested loops and high data dependence, such as matrix multiplication, auto-vectorization becomes helpless. Because of that, we implemented the multiplication kernel in NEON assembly code directly.

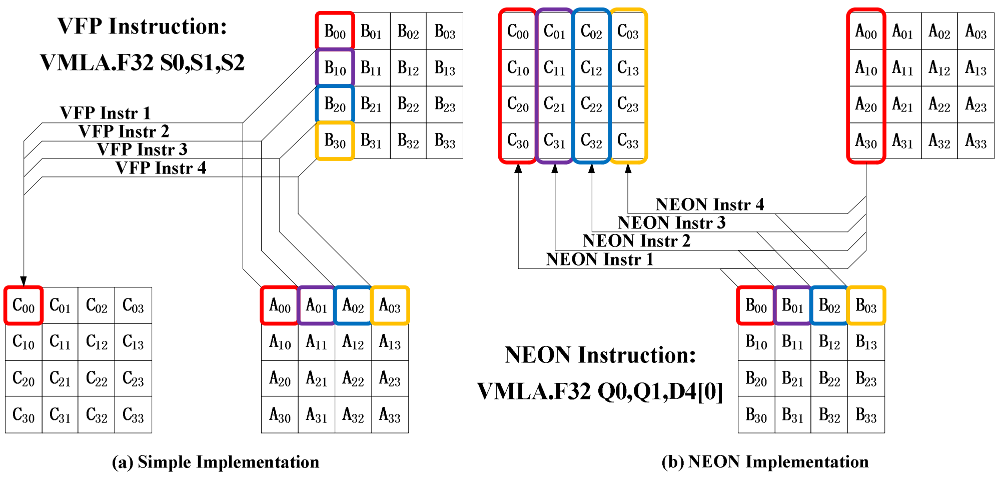


**Figure 3.** Comparison between NEON and simple matrix multiplication

In the proposed algorithm, the whole multiplication can obtain the best performance only when we meet the prerequisite that the kernel can get the peak efficiency. So the kernel is carefully scheduled by hand to take fully usage of the pipeline especially in Cortex-A8.

Figure 3 shows a comparison of parts of two implementations of the kernel which multiplies two $4 \times 4$ matrices. The real kernel processes matrices A and B at the size of $4 \times k_c$ and $k_c \times 4$ respectively, which means $m_r$ and $n_r$ are set to 4. In the data processing stage shown above, the multiplication and addition don't execute row by row and column by column as in simple matrix multiplication (Figure 3(a)). Instead, the processing in NEON implementation (Figure 3(b)) is vectorized so that every element of $C$ is calculated in four steps and every step is a vector calculation. On the contrary, the simple implemented one calculates the result in VFP not NEON since $S$ registers can only be accessed by VFP. What's more, those steps in NEON implementation are separated so that NEON instruction 1-4 doesn't stall each other. When the first four NEON instructions finish and the loop goes to the second column of $A$ and the second row of $B$, the register which stores the first column of $C$ is available. As a result of that, the whole kernel has no stall at all and the performance is maximized.

It can be easily proved that the kernel is lean enough without executing redundant instructions. The computation in Figure 3, for example, requires 128 FLOPs. Every NEON VMLA instruction accomplishes two FLOPs (multiplication and accumulation) on four single precision floating-point operators. 16 NEON VMLA instructions all together fulfill exactly 128 FLOPs in total.

We call this NEON accelerated matrix multiplication ARMM-NEON for the rest of this paper.

## 3.3 NEON optimized memory access

In the proposed algorithm, data needs to be copied so that CPU can access it continuously. Traditionally, it is achieved by calling the memcpy function in standard library. Nevertheless, that is for general purpose and never optimized for any architecture. At the same time, there are two sets of load/store instructions in ARMv7 architecture, one for general core and the other for NEON. It has been proved officially by ARM that memory copy implemented in NEON with PLD (preload instruction) is about 50% faster than the general word-by-word one. Moreover, the data bandwidth between L1 cache and registers for NEON (8 bytes/cycle) is twice as that of general core (4 bytes/cycle). Based on those benefits, we decide to replace the standard memcpy function with the NEON optimized memory copy to earn better performance.

In order to distinguish this further optimized matrix multiplication with the previous one, it is named ARMM-NEON-MAOPT (ARMM-NEON with Memory Access Optimized).

## 4. Experiment and evaluation

The experiments are designed on three different platforms: TI OMAP3630, Qualcomm MSM8260 and AMLogic AM8726-M all of which contain an implementation of ARMv7 architecture. The key specifications and environments are listed as Table 2.

**Table 2.** Comparison of platforms

| Comparison | OMAP3630 | AML8726-M | MSM8260 |
|---|---|---|---|
| Type of processor | Cortex-A8 | Cortex-A9 | Scorpion |
| Number of cores | 1 | 1 | 2[1] |
| Frequency(MHz) | 800 | 800 | 1500[2] |
| L2 cache size(KBytes) | 256 | 128 | 256*2 |
| Data width of NEON(bit) | 64 | 64 | 128 |
| Pipelined VFP | No | Yes | Yes |
| Out-of-order execution | No | Yes | Partly |

1 Only one core in MSM8260 is used. 2 The frequency of MSM8260 is locked at 800MHz in experiments.

In the experiments, we compare the proposed optimization mainly with Eigen, the only library that supports NEON. The GotoBLAS is modified by us since it doesn't support ARM architecture. Hence the performance of GotoBLAS is an approximate evaluation. In order to evaluate the improvement discussed in Section 3.1, we add an imaginary competitor GotoBLAS-NEON which is a preliminarily NEON optimized GotoBLAS. The critical parameters $m_c$, $k_c$, $m_r$ and $n_r$ are firstly constrained by architecture parameters and then empirically tuned. According to Section 3.1, the upper limit of $m_c$ and $k_c$ is set to 256 and 64. Then we realize that the best performance can be reached when the total elements of $A$, $B$ and $C$ takes about half L2 cache and elements of $A$ takes about 8 TLB entries.

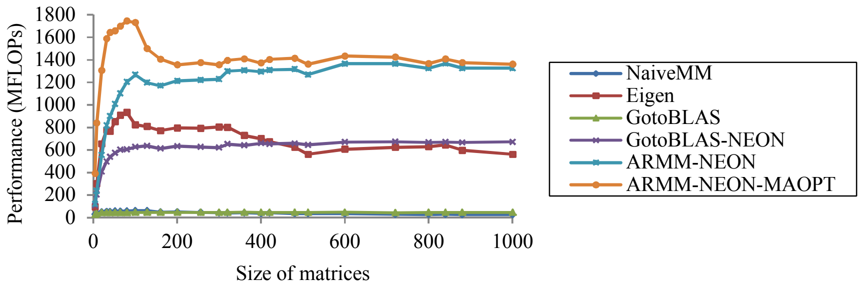


**Figure 4.** Performance on OMAP3630 platform

First on OMAP3630 platform (Figure 4), since the non-pipelined VFP in its architecture, the naively implemented matrix multiplication gets a terrible performance. Lacking of the support of NEON, GotoBLAS doesn't show a significant improvement towards the naïve implementation either. Because of the tiny difference between naïve implementation and GotoBLAS, there is no need to show the performance of ARMM, which is probably same as GotoBLAS. When compared with Eigen which supports NEON as well, the blocking scheme provides ARMM-NEON a steady and much better performance even when the size of matrix increases up to 1000. Specifically, the $k_c$ in ARMM-NEON is 256 while it is 200 in GotoBLAS. Larger $k_c$ which is generated by the enrichment in optimization for TLB and cache replacement consequently results in better performance. ARMM-NEON doubles the performance of the imaginary GotoBLAS-NEON, which proves that the proposed optimization for general core side of ARMv7 architecture is fairly effective. A little bit more, the memory access optimized ARMM-NEON (ARMM-NEON-MAOPT) is about 8.5% better than ARMM-NEON when the size of matrices is larger than 100. The excellent performance of ARMM-NEON-MAOPT for small matrices is the result of the dominant memory access latency in overall time. So the performance is much better than average value when the data is preloaded.

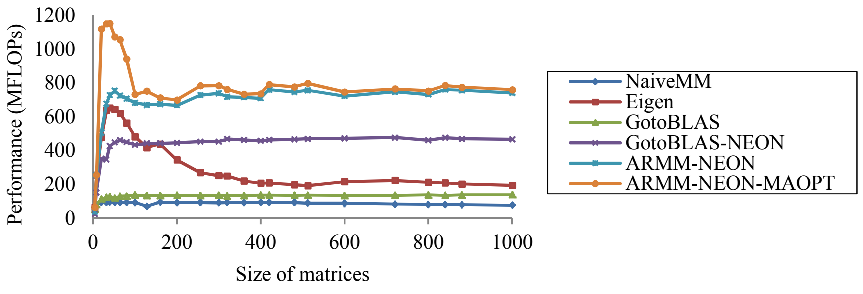


**Figure 5.** Performance on AML8726-M platform

When it comes to AML8726-M (Figure 5), the NaïveMM shows a much better performance because of the pipelined VFP. However, all optimized versions provide much better performance. Similar to situation on OMAP3630, ARMM-NEON and ARMM-NEON-MAOPT are superior to Eigen and GotoBLAS. On account of its out-of-order execution support, the gap between GotoBLAS and the proposed optimizations becomes smaller. The consideration of instruction sequence is weakened by smarter work of instruction scheduler in Cortex-A9. The performance of Eigen decreases with the size of matrices increasing, which is probably caused by the relatively small L2 cache.

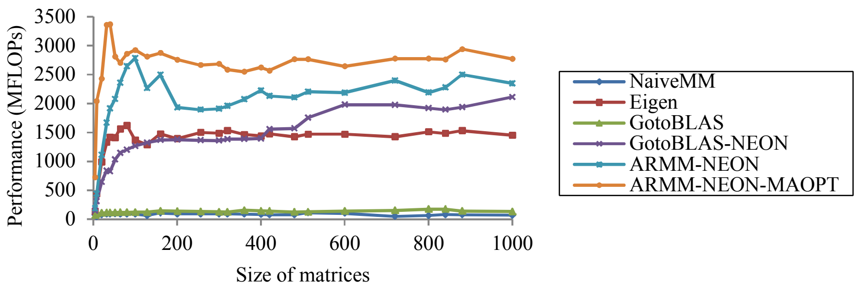


**Figure 6.** Performance on MSM8260 platform

The refined Cortex-A8 architecture, the Scorpion from Qualcomm, shows us an amazing performance. As we can see in Figure 6, double instruction execution width boosts the performance more than two times higher. For the same reason, the improvement that memory access optimization brings becomes larger. In contrast to AML8726-M, the normal L2 cache size (256KB) keeps the performance of Eigen steady even when the size of matrices increases to 1000. As the other two platforms, the proposed scheme still stands out in all algorithms.

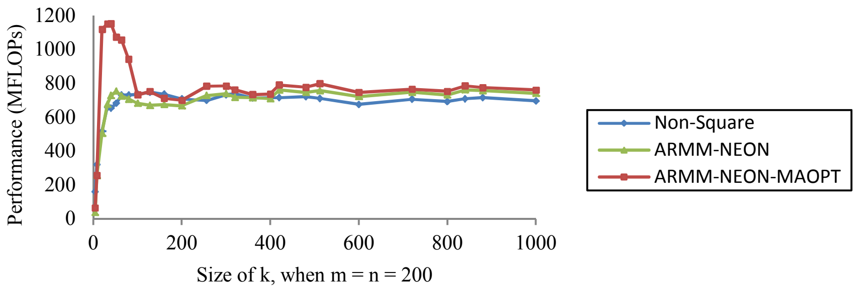


**Figure 7.** Performance of non-square matrices on AML8726-M platform

Figure 7 shows the performance when the matrices are non-square. We can see a similar trend with performance of square matrices multiplication. This non-square multiplication is also optimized in memory access. But it doesn't show a huge improvement as ARMM-NEON-MAOPT when the matrices are small because the blocking scheme for non-square matrices makes the $k_c$ too small so that the improvement of preloading is diminished.

## 5. Conclusion

So far there is only one optimized matrix multiplication for embedded systems. In this paper, a series of optimizations are proposed specifically for ARMv7 architecture. Both general and specific cores are utilized to improve the performance. Quantitative and qualitative experiments on different platforms show that the proposed methods are superior to other optimization schemes and libraries.

## 6. Acknowledge

This work is funded by Scientific Innovation Research Foundation of Hunan Province of China (Grant No. 2010TT1002), Natural Science Foundation of China (Grant No.60973030), and Fundamental Research Funds for the Central Universities (Grant No.531107040421). These financial supports are gratefully acknowledged.

## 7. References

[1] Qiu Chen, Koji Kotani, Feifei Lee, Tadahiro Ohmi, “Face Recognition Using VQ Histogram in Compressed DCT Domain”, Journal of Convergence Information Technology, vol. 7, no. 1, pp. 395-404, 2012
[2] Rong Hu, Weihong Xu, Fangjun Kuang, “An Improved Incremental Singular Value Decomposition”, International Journal of Advancements in Computing Technology, vol. 4, no. 2, pp. 95-102, 2012
[3] Lingyun Xu, Xiaofei Zhang, Zongze Xu, "Novel Blind Joint 2D Direction of Arrival and Frequency Estimation with L-shaped Array", JDCTA: International Journal of Digital Content Technology and its Applications, vol. 5, no. 9, pp. 194-202, 2011
[4] Charles L Lawson, Richard J Hanson, David Ronald Kincaid, and Fred T Krogh, "Basic Linear Algebra Subprograms for FORTRAN usage", Journal of ACM Transactions on Mathematical Software, vol. 5, no. 3, pp. 308-323, 1979.
[5] Richard Clint Whaley, Jack J. Dongarra, "Automatically Tuned Linear Algebra Software", In Proceeding of IEEE/ACM Conference on Supercomputing, pp. 1-27, 1998.
[6] Richard Clint Whaley, Antoine Petitet, Jack J. Dongarra, "Automated Empirical Optimizations of Software and the ATLAS project". Journal of Parallel Computing, vol. 27, no. 1-2, pp. 3-35, 2001.
[7] Kazushige Goto, Robert van de Geijn, "On Reducing TLB Misses in Matrix Multiplication", The University of Texas at Austin, Working Note 9, 2002.
[8] Kazushige Goto, Robert A. van de Geijn, "Anatomy of high-performance matrix multiplication", Journal of ACM Transactions on Mathematical Software, vol. 34, no. 3, pp.1-25, 2008.
[9] John A. Gunnels, Greg M. Henry, Robert A. van de Geijn, "A Family of High-Performance Matrix Multiplication Algorithms", In Proceedings of the International Conference on Computational Sciences-Part I, pp. 51-60, 2001.
[10] Eigen, “Eigen”, http://eigen.tuxfamily.org, 2012
[11] Aberdeen, Douglas, Baxter, Jonathan, "Emmerald: a fast matrix–matrix multiply using Intel's SSE instructions", In Concurrency and Computation: Practice and Experience, 2001.
[12] Rasha El-Atfy, Mohamed A Dessouky, Hassan El-Ghitani, "Accelerating Matrix Multiplication on FPGAs", In Proceeding of International Design and Test Workshop, pp. 203-204, 2007.
[13] Vinay B Y Kumar, Siddharth Joshi, Sachin B. Patkar, H. Narayanan, "FPGA Based High Performance Double-Precision Matrix Multiplication", In 22nd International Conference on VLSI Design, pp. 341-346, 2009.
[14] Jakub Kurzak, Wesley Alvaro, "Optimizing matrix multiplication for a short-vector SIMD architecture - CELL processor.", Journal of Parallel Computing, vol. 35, no. 3, pp. 138-150, 2009.
[15] Kazuya Matsumoto, Stanislav G Sedukhin, "Matrix Multiply-Add in Min-plus Algebra on a Short-Vector SIMD Processor of Cell/B.E.", In Proceedings of International Conference on Networking and Computing, pp.272-274, 2010.