

Matrix multiplication on the multi-core Anemone processor

April 2011

Paralant Ltd. (www.paralant.com)

1 Abstract

This paper presents an xGEMM (matrix multiplication) multi-core implementation on BittWare's Anemone floating-point FPGA co-processor that has 16-core Epiphany cores on an eMesh. The Insight BLAS library contains single-core matrix multiplication SGEMM and CGEMM functions. These functions assume the matrix data is held in contiguous memory. For small dimensions, where the matrices data can be located in the DSP core local memory, the library routines provide exceptional performances. For large matrices that are located off-chip because they cannot be held fully in the local memory, even greater performances can be obtained through the use of the multiple DSP cores. This paper describes the single DSP core SGEMM routine and how to perform a multi-core SGEMM across the Anemone. The performance of the SGEMM and CGEMM routines are presented for both single core and multiple cores.

2 Introduction

xGEMM is a level-3 BLAS (basic linear algebra subprogram [1]) routine, where BLAS has become a de-facto standard library to perform basic linear algebra operations, such as vector and matrix multiplication. The BLAS API was first released in 1979 and is used throughout countless industries directly or indirectly within their applications. Large library packages, such as LAPACK (Linear Algebra PACKage [2]), use BLAS; and off-the-self programs such as Matlab and Mathematica rely on BLAS implementations to gain access to a processor's full compute potential. As such, chip manufactures like Intel, AMD, Nvidia, ARM, TI and ADI, all provide a BLAS library that give the best possible performance of their respective processor.

Level-1 BLAS contains vector linear algebra routines; level-2 BLAS contains vector-matrix routines and level-3 BLAS routines are matrix-matrix operations. This paper focuses on xGEMM, which is a matrix-matrix multiplication routine. Each of the xGEMM routines calculates a new matrix C based on the matrix-product of A and B , and the old value of matrix C :

$$C \leftarrow \alpha AB + \beta C$$

where α and β values are scalar coefficients. The "x" in xGEMM is replaced with S, D, C or Z depending on the precision and data type. The letters S, D, C and Z stand for single precision, double precision, single precision complex and double precision complex respectively. SGEMM is thereby a single precision real-type matrix-matrix multiplication routine.

Figure 1 shows two graphical ways of visualizing the xGEMM operation. The first is a two-dimensional image that aligns the three matrices; A , B and C along the edges where they share common dimensions. The three-dimensional "box" image is seen in many publications, which is a folded up version of the two-dimensional representation.

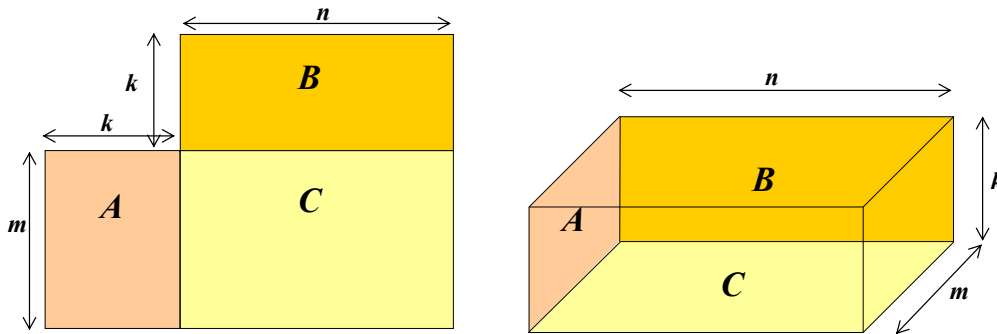


Figure 1 SGEMM matrices and dimensions

The box in Figure 1 has a volume of mnk and this volume is directly proportional to the total arithmetic compute of the SGEMM algorithm, which is $2mnk$ with each multiply-add operation performed as two floating-point operations. The amount of data is directly proportional to the area of each box side. Using this analogy, the SGEMM function can be seen to be an $O(n^2)$ data and $O(n^3)$ compute problem. [3] For problems with sufficiently large matrices dimensions, the compute will always take longer to perform than the time to transfer the data from memory. A limiting factor to the SGEMM performance is the latency between the start of transferring the data and the start of the compute. A perfect SGEMM implementation would overlap all the data transfers with the compute.

3 Single core implementation

The Insight BLAS library provides a SGEMM routine that executes on a single core. This is optimised to read/write the matrix data from local memory. The inner loop of the matrix-matrix multiplication performs dual issue instructions so that the load/store is overlapped with the compute. The Epiphany fused-multiplication-addition operator is used in the optimised inner loop to perform two floating-point operations on every clock cycle. For best performance, the routine should spend the majority of its time iterating within the innermost loop without having to perform the outer loop control code. In the current version of the SGEMM, the routine operates over the k dimension in the innermost loop. The implementation is therefore most optimal when $k > m, n$ [4]

The BLAS library routine will operate with matrix data in either local or external memory, but the optimised routine is designed to *hide* the IO latency of local core memory load/stores. Load stalls will be observed with external memory locations. When the matrix data is initially located outside of the local memory, a higher performance will be achieved if all of the data is pre-loaded (or cached) into local memory prior to the SGEMM routine call.

3.1 Epiphany core architecture

A single Anemone 104 processor contains multiple Epiphany DSP cores, where all the cores are identical and contain an integer ALU, a floating-point unit (FPU), 64 32-bit registers, and four 8k-byte memory banks (Figure 2).

The Epiphany instruction pipeline will issue dual instructions when possible. This requires the successive instructions to utilise the IALU and FPU, and for there to be no contentions in the register file. A data load from memory whilst performing a floating-point operation will be executed in the same cycle when the two instructions use completely separate registers. When the instruction pipeline detects register or resource contentions, it will issue the instructions in the given program order on consecutive cycles. [5]

The Epiphany C compiler contains optimisation engines that attempt to generate dual issue instructions. The SGEMM routine is hand optimised and makes full use of the dual issue functionality, along with other algorithm design concepts.

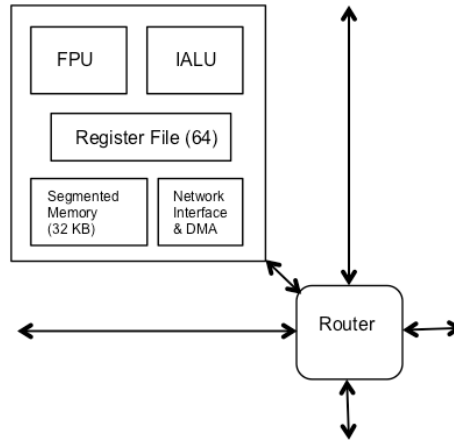


Figure 2 A single Epiphany core

3.2 Matrix-matrix multiplication

The values in matrix C are calculated as dot products between the corresponding row in A and column in B . Figure 3 depicts, using a white circle, the first value in C , along with two white rectangles that correspond to the row and column in A and B that are required to compute this value.

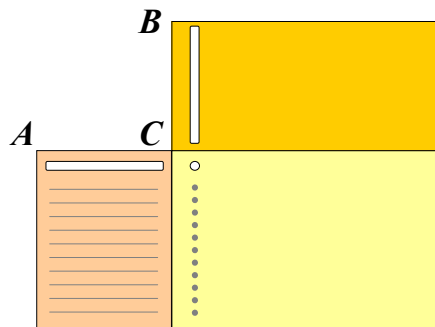


Figure 3 The row in A and column in B that is required to compute the first element of the result matrix C

The column values in matrix B are used to calculate the first value element in each matrix C columns. The same B column values are used to compute each of the other C values in the corresponding column. As such, if the values in B can be loaded once and reused, the number of load operations from B can be minimised.

In the same way that the values in B can be reused once loaded, the values in matrix A are also required in multiple computations. The values in a row of A are required to compute each value in the same row of matrix C . The ideal scenario would allow all the values in both matrices A and B to be loaded into registers once and used until they are no longer required. It is not possible to load all the data for large dimensions because of the finite number of registers. However, due to the Epiphany's large register file size, it is possible to load matrices with dimensions of $m=n=k=4$.

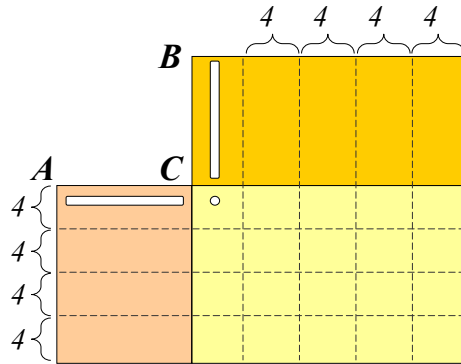


Figure 4 SGEMM computation split into smaller $m=n=k=4$ sub-matrices

The Epiphany SGEMM minimises the number of required memory loads by partitioning the problem into smaller dimensions of $m=n=k=4$; this is known as *blocking*. This maximises the ratio between compute operations and loads from memory before subsequent loads from the next sub-block are required. The algorithm is written in a manner that computes multiple matrix C values simultaneously, and the final value is computed fully before it is stored back to memory.

The outer SGEMM compute loops iterate over each of the 4-by-4 sub-blocks of matrix C . As a consequence of this implementation, each of the matrix dimensions has to be a multiple of four.

4 Multi-core implementation

4.1 Epiphany mesh architecture

The Epiphany architecture provides a mesh of DSP cores, as shown in Figure 5. These DSP cores are interconnected via an IO mesh, which allows each core to access every other core's local memory and any off-chip memories. Each DSP contains two DMA engines that are optimised to read the local memory and write the data to an off-core memory location via the mesh. The DMA engines can be used to overlap the core compute with the background loading and storing of data across the mesh. [5]

The mesh has an 8 GBytes per second full duplex bandwidth between each router; that is 8 GBytes per second in each direction. The data is routed around the mesh according to the destination coreIDx and coreIDy coordinate, where each router resolves the coreIDx coordinate first, then the coreIDy by passing the data out through the most relevant port. A round-robin scheduler is used in the router so that each input port will not be blocked for longer than 4 cycles.

The off-chip data transfers pass through one of the four external bridges; which of the four bridges (north, south, east or west) is determined by the destination coreIDx and coreIDy address coordinates. The external bridge has multiple cores accessing a one 1 GByte per second port. There is an understandable bottleneck in the off-chip bandwidth; therefore a tight control over the off-chip data transfers will provide better performance. Off-chip data is best read once, then shared between the multiple DSP cores as appropriate.

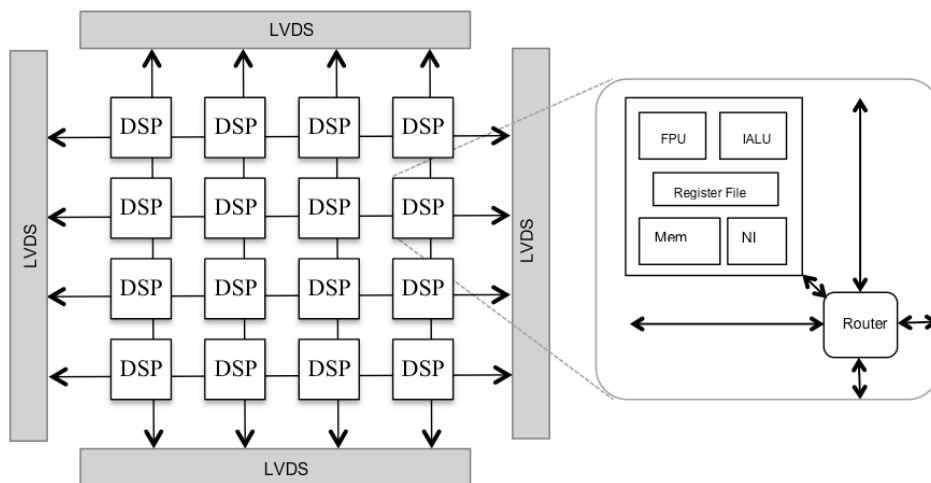


Figure 5 Epiphany processor mesh of cores

4.2 Partitioning the SGEMM for multiple cores

When designing a SGEMM for multiple cores, there is a well understood way of partitioning the problem; this is known as *blocking* of the matrices, which partitions the problem into smaller parts. Figure 6 depicts the partitioning of the SGEMM into four quarters, where each can be computed independently on separate cores. This type of partitioning can be extended to as many sub-SGEMM problems as desired, where it is reasonable to match the number of quadrants in matrix C with the number of available cores. The SGEMM described in Figure 6 uses only four quadrants, where this is used to keep the descriptions and diagrams more concise. All sub-matrix sizes are chosen to fit all of the data into the associated core local memory, where the data resides in off-chip memory prior to executing the multi-core SGEMM implementation.

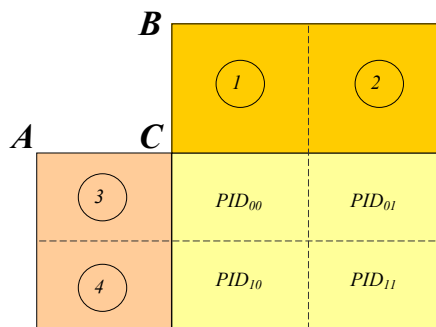


Figure 6 SGEMM computation split into sub-matrices

4.2.1 One quadrant per core

Referring to Figure 6, there are four quadrants in C , these are labelled: PID_{00} , PID_{01} , PID_{10} and PID_{11} , where PID stands for process ID. Let these quadrant processes be implemented on the cores with coreIDs: 00, 01, 10 and 11; i.e. the top-left 2-by-2 cores in the mesh of Figure 5. The DSP with coreID=00 requires the preload of sub-matrices 1 and 3 before calling the

SGEMM. Likewise coreID=01 requires the preload of sub-matrices 2 and 3; coreID=10 requires sub-matrices 1 and 4; and coreID=11 requires sub-matrices 2 and 4.

Each DSP core could read the required data from external memory independently, thereby requiring no synchronisation between any of the cores. However, as discussed in section 4.1, there is an off-chip bottleneck that will restrict performance especially when all the participating cores do not cooperate.

A resource friendly solution to reading the off-chip data is to bring the data on-chip only once, then to use the high bandwidth on-chip mesh to pass the data to all the required locations. In this example; if coreID=00 reads sub-matrix 1, coreID=11 reads sub-matrix 2, coreID=01 reads sub-matrix 3 and coreID=10 reads sub-matrix 4, then subsequently the data is passed around requiring fewer data transactions over the external bridge. Using this method of loading the off-chip data, the subsequent data movement between cores requires only one hop around the mesh; sub-matrix 1 is passed from coreID=00 to coreID=10, sub-matrix 2 is passed from coreID=11 to coreID=01, sub-matrix 3 is passed from coreID=01 to coreID=00, and sub-matrix 4 is passed from coreID=10 to coreID=11. There is no resource conflict in this data movement.

The above example is given for a four-core example that can be said to be relatively straight forward. Larger examples with more cores and larger dimensions will require more inter-core transfers and in some cases will require hops over multiple mesh routers. The design layout of the PID_{xy} to coreID must be considered in the design/layout process to either remove or reduce the resource contentions.

Once all the data resides in local core memory, the SGEMM routine can be called and the resulting C matrix can be written back to off-chip memory. Each core is writing back to different location in the off-chip matrix C , therefore no race condition will occur.

The procedure of one quadrant per core, as described here, requires all of the matrix data to be preloaded into the local core memory prior to the SGEMM compute. The result is only written back once the SGEMM is completed. This implementation does not overlap the external data transfers with the compute. Ideally, the SGEMM compute would completely overlap the data transfers.

4.2.2 Multiple quadrants per core

The ideal scenario of completely overlapping data transfers with compute is not feasible in a single SGEMM; however it is possible to overlap some of the data transfers with the SGEMM compute by partitioning a single SGEMM or by performing multiple SGEMMs in a sequence. Consider a single SGEMM problem as the four quadrants in Figure 6. The four quadrants can be performed on a single core as four SGEMMs. To start the first SGEMM requires only sub-matrix 1 and 3. Whilst computing PID_{00} the sub-matrix 2 can be loaded ready for the SGEMM in PID_{01} . Completing this sequence; sub-matrix 4 can be loaded whilst computing PID_{10} , leaving PID_{11} to be computed without loading any more values. The storing back of the resulting matrix C would overlap with the compute where possible.

To extend this overlapping of compute with data loading, the four quadrants in Figure 6 can be broken down further into eight, twelve, sixteen etc. quadrants, where each core computes multiple quadrants. Each quadrant corresponds to one SGEMM routine call.



Figure 7 When the data initially resides in the external memory, the data is first transferred into local memory before computation starts. The result data can be written back to external memory once completed.

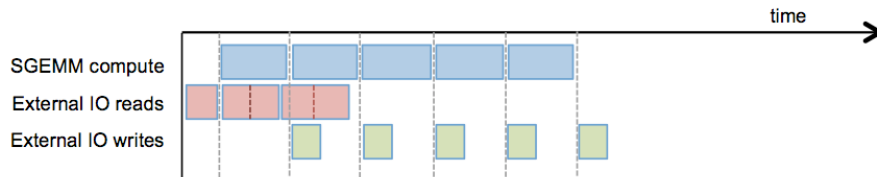


Figure 8 Each core can call the SGEMM function multiple times to give the opportunity to overlap the core function compute with external memory data transfers. A small data transfer is required before starting the first SGEMM. Subsequent external transfers can be started as a DMA and operate in the background. Results can be transferred after each SGEMM function call.

In Figure 7 the external data transfers are not overlapped with the core compute whereas in Figure 8 the compute is overlapped and the overall performance can be improved. Figure 8 shows a small initial IO read of data, sufficient to start computation that will overlap with the data IO read for the second function call compute. This figure shows the second IO read that is twice in size. Sufficient data is loaded for two subsequent function calls. Once each SGEMM function is complete, its results can be written back to external memory. This is overlapped with both the core compute and the IO reads. The Epiphany mesh is full duplex, and therefore reads and writes down the same mesh paths will not affect each other. Ideally, the input and output data will be in separate external memory banks to limit memory-paging issues.

4.2.3 Inter-core synchronisation

When data is shared between multiple cores, inter-core synchronisation is required. When correctly implemented, synchronisation prevents race conditions to ensure data is not over written by another core and to ensure the correct data is present before reading remote memory locations. In processors that execute multiple cores in lock-step; i.e. the same code running on all cores in each cycle, synchronisation is not always essential. However, in processors such as the Anemone 104 that have multiple Epiphany cores executing separate programs that can be independently interrupted, stopped, restarted etc. synchronisation is essential.

There are two places in the multi-core SGEMM implementation where synchronisation is either essential and/or helpful. The first is when preloading the matrix data from off-chip memory. A core can read its required off-chip data when it is required within the program control flow. When data needs to be shared with another core, as described in section 4.2.1, the program needs to ensure the other core is ready to receive it.

Secondly, synchronisation can be used to help share resources in a friendly manner to improve performance. As an example, the external memory interface may be better utilised exclusively by one core at a time. When multiple cores are reading and writing to the external memory, page swapping can slow the transfer considerably. Exclusive access by one core at a time can give an overall performance benefit.

5 xGEMM performance

The xGEMM computation is timed using the Epiphany core CTIMER functionality [5]. A range of matrix dimension sizes is covered to demonstrate the effect of the ratio between the inner loop (k-dimension) and the outer loops (m and n-dimension) on the performance.

The BLAS performance is commonly measured in FLOPS (floating point operations per second), where the SGEMM performance is calculated as:

$$FLOPS = \frac{2mnk}{time}$$

and the CGEMM performance is calculated as:

$$FLOPS = \frac{8mnk}{time}$$

Each core has a theoretical peak of 2×10^9 FLOPS, or 2 GFLOPS, with the core clock frequency of 1GHz.

5.1 Single-core SGEMM performance

The single core performance of the Epiphany SGEMM implementation is given in Figure 9 through Figure 12 for the four transpose-types. Each of the SGEMM transpose types is optimised as a separate Insight library routine that may be called directly or via the standard API.

The SGEMM performance attenuates to 1.8 GFLOPs and this is 90% of the theoretical peak of a single core. Each of the transpose option types has very similar performance characteristics. When the dimension sizes are increased the routine has more to compute within the inner loops without the control code overhead. This is reflected in the performance. The figures provide a performance point for each of the possible sizes where $N=M$.

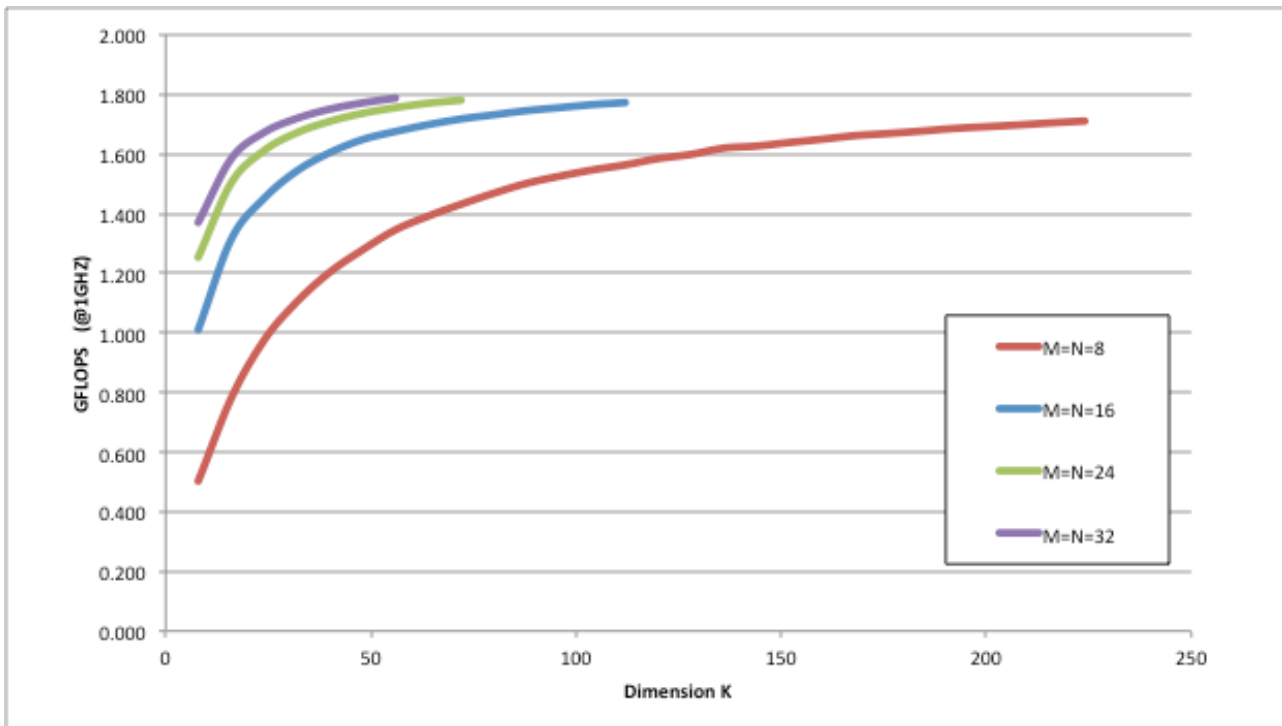


Figure 9 Libblas: SGEMM non-transposed/non-transposed [NN] performance on a single core

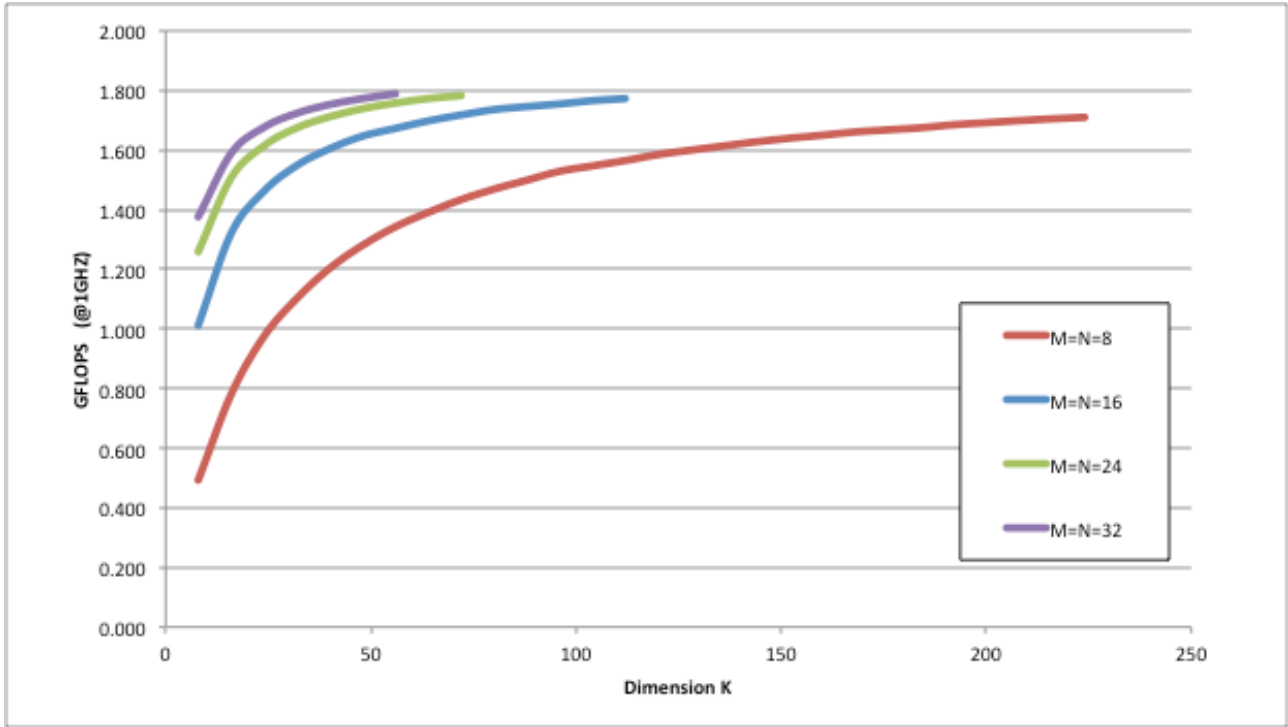


Figure 10 Libblas: SGEMM non-transposed/transposed [NT] performance on a single core

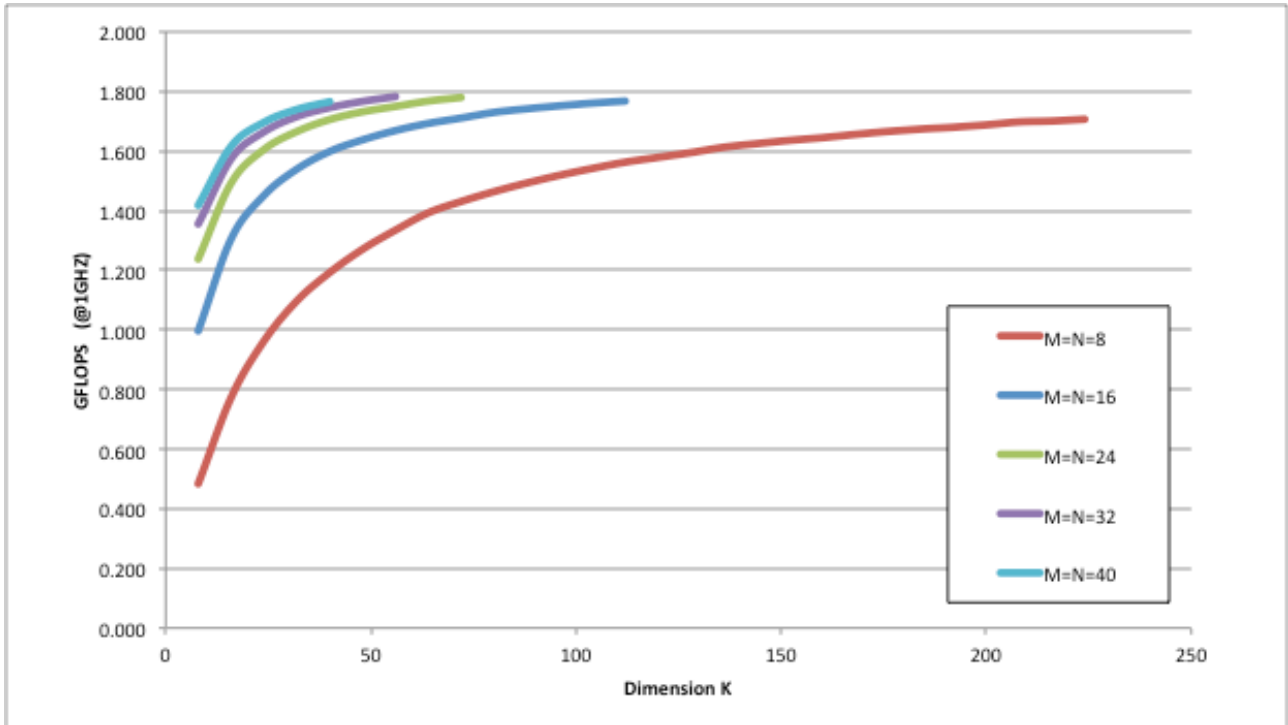


Figure 11 Libblas: SGEMM transposed/non-transposed [TN] performance on a single core

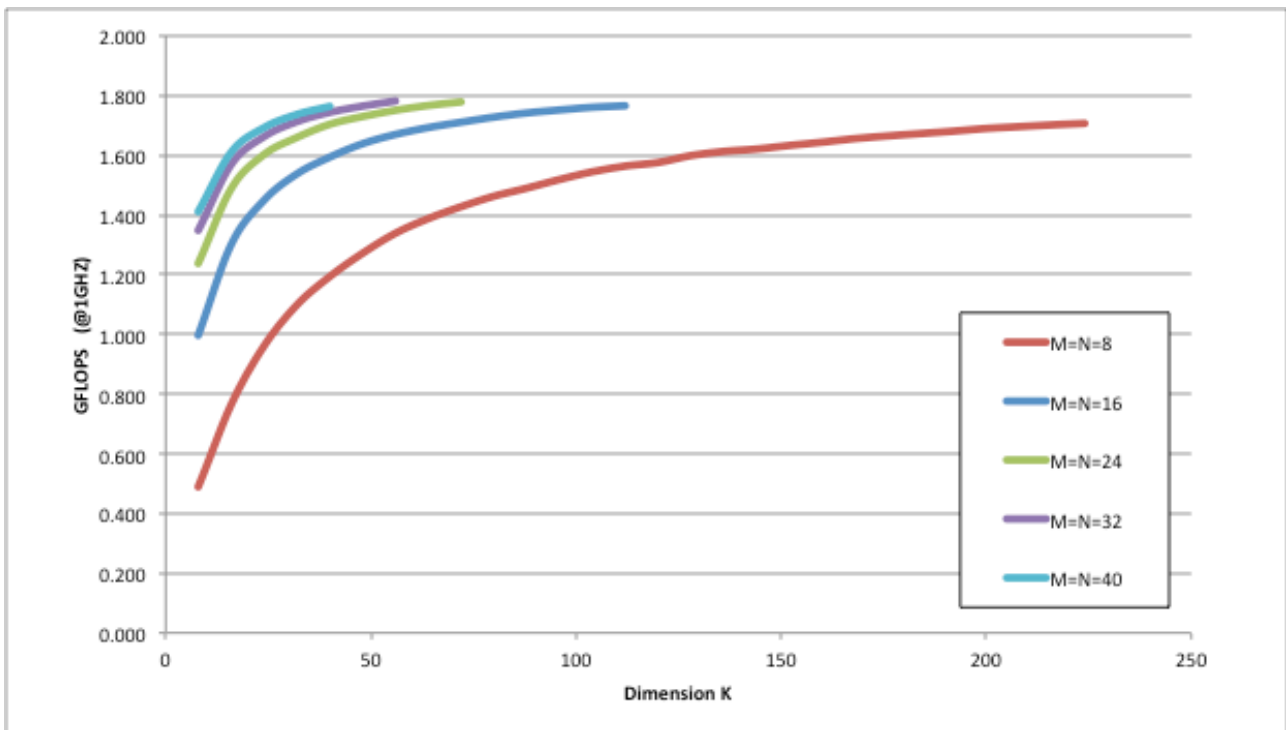


Figure 12 Libblas: SGEMM transposed/transposed [TT] performance on a single core

5.2 Single-core CGEMM performance

The single core performance of the Epiphany CGEMM implementation is given in Figure 13 through Figure 15 for three of the nine transpose-types. Each of the CGEMM transpose types is optimised as a separate Insight library routine that may be called directly or via the standard API.

The CGEMM performance attenuates to 1.75 GFLOPs and this is 88% of the theoretical peak of a single core. Like the SGEMM, each of the transpose option types has very similar performance characteristics. The figures provide a performance point for each of the possible sizes where $N=M$.

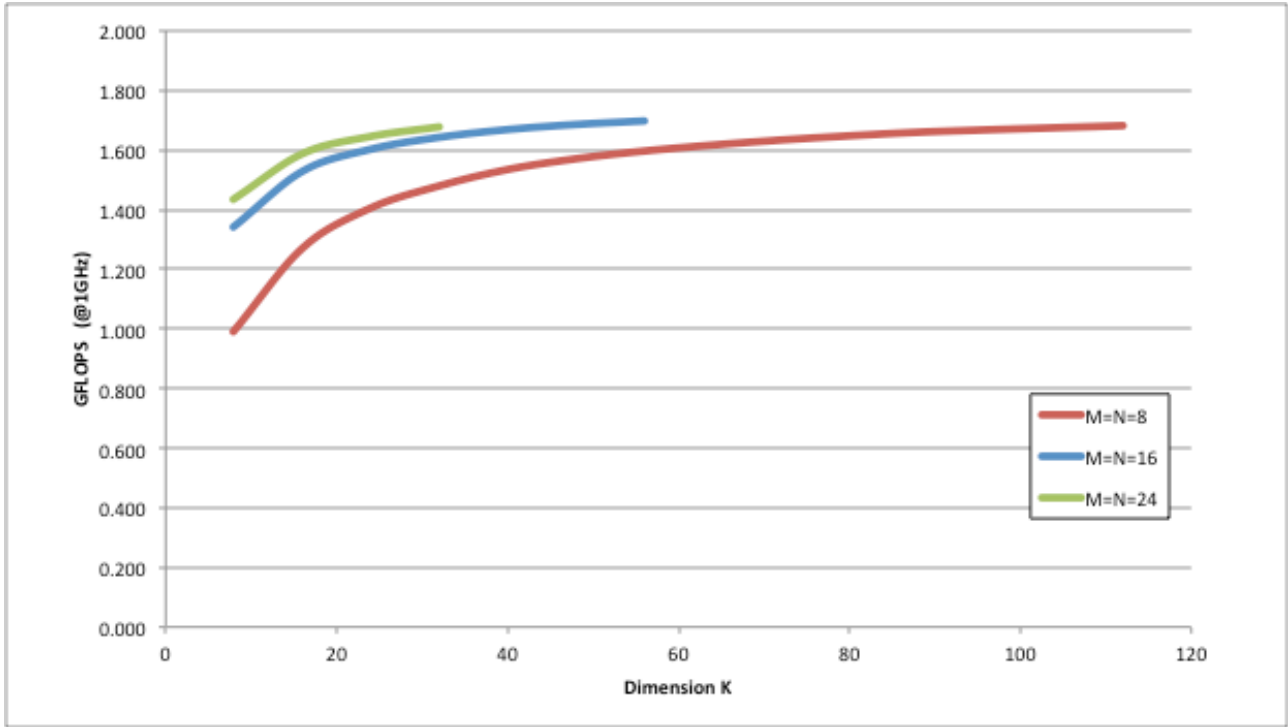


Figure 13 Libblas: CGEMM non-transposed/non-transposed [NN] performance on a single core

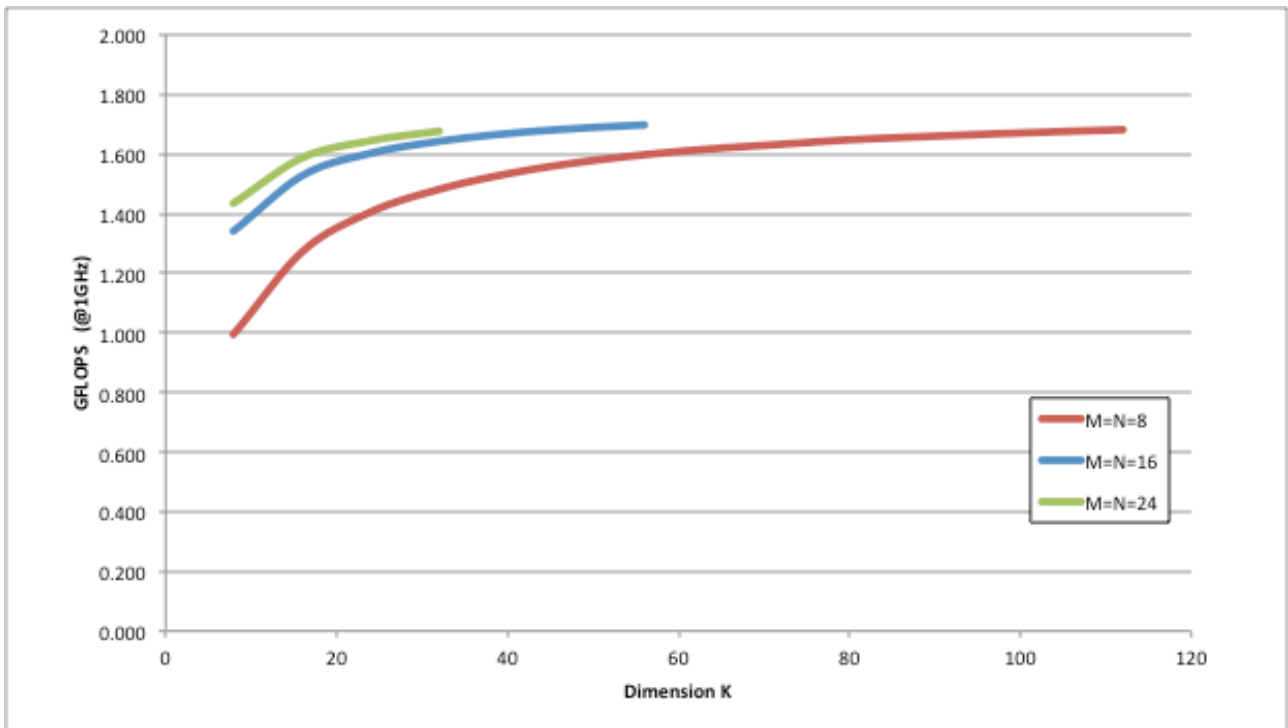


Figure 14 Libblas: CGEMM transposed/transposed [TT] performance on a single core

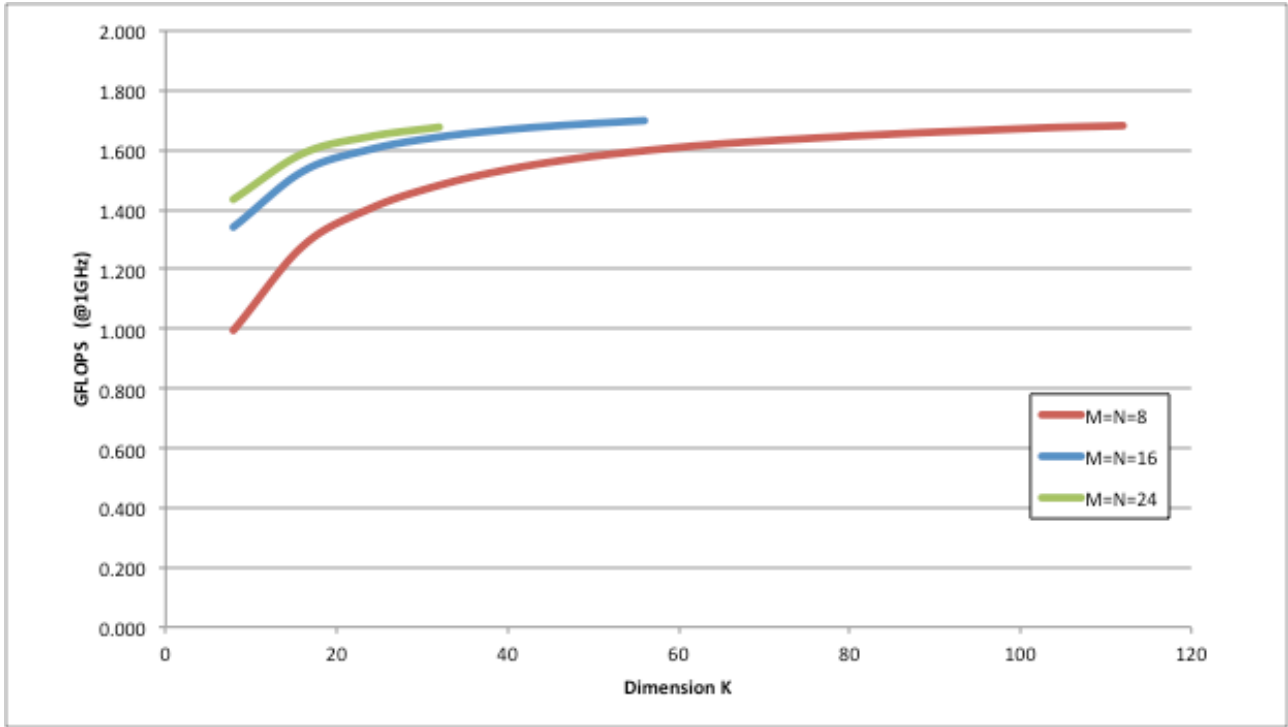


Figure 15 Libblas: CGEMM conjugate/conjugate [CC] performance on a single core

5.3 Four-core SGEMM performance

The four-core performance of the Insight SGEMM implementation is given in Table 1 through Table 3. The full matrix multiplication compute is distributed over the four cores. Each core computes one quarter of the C matrix. All of the data initially resides in the external memory with the cores transferring the data into the local core memory. The timer starts with the data in the respective cores local memory and ends when the all of the SGEMM functions complete. The resulting C matrix resides in the cores local memory when the SGEMMs complete and can be transferred back to external memory if not required for further compute.

The figures show the additional performance that can be achieved when the algorithm compute is distributed over the multiple cores. The $m=n=k=64$ four-core matrix multiplication implementation gives 88% of the theoretical peak performance of the four cores running at 2 floating-point operations every cycle.

The execution of the matrix multiplication is very scalable over the multiple Epiphany core, as shown here. Both the single-core and four-core implementations achieve approximately the same percentage of the peak performance (~90%). Once the data resides in local memory the algorithm does not require any further external IO and therefore is not influenced by the other cores. A good system design will therefore load the data into the local memories and keep it there for as long as possible.

Table 1 Four Epiphany core SGEMM performance with $m=n=16$

| | | | | |
|--------|-------|-------|-------|--------|
| k | 16 | 32 | 64 | 128 |
| Cycles | 4,419 | 5,553 | 7,685 | 12,045 |
| GFLOPS | 1.9 | 3.0 | 4.3 | 5.4 |

Table 2 Four Epiphany core SGEMM performance with $m=n=32$

| | | | | |
|--------|-------|--------|--------|--------|
| k | 16 | 32 | 64 | 128 |
| Cycles | 8,051 | 12,379 | 21,521 | 38,509 |
| GFLOPS | 4.1 | 5.3 | 6.1 | 6.8 |

Table 3 Four Epiphany core SGEMM performance with $m=n=64$

| | | | | |
|--------|--------|--------|--------|--|
| k | 16 | 32 | 64 | |
| Cycles | 22,497 | 39,869 | 74,741 | |
| GFLOPS | 5.8 | 6.6 | 7.0 | |

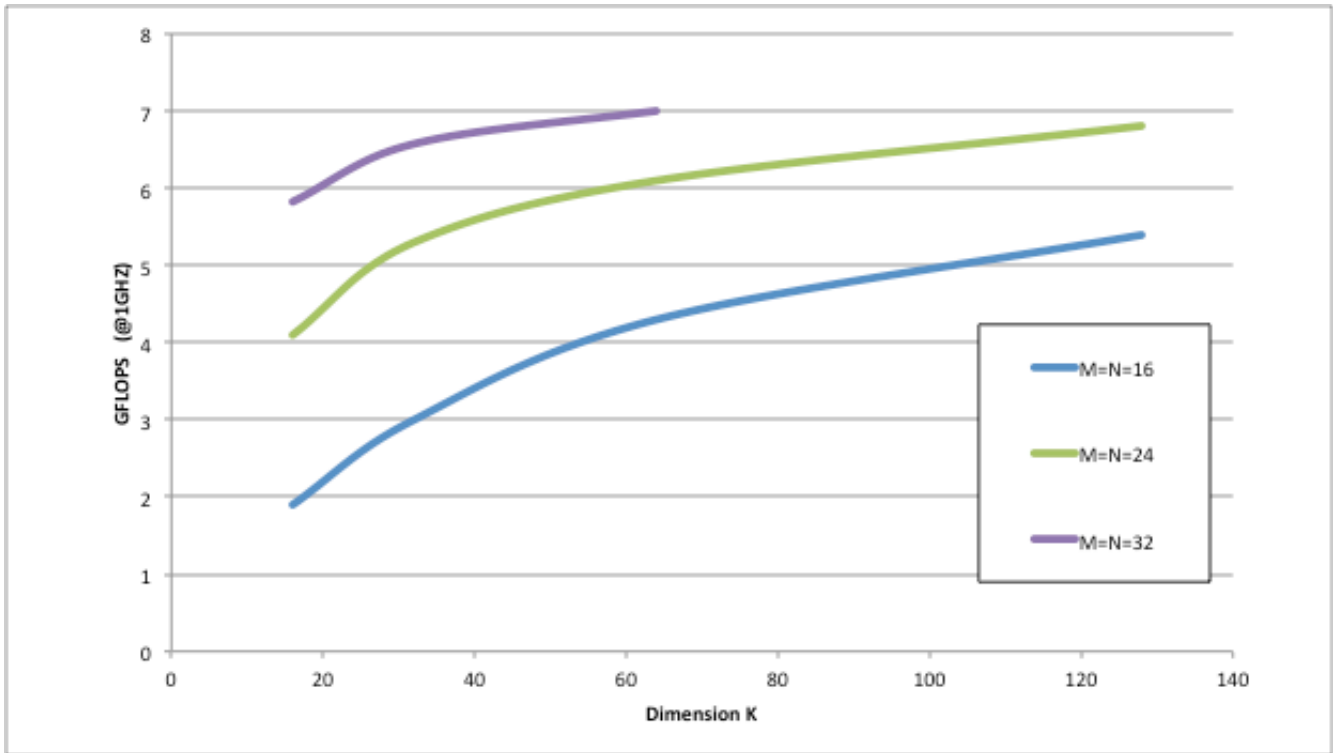


Figure 16 Libblas: SGEMM non-transposed/non-transposed [NN] performance on four-cores

6 Summary

Applications are very often difficult to partition across multiple cores. A common approach is to break a sequential application into component parts, including component parts that can partition over many cores. LAPACK is cited as an example of routines that are sequential algorithms, which are designed to call the parallelisable BLAS functions. The xGEMM is an example of an algorithm that partitions well over multiple cores and is commonly used in applications to help gain performances that only multi-core systems can provide.

This paper describes a SGEMM implementation for the multi-core Anemone processor. The algorithm is described in detail, showing how the compute can be distributed over independent processor cores. The matrices are partitioned into sub-matrices that can be computed independently. There is some duplication of the input data in the separate cores, but even this can be used as a performance advantage where data can be read once from external memory and copied between the cores via the high speed Epiphany mesh.

Matrix multiplication has an $O(n^3)$ compute and $O(n^2)$ bandwidth overhead. This means the algorithm is compute intensive and is ideal for demonstrating the compute performance of a processor. The results in this paper show that 90% of the theoretical peak performance within a core. The multi-core results show how the individual cores combine together to provide the improved overall performance that one expects to see in a multi-core processor.

The Epiphany library release provides example projects for the single core and multi-core SGEMM implementations. These act as templates from which specific application algorithms can be developed.

7 References

1. <http://www.netlib.org/blas/>
2. <http://www.netlib.org/lapack/>
3. Parallel Matrix-Matrix Multiplication, http://www.cs.indiana.edu/classes/b673/notes/matrix_mult.html
4. “Analysis of a Class of Parallel Matrix Multiplication Algorithms”, John Gunnels, Calvin Lin, Greg Morrow, Robert van de Geijn; The University of Texas, <http://www.cs.utexas.edu/users/plapack/papers/ipps98/ipps98.html>
5. Epiphany hardware manual

Paralant Ltd.

Copyright 2011 Paralant Ltd. The information contained herein is subject to change without notice.
Paralant shall not be liable for technical or editorial errors or omissions contained herein.
All marks are the property of their respective owners.