

Scalable Parallel Multiplication of Big Matrices

Yaniv Sapir Adapteva, Inc.
Email: yaniv@adapteva.com

Introduction

Multiplication of matrices is a fundamental operation in linear algebra and is applied in many engineering and scientific areas. The popular SGEMM function is an example of a standard API used for implementing this operation in its most general form. It is not uncommon to encounter products of very large matrices. For example, in problems solved in finite elements methods, relationships between the model elements are expressed as elements of the state matrices. Stimuli to the state are expressed as input matrices or vectors and the new state can be calculated.

In a previous paper [1] we demonstrated how the Epiphany multicore processor can perform the $N \times N$ multiplication when written as a single-thread, pure standard C program. The compiled C program achieved about 70% of the peak theoretical performance. Another implementation, written in native ASM, achieved a ratio of about 90% of the peak theoretical performance. These benchmark implementations were limited by the amount of per-core on-chip memory to relatively small matrices (up to 45×45).

In this paper we will present a multi-core highly parallel implementation of matrix multiplication of arbitrary size. In order to implement this operation, we used the single-thread C multiplication mentioned above as a building block for the larger operation.

Block Matrix Multiplication

From obvious reasons, when the matrix operands are large enough, data has to be retrieved from the external memory (e.g., system's DRAM). In order to generate optimal code, it is important to minimize the accesses to the (relatively) slow DRAM. So, when designing the algorithm, we should try to reuse the loaded data as much as possible.

In the following discussion we will consider square matrices of size $N \times N$, but the same principles are applicable to non-square matrices. The multiplication of two matrices A and B gives the product C according to the following equation:

$$(1) \quad C = A \cdot B \rightarrow c_{ij} = \sum_{k=1}^N a_{ik} \cdot b_{kj}$$

When examining the above equation, one can deduce that the product of two matrices can also be represented as the multiplication of two *block matrices*. A block matrix is a matrix whose elements are

themselves matrices. All blocks in a row have the same number of rows and all blocks in a column have the same number of columns. Thus, if A is a block matrix of the form:

$$(2) \quad A = \begin{pmatrix} \tilde{A}_{11} & \tilde{A}_{12} & \cdots & \tilde{A}_{1M} \\ \tilde{A}_{21} & \tilde{A}_{22} & \cdots & \tilde{A}_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ \tilde{A}_{M1} & \tilde{A}_{M2} & \cdots & \tilde{A}_{MM} \end{pmatrix}, \text{ (} B \text{ and } C \text{ matrices would be similar)}$$

Then the product of two block matrices can be calculated according to the equation:

$$(3) \quad \tilde{C}_{pq} = \sum_{r=1}^M \tilde{A}_{pr} \cdot \tilde{B}_{rq}$$

This result suggests that the block matrix multiplication can be calculated recursively, where at each stage smaller and smaller blocks are being multiplied to produce intermediate block results, until we reach the deepest level, where the blocks are actually scalars. At this point we are back at the regular matrix multiplication case.

Implementation

Implementing this algorithm on the Epiphany based system EMEK3 was done in three levels:

- 1) At the 1st level is the fundamental per-core matmul operation. At this level we multiply blocks that fit inside a single core's SRAM, of size $H \times H$.
- 2) The 2nd level is the per-chip matmul. In this level, we use the per-core blocks to generate chip-size blocks of products according to eqn. (3). Since each core uses multiple core-size blocks from rows of A and columns of B to generate its intermediate result, we need to repeatedly load the operand blocks from DRAM. However, further examining the problem, it can be seen that instead of repeatedly reading the same blocks from DRAM, we can cycle the operand data through the cores, such that the A blocks are cycled row-wise and the B blocks are cycled column-wise.

To demonstrate the above, we examine the case of a 16-cores Epiphany chip used to implement the 2nd level. In this case, we define the chip-level block as a 4×4 block matrix of the form:

$$(4) \quad A = \begin{pmatrix} \tilde{A}_{11} & \tilde{A}_{12} & \tilde{A}_{13} & \tilde{A}_{14} \\ \tilde{A}_{21} & \tilde{A}_{22} & \tilde{A}_{23} & \tilde{A}_{24} \\ \tilde{A}_{31} & \tilde{A}_{32} & \tilde{A}_{33} & \tilde{A}_{34} \\ \tilde{A}_{41} & \tilde{A}_{42} & \tilde{A}_{43} & \tilde{A}_{44} \end{pmatrix}, \text{ and } B \text{ and } C \text{ matrices would be similar.}$$

Here, \tilde{A}_{pq} is the core-size block of size $H \times H$. Thus, the chip-size block is of size $W \times W = 4H \times 4H$.

Solving the equation for elements along the first row of C , one from each block, gives:

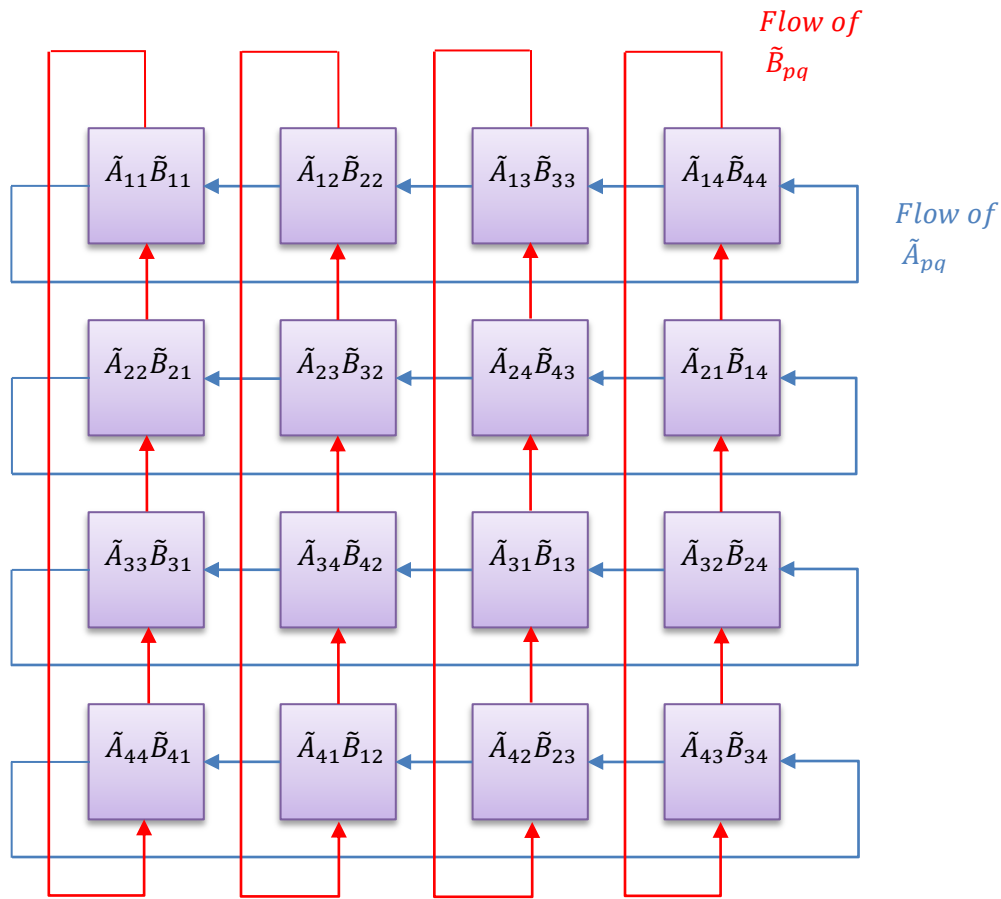
$$\begin{aligned}
 (5) \quad c_{11} &= a_{11} \cdot b_{11} + \dots + a_{1W} \cdot b_{W1} = \sum_{r=1}^W a_{1r} \cdot b_{r1} = \\
 &= \sum_{r=1}^H a_{1r} \cdot b_{r1} + \sum_{r=H+1}^{2H} a_{1r} \cdot b_{r1} + \sum_{r=2H+1}^{3H} a_{1r} \cdot b_{r1} + \sum_{r=3H+1}^W a_{1r} \cdot b_{r1}
 \end{aligned}$$

$$\begin{aligned}
 (6) \quad c_{1,H+1} &= a_{11} \cdot b_{1,H+1} + \dots + a_{1W} \cdot b_{W,H+1} = \sum_{r=1}^W a_{1r} \cdot b_{r,H+1} = \\
 &= \sum_{r=1}^H a_{1r} \cdot b_{r,H+1} + \sum_{r=H+1}^{2H} a_{1r} \cdot b_{r,H+1} + \sum_{r=2H+1}^{3H} a_{1r} \cdot b_{r,H+1} + \sum_{r=3H+1}^W a_{1r} \cdot b_{r,H+1}
 \end{aligned}$$

$$\begin{aligned}
 (7) \quad c_{1,2H+1} &= a_{11} \cdot b_{1,2H+1} + \dots + a_{1W} \cdot b_{W,2H+1} = \sum_{r=1}^W a_{1r} \cdot b_{r,2H+1} = \\
 &= \sum_{r=1}^H a_{1r} \cdot b_{r,2H+1} + \sum_{r=H+1}^{2H} a_{1r} \cdot b_{r,2H+1} + \sum_{r=2H+1}^{3H} a_{1r} \cdot b_{r,2H+1} + \sum_{r=3H+1}^W a_{1r} \cdot b_{r,2H+1}
 \end{aligned}$$

$$\begin{aligned}
 (8) \quad c_{1,3H+1} &= a_{11} \cdot b_{1,3H+1} + \dots + a_{1W} \cdot b_{W,3H+1} = \sum_{r=1}^W a_{1r} \cdot b_{r,3H+1} = \\
 &= \sum_{r=1}^H a_{1r} \cdot b_{r,3H+1} + \sum_{r=H+1}^{2H} a_{1r} \cdot b_{r,3H+1} + \sum_{r=2H+1}^{3H} a_{1r} \cdot b_{r,3H+1} + \sum_{r=3H+1}^W a_{1r} \cdot b_{r,3H+1}
 \end{aligned}$$

It can be seen how the calculation of each element can be divided among 4 cores in a row, all share the same a_{ij} data. Since addition is commutative, we can change the order in which we calculate the core-size block results. Thus, loading each element of A and B from DRAM is done just once, instead of 4 times. After calculating a core-size block, the data is cycled through the cores, such that the elements of A are copied to the neighboring core on the left side and the elements of B are copied to the neighboring core above. The copying is cyclic, which means that the A data from the top most row is copied to the bottom row and the B data from the leftmost column is copied to the rightmost column. The following figure shows the initial data assignment and the direction of the data flow:



Initial data assignment and data flow direction in a chip-size multiplication

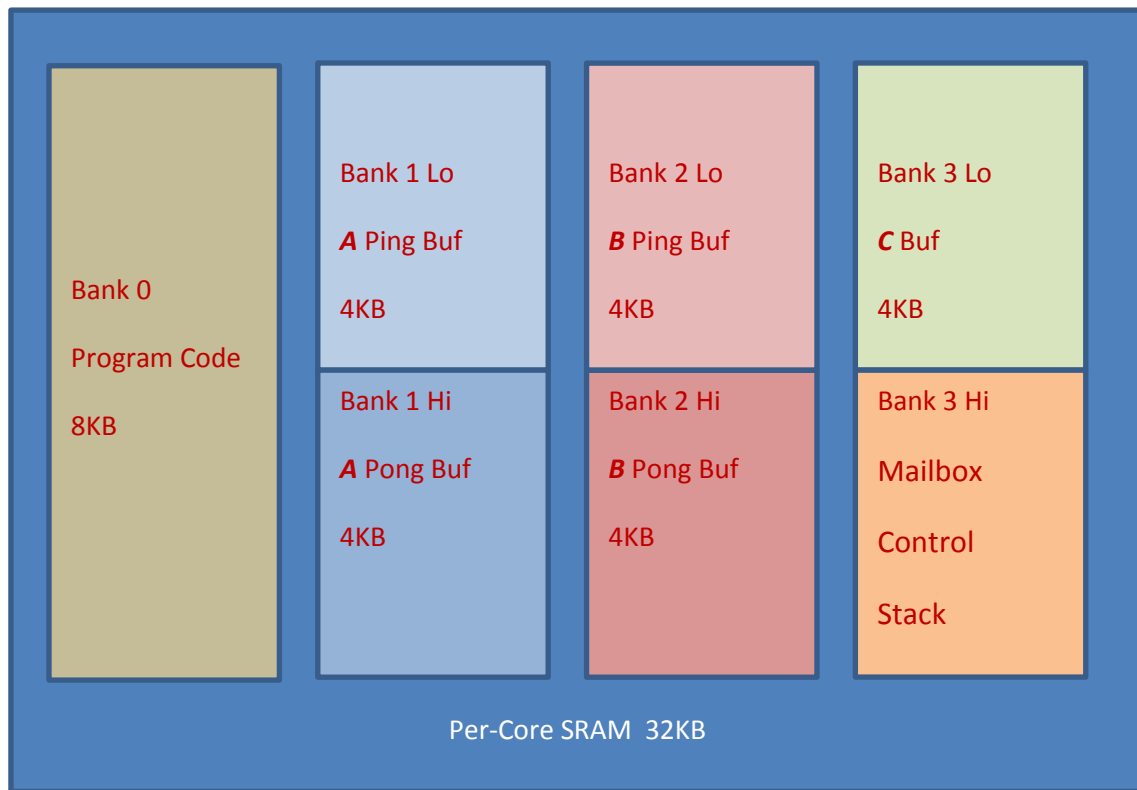
- 3) The 3rd level is the full matrix matmul. In this level, we use the chip-size blocks to generate the final full result. The program's outer loop loads chip-size amount of data at a time onto the 16 cores to calculate the intermediate block result. The number of loop iterations required to complete the calculation depends on the size of the matrix operands. Generally, it is limited only by the size of the available DRAM. At this stage, it is unavoidable to load the operands data multiple times. However, it is possible to avoid multiple write and read cycles of the intermediate block results. To do this, each block of C is calculated using all of its relevant data before writing it back to the result buffer.

Memory Arrangement

The memory allocation for the program is divided to two major parts. The 1st part is the on-chip, per-core memory. There, the innermost block matrix product is calculated. The 2nd part is the off-chip

DRAM. There, the *A*, *B* and *C* operand matrices are stored. Additionally, a system-level host-device Mailbox and control structure is allocated in the DRAM and is used for the host-device communications.

The E16G3 Epiphany chip in the EMEK3 system has 32KB of per-core SRAM, arranged in four 8KB banks. Concurrent access to more than one bank is possible without a performance penalty. We reserved the first bank for program code. The second bank contains the double-buffers of the *A* operand sub-blocks. The third bank contains the double-buffers of the *B* operand sub-blocks. The lower half of the fourth bank contains the buffer for the *C* result sub-block. Because there is no writing back of the result until the end of the calculation, there is no need for double buffering of *C* sub-blocks. The higher half of the fourth bank contains the per-core control structure and inter-core mailbox, as well as the program's stack.



Results and Analysis

The above algorithm was implemented and tested on the EMEK3 and EMEK4 platforms. The EMEK3 kit is based on the E16G3 Epiphany chip which has 16-cores. The chip clock frequency is set to 400MHz but may be increased to 1GHz. The EMEK4 kit utilizes the E64G4 Epiphany chip boasting 64 cores. The chip runs at 500MHz, which may be increased to 800MHz.

The DRAM is connected to the chip via an extension of the eMesh network that was implemented on the STARTIX FPGA. This eMesh extension is clocked at 50MHz, which is very slow compared to the chip

and link clock. As a consequence, data transfer to and from DRAM is slow and this should be considered when interpreting the benchmark results below. The Epiphany links are capable of writing and reading one byte every clock cycle. Thus, on a balanced system, for the E16G3 @ 400MHz, the theoretical write data rate from chip to DRAM is 400MB/s. On the EMEK3 however, the actual measured write data rate is 82MB/s!

We tested the program's performance by calculating the multiplication of two real 512x512 sized matrices. The measured timings were as follows:

EMEK3

The total time for the calculation was 369.2 msec = 147.6 Mcycles.

Out of this, about 12.5% of the time is calculation and the rest is read/write to DRAM.

This implementation included hand optimized version of the core matrix multiplication function. When using a naïve implementation (3 simple nested loops), the timing was 534.0 msec

EMEK4

The total time for the calculation was 179.0 msec = 89.5 Mcycles.

Out of this, about 5.2% of the time is calculation and the rest is read/write to DRAM.

When using the naïve implementation, the timing is 220.0 msec.

PC

In comparison to the above measured numbers, we tested the same multiplication, using the naïve implementation and the similar compiler options, on a dual-core AMD desktop PC. The measured timing for the calculation is 2,450 msec.

<i>Stage</i>	EMEK3 (E16G3)		EMEK4 (E64G4)	
	<i>Cycles [x10⁶]</i>	<i>%</i>	<i>Cycles [x10⁶]</i>	<i>%</i>
core matmul	17.5	12.4	4.4	4.9
Int. data swap	0.75	0.5	0.20	5.1
ext. DRAM access	132.5	89.9	85.4	95.5
Total^(*)	147.6	100	89.5	100

(*) note that the total count does not equal the simple sum of the components b/c some operations are performed in parallel across multiple cores.

Testing the program for different matrix sizes shows the expected theoretical performance scaling. For example, calculating 256x256 size product is about 8 times faster than 512x512. This is exactly as expected since the calculation complexity is $O(n^3)$.

Additionally, the calculation time (excluding the DRAM transfer times) is scaled according to the theoretical prediction, such that the E64G4 performs the calculation about 4 times faster than the E16G3.

Conclusions

In this paper we presented the implementation of a parallel big matrix multiplication program. We saw how with smart division of the workload among symmetric tasks, which are performed on a fast array of CPU cores, we can achieve almost theoretical scaling of performance, with a very little overhead.

The program was tested on two evaluation system, using 16-cores chip and 64-cores chip. These systems are inherently unbalanced in terms of DRAM throughput. Thus, the actual calculation takes a small portion of the total time. With a balanced system the throughput can be increased significantly and the benchmark performance will be improved dramatically. Furthermore, when the chips are clocked at their fastest clock speed, even more performance can be squeezed from the system.

Comparing the results to what was achieved on a standard desktop PC, we saw a huge advantage to the Epiphany based systems - up to 10 times faster on an unbalanced system and under-clocked chips.

References

- [1] Adapteva whitepaper "[Approaching Peak Theoretical Performance with Standard C](http://www.adapteva.com/white-papers/approaching-peak-theoretical-performance-with-standard-c)", <http://www.adapteva.com/white-papers/approaching-peak-theoretical-performance-with-standard-c>
- [2] The program's code, in the form of an Eclipse project "matmul-16" is available for download from the Adapteva website at <http://www.adapteva.com/support/examples/>