SummaGen: Parallel Matrix-Matrix Multiplication Based on Non-rectangular Partitions for Heterogeneous HPC Platforms

Stephen Patton*, Hamidreza Khaleghzadeh*, Ravi Reddy Manumachu†, and Alexey Lastovetsky†
School of Computer Science
University College Dublin
Dublin, Ireland
Email: *{stephen.patton,hamidreza.khaleghzadeh}@ucdconnect.ie, †{ravi.manumachu,alexey.lastovetsky}@ucd.ie

***Abstract*—Parallel matrix-matrix multiplication (PMM) of dense matrices is a foundational kernel of parallel linear algebra libraries in high performance computing (HPC) domain. The problem of finding the optimal shape of matrices for efficient execution of PMM on heterogeneous platforms has an engrossing history comprising of two distinct threads. The first thread focused purely on rectangular partitions whereas the second thread relaxed the rectangular partition constraint to allow non-rectangular partitions. The research works in the second thread, however, are entirely theoretical. There is no software implementation that would facilitate experimental studies of the practical performance and optimality of the proposed partition shapes. We address this gap in this work.**

We propose an implementation of PMM based on non-rectangular partitions called *SummaGen*. To study its efficacy, we compare the performances of PMM for four partition shapes proven optimal for three processor case where speeds of the processors are represented by positive real numbers. We conduct the experiments on a hybrid heterogeneous multi-accelerator NUMA node comprising of three heterogeneous devices, a dual-socket Intel Haswell multicore CPU, an Nvidia K40 GPU, and an Intel Xeon Phi 3120P. We show that the four shapes exhibit equal performances (with an average percentage difference of 8%) for a range of problem sizes where the speeds are constant confirming the optimality of these shapes in practice. We demonstrate further that the four shapes exhibit equal dynamic energy consumptions for this case.

We also present a study of performances of PMM for the same partition shapes for a matrix decomposition using load imbalancing data partitioning algorithm employing functional performance models (FPMs). The peak and average performances of the implementation are 80% and 70% of the theoretical peak floating-point performance of the machine.

***Index Terms*—Parallel Matrix-Matrix Multiplication; SUMMA; heterogeneous platforms; GPU; Intel Xeon PHI; multicore CPU**

I. INTRODUCTION

Parallel matrix-matrix multiplication (PMM) of dense matrices is a foundational kernel of parallel linear algebra libraries in high performance computing (HPC) domain. The problem of finding the optimal shape of matrices or optimal partitioning of matrices for efficient execution of PMM for heterogeneous platforms has an engrossing history comprising of two distinct threads or developments. We will present them first as the motivation for our work.

The first thread purely focused on finding the optimal shape of matrices based on rectangular partitions on heterogeneous platforms. Kalinov et al. [1] proposed a column-based rectangular partitioning algorithm based on speeds of processors that are positive real numbers. Beaumont et al. [2] proved that the partitioning problem is NP-complete (when speeds are positive rational numbers) and proposed a column-based rectangular partitioning approximation algorithm with an approximation ratio of 1.75. Nagamochi et al. [3] presented an approximation algorithm which improved the ratio to 1.25. Then, the focus shifted to the study of the problem where advanced performance models of computation are employed. Lastovetsky et al. [4] presented a column-based partitioning algorithm that takes as input 2D functional performance models (FPMs). Clarke et al. [5] proposed a variant of the column-based approach of [2] that used 1D FPMs. Fügenschuh et al. [6] subsequently reduced the approximately ratio to 1.15 satisfying some assumptions.

Round about 2006, second thread began when Becker at al. [7], [8] relaxed the constraint of rectangular partitions. The authors proposed optimal shape, called the *square corner*, for two heterogeneous processors where one partition is non-rectangular. DeFlumere et al. [9], [10] extended the findings to three heterogeneous processors and identified six potentially optimal shapes, some of which contain non-rectangular partitions. This inspired the work of [11] who combined the recursive technique of [3] and work on non-rectangular partitions to further reduce the approximation ratio to $\frac{2}{\sqrt{3}}$ with no assumptions and for an arbitrary number of processors. Finally, in [12], the optimality of four partition shapes has been mathematically proven and the accuracy of the best approximate solutions is analyzed against the optimal solutions for the case of three partitions where they can be found using the exact algorithm.

While efficient implementations of PMM based on rectangular partitions exist [13], [14]), the research works in the second thread on non-rectangular partitions are entirely theoretical. There is no software implementation for PMM that would facilitate experimental studies of the practical performance and optimality of the proposed partition shapes. We address this

978-1-5386-5555-9/19/$31.00 ©2019 IEEE
DOI 10.1109/IPDPSW.2019.00017

IEEE computer society

gap in this work.

We propose an implementation of PMM based on non-rectangular partitions called *SummaGen*. To study its efficacy, we compare the performances of PMM for four partition shapes proven to be optimal for three processor case where the speed of a processor is represented by a constant function of problem size [9], [10]. The partition shapes are a). Square corner, b). Square rectangular, c). Block 2D rectangular and d). Traditional 1D rectangular.

We perform our experiments on a hybrid heterogeneous multi-accelerator NUMA node comprising of three computing devices, an Intel Haswell multicore CPU consisting of two sockets of 12 physical cores each, an Nvidia k40 GPU, and an Intel Xeon Phi 3120P. Each accelerator is connected to a dedicated host core via a separate PCI-E link. A data-parallel application executing on this heterogeneous hybrid platform, consists of a number of kernels (generally speaking, multi-threaded), running in parallel on different computing devices of the platform. Due to tight integration and severe resource contention in such a heterogeneous hybrid platform, the load of one computational kernel in a given hybrid application may have a noticeable impact on the performance of others to the extent of preventing the ability to model the performance of each kernel in the hybrid application individually [15]. To address this issue, we restrict our study in this work to such configurations of the hybrid application, where individual kernels are coupled loosely enough to allow us to build their individual performance profiles with sufficient accuracy. To achieve this, we only consider configurations where no more than one CPU kernel or accelerator kernel is running on the corresponding device. Each group of cores executing an individual kernel of the application is modelled as an abstract processor [15] so that the executing platform is represented as a set of heterogeneous abstract processors. We make sure that the sharing of system resources is maximized within groups of computational cores representing the abstract processors and minimized between the groups. This way, the contention and mutual dependence between abstract processors are minimized.

The PMM applications are executed using three abstract processors. The first abstract processor contains 22 CPU cores executing the multi-threaded CPU kernel. The second abstract processor comprises the Nvidia K40c GPU along with its dedicated host CPU core executing the GPU kernel. And finally, the third abstract processor consists of Intel Xeon Phi 3120P co-processor along with its dedicated host CPU core executing the Xeon Phi kernel. The dedicated host CPU core is responsible for sending data from host to accelerator, kernel invocations on the accelerator and then copying results back from the accelerator to host. Therefore, the pair consisting of an accelerator and its dedicated host core executing one accelerator kernel is modelled by an abstract processor. The kernel executing on the accelerator uses all the cores of the accelerator. The execution time of a kernel in the GPU and Xeon Phi abstract processors includes the times of data transfer between the accelerators and their host cores. Because the abstract processors contain CPU cores that share some resources such as main memory and QPI, they cannot be considered independent. Therefore, the performance of these loosely-coupled abstract processors must be measured simultaneously, thereby taking into account the influence of resource contention [15].

We show that the four shapes exhibit the same performances for a range of problem sizes where the performance models are constant functions of problem size. *SummaGen* therefore confirms the optimality of these shapes in practice. We also demonstrate that the four shapes exhibit equal dynamic energy consumptions. The optimality of these shapes for dynamic energy is a subject for our current research.

The problem of finding the optimal shape of matrices for efficient execution of PMM when the speed of a processor is represented by a non-constant function of problem size is an open research problem. We study experimentally the performances of PMM for the four partition shapes where the matrix decomposition is determined using a data partitioning algorithm employing non-smooth functional performance models and uses load imbalancing technique [16], [17]. Optimal solutions found by this algorithm may not load balance the application in terms of execution time. We find that the partition shapes *square rectangle* and *block rectangle* perform better than the other two shapes.

The peak and average performances of the implementation are 80% and 70% of the theoretical peak floating-point performance of the machine, which is 2.5 TFLOPs calculated using the summation of the theoretical peaks for the three heterogeneous devices.

Our contributions can be summarized below:

- A software implementation called *SummaGen* of parallel matrix-matrix multiplication based on non-rectangular partitions for heterogeneous platforms.
- An experimental study employing *SummaGen* to determine the practical performance of four partition shapes on a hybrid heterogeneous multi-accelerator NUMA node comprising of three heterogeneous devices, an Intel Haswell multicore CPU and two accelerators, an Nvidia k40 GPU and an Intel Xeon Phi 3120P. We show that the four shapes exhibit the same dynamic energies and performances for a range of problem sizes where the speeds are constant functions of problem size. While this confirms their optimality in practice for performance, optimality for dynamic energy is an open research topic.
- An experimental study to determine the practical performance of four partition shapes when the speed of a processor is represented by a non-constant function of problem size.

The paper is organized as follows. Section II contains the formulations for PMM on heterogeneous platforms based on rectangular and non-rectangular partitions. Section III presents related work. Section IV contains the description of the implementation of SummaGen. Section VI contains the experimental results. Section VII concludes the paper.

II. Parallel Matrix-Matrix Multiplication Optimization Problem: Problem Formulation

The problem of optimal rectangular partitioning of square matrices for Parallel Matrix-Matrix Multiplication (PMM) on heterogeneous platforms can be formulated as follows (PMMR-OPT): Consider a matrix product $C = A \times B$, where A, B, and C are square matrices of $n \times n$ blocks of size r. Assume that we have p heterogeneous processors $P_i, 1 \leq i \leq p$, whose speed functions of problem size vector $\mathbf{x}$ are represented by $\mathcal{S} = \{s_i(\mathbf{x})\}_{i=1}^{p}, s(\mathbf{x}) : \mathbb{Z}^2_{\geq 0} \to \mathbb{R}_{\geq 0}$. The objective is to partition the matrices into p rectangles R_i of size $h_i \times v_i, \sum_{i=1}^{p} h_i \times v_i = n^2$, so as to:

- Minimize computation time:

$$\min T_{comp} = \min \max_{i=1}^{p} \frac{h_i \times v_i}{s_i((h_i, v_i))} \tag{1}$$

- Minimize total communication volume:

$$\min T_{comm} = \min \sum_{i=1}^{p} (h_i + v_i) \tag{2}$$

The aim of PMMR-OPT is to minimize the sum of computation (T_{comp}) and communication times (T_{comm}).

The total volume of communications during the execution of PMM is proportional to the sum of the half-perimeters of the p rectangles R_i given by the formula 2.

While achieving the objective of minimization of time of computations during the execution of PMM decides the sizes (h_i, v_i) of the rectangles R_i, achieving the objective of minimization of total communication volume shapes the layout of the rectangles in the square matrix.

PMMR-OPT is an open research problem. For the case where the speeds of the processors are represented by constant functions of problem size or scalars (positive rational numbers), the problem is proved to be NP-complete [2].

The problem of optimal non-rectangular partitioning of matrices is, however, comparably difficult to formulate. We call this problem PMMNR-OPT where the goal is to partition the square matrices into p non-rectangular shapes. A continuous version of this problem formulation is presented in [11].

Let Z denote a non-rectangular shape whose area is given by $\mathcal{A}(Z)$ and its covering rectangle by $R(Z)$, *i.e.* the Cartesian product of the projections of Z along both dimensions. If $R(Z) = [x_1, x_2] \times [y_1, y_2]$, then the height of Z is defined by $h(Z) = x_2 - x_1$ and its width, $w(Z) = y_2 - y_1$. The half-perimeter of Z defined as $\mathrm{c}(Z) = h(Z) + w(Z)$ represents the volume of communications.

PMMNR-OPT can be stated as follows: Given p heterogeneous processors $P_i, 1 \leq i \leq p$, whose speed functions are represented by $\mathcal{S} = \{s_i(a)\}_{i=1}^{p}, s(a) : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$, partition the matrices into p non-rectangular shapes Z_i of area $\mathcal{A}(Z_i)$, $\sum_{i=1}^{p} \mathcal{A}(Z_i) = n^2$, so as to:

- Minimize computation time:

$$\min T_{comp} = \min \max_{i=1}^{p} \frac{\mathcal{A}(Z_i)}{s_i(\mathcal{A}(Z_i))} \tag{3}$$

- Minimize total communication volume:

$$\min T_{comm} = \min \sum_{i=1}^{p} \mathrm{c}(Z_i) \tag{4}$$

The aim of PMMNR-OPT is to minimize the sum of computation (T_{comp}) and communication times (T_{comm}).

Here, for the sake of simplicity, we assume that there exist speed functions of processors of areas of the zones so that given an area of a zone, $\mathcal{A}(Z)$, one can determine the time of computations using the formula $\frac{\mathcal{A}(Z)}{s(\mathcal{A}(Z))}$.

III. Related Work

We divide our survey into five categories. First category reviews computation and communication performance models used in the theoretical study of PMM. Second category reviews research works that study the problem of optimal rectangular partitioning of matrices in Parallel Matrix-Matrix Multiplication (PMMR-OPT). Third category surveys works that have proposed both rectangular and non-rectangular shapes for the partitions. The fourth category presents an overview of communication-optimal and communication-avoidance algorithms. Final category looks at software libraries for PMM.

A. Computation and Communication Models

We review in this section performance models of computations and communications that are commonly employed in the analysis of PMM for heterogeneous platforms.

The most simple model is a constant performance model (CPM) where different notions such as normalized cycle time, normalized processor speed, average execution time, task computation time, etc. characterize the speed of an application [1], [2]. In CPMs, no dependence is assumed between the performance of a processor and the workload size.

The most advanced load balancing algorithms employ functional performance models (FPMs) that are application-specific and that represent the speed of a processor by a continuous function of problem size [18]. The FPMs capture realistically and accurately the real-life behaviour of applications executing on nodes consisting of uniprocessors (single-core CPUs).

The complex nodal architecture of modern HPC systems comprising of tightly integrated processors with inherent severe resource contention and NUMA result in significant variations (drops) in the performance profiles of parallel applications executing on these platforms thereby violating the assumptions on the shapes of the performance profiles considered by the FPM-based load balancing algorithms. In [16], [17], novel model-based data partitioning algorithms are proposed that employ load imbalancing parallel computing method employing non-smooth FPMs.

For the cost of communications, Hockney model is most commonly used where the cost of data transfer (of m bytes) between a pair of processors is represented by $\alpha + \beta \times m$ where α is the latency and β is the reciprocal of bandwidth of the communication link connecting the processors.

B. Rectangular Matrix Partitioning

Kalinov et al. [1] propose a column-based partitioning heuristic (KL) to solve PMMR-OPT. It is based on speeds of processors that are positive real numbers and does not take into account the cost of communications.

Beaumont et al. [2] prove that PMMR-OPT on heterogeneous platforms where the load is balanced between the processors whose speed functions are represented by positive rational numbers and the communication volume is minimized, is NP-complete. They propose a column-based partitioning heuristic (BR) and prove its optimality among all column-based approaches.

Lastovetsky et al. [4] present a column-based partitioning heuristic (FPM-KL) solving PMMR-OPT that takes as input 3D functional performance models (FPMs). FPM-KL takes an input a fixed 2D grid of processors and does not take into account cost of communications. Clarke et al. [5] propose a variant of BR that is based on 2D FPMs. It outperforms FPM-KL.

Nagamochi et al. [3] present an approximation algorithm solving PMMR-OPT having the approximation ratio 1.25. Fügenschuh et al. [6] improved the ratio to 1.15.

Li et al. [19] propose matrix-matrix multiplication on a heterogeneous platform composed of a CPU and an ATI GPU. They optimize PMM using a software pipeline design.

C. Non-Rectangular Matrix Partitioning

Brett et al. [7] study PMM for two interconnected heterogeneous processors and propose a non-rectangular matrix partitioning algorithm called *square corner*. While one partition is square, the other is non-rectangular. They demonstrate that *square corner* is superior to rectangular partitioning for ratios of speeds greater than three to one between the processors. Brett et al. [8] extend the *square corner* algorithm [7] for three interconnected heterogeneous processors. Here, two partitions in the corners are squares; the remaining partition is therefore non-rectangular. They show that the *square corner* partitioning is better than rectangular partitioning for highly heterogeneous platforms and non-fully connected network topologies. Both the works show that non-rectangular partitioning can be optimal but do not prove the optimality of the shapes.

DeFlumere et al. [9], [10] prove the optimality of *square corner* shape for two heterogeneous processors using a novel method, called "Push Technique". This method incrementally improves a partition shape by decreasing its volume of communication. It is applied to the case of three heterogeneous processors and six potentially optimal partition shapes are identified.

Beaumont et al. [11] present a non-rectangular recursive partitioning approximation algorithm (NRRP) that combines the work on non-rectangular partitioning by [7]–[10] and the recursive rectangular approximation algorithm by [3]. NRRP has an approximation ratio of $\frac{2}{\sqrt{3}}$.

Beaumont et al. [20] propose a generalization of partitioning a square into zones to three dimensions. They study partitioning a cuboid into zones of prescribed volumes, which represent the number of computations to perform while minimizing the surface of the boundaries between zones, which represent the data transfers. They prove the NP-completeness of this problem and propose a 1.51-approximation algorithm.

D. Communication-Optimal and Communication-Avoidance Algorithms

In communication-optimal and communication-avoidance algorithmic research for matrix-matrix multiplication, the theoretical lower bounds on the communications during the execution of sequential and parallel matrix-matrix multiplication are first proven using simple but accurate enough architectural models for shared memory and distributed memory machines. For a sequential algorithm, communications represent data movement between different levels of memory hierarchy. For a parallel algorithm executing on a distributed-memory machine, communications represent data transfers over the interconnection network links. An algorithm is called *communication optimal* if its communication costs (asymptotically) match the lower bounds. Communication avoidance algorithms minimize the volume of communications by employing such techniques as, for example, neighbouring processors minimizing data movement between them by executing computations using redundant copies of matrix data shared between them.

The research reviewed in this section are focused specifically on homogeneous parallel platforms. SUMMA [21] is proven to be communication-optimal for particular memory ranges for square matrix-matrix multiplication. 2.5D algorithms [22] are proven to be communication-optimal for all square matrix sizes. Solomonik et al. [22] present a 2.5D matrix multiplication algorithm that attains lower bounds on the number of words and messages communicated. Both SUMMA and 2.5D algorithms assume the processors are arranged in a two or three-dimensional grid.

Communication-optimal algorithms named BFS/DFS (Breadth-first steps/Depth-first steps) [23] have processors arranged in a hierarchy instead of a grid and employ sequential recursive techniques. They do not make any assumptions about processors and network topology. Demmel et al. [24] propose an algorithm for matrix multiplication that is communication optimal for all dimensions of rectangular matrices.

We present in this work a software implementation for parallel matrix-matrix multiplication for a specific case of three heterogeneous processors where the partition shapes are proven to be communication-optimal.

E. Software Libraries for PMM

DPLASMA [13] provides dense linear algebra factorizations for distributed architectures that feature heterogeneous many-core nodes. The dense matrix is partitioned into tiles (square blocks) and its factorization is modeled as a distributed directed acyclic graph (DAG) of tasks (operating on tiles) that are then scheduled dynamically.

Elemental [25] contains PMM implementations for homogeneous platforms based on SUMMA [21] that can be executed

using 2D and 3D processor grids with support for different matrix distributions including block-cyclic distribution.

FuPerMod [14] provides PMM implementations for heterogeneous platforms based on three performance models: a). Constant performance models (CPM), b). FPM based on piecewise linear interpolation of the speed, and c). FPM based on Akima spline interpolation of the speed.

To the best of our knowledge, there is no software library offering efficient PMM implementations based on non-rectangular partitioning of matrices. We address this gap in this work.

IV. SummaGen for Heterogeneous Platforms Based on Non-rectangular Partitions

SummaGen computes the matrix product $C = A \times B$, where A, B, and C are square matrices of size $N \times N$ ($n \times n$ blocks of size r) using p heterogeneous processors. The inputs to it are the following:

- p processors.
- Square matrices A, B, C of size $N \times N$.
- Integer array of sub-partitions, $subp$, of size $subplda \times subpldb$.
- Integer array containing heights of sub-partitions, $subph$, of size $subplda$.
- Integer array containing widths of sub-partitions, $subpw$, of size $subpldb$.

The three arrays, $\{subp, subph, subpw\}$, are used to specify the layout of partitions in the square matrices.

To illustrate the usage of these arrays, consider four examples depicting the square corner, square rectangle, block 2D rectangular, and traditional 1D rectangular partition shapes shown in the Figures 1a, 1b, 1c, 1d for three processors $\{P_0, P_1, P_2\}$ solving PMM of matrices A, B, C of size 16×16. They are considered among the six potentially optimal three processor shapes [9], [10]). The arrays for the square-corner partition shape (Figure 1a) are:

$$subplda = 3; subpldb = 3$$
$$subp[9] = \{0, 1, 1, 1, 1, 1, 1, 1, 2\}$$
$$subph[3] = \{9, 3, 4\}$$
$$subpw[3] = \{9, 3, 4\}$$

The sub-partitions in the row-major order is given by the Cartesian product of $subph \times subpw = \{9 \times 9, 9 \times 3, 9 \times 4, 3 \times 9, 3 \times 3, 3 \times 4, 4 \times 9, 4 \times 3, 4 \times 4\}$. Processor P_0 owns the sub-partition $\{9 \times 9\}$, processor P_1 owns the sub-partitions $\{9 \times 3, 9 \times 4, 3 \times 9, 3 \times 3, 3 \times 4, 4 \times 9, 4 \times 3\}$, and finally processor P_2 owns the sub-partition $\{4 \times 4\}$.

The arrays for the square-rectangle partition shape (Figure 1b) are:

$$subplda = 2; subpldb = 3$$
$$subp[6] = \{0, 0, 1, 0, 2, 1\}$$
$$subph[2] = \{12, 4\}$$
$$subpw[3] = \{9, 4, 3\}$$

The sub-partitions in the row-major order is given by the Cartesian product of $subph \times subpw = \{12 \times 9, 12 \times 4, 12 \times 3, 4 \times 9, 4 \times 4, 4 \times 3\}$ Processor P_0 owns the sub-partitions $\{12 \times 9, 12 \times 4, 4 \times 9\}$, processor P_1 owns the sub-partition $\{12 \times 3, 4 \times 3\}$, and finally processor P_2 owns the sub-partition $\{4 \times 4\}$.

The arrays for the block 2D rectangular and traditional 1D rectangular partition shapes (Figures 1c and 1d) are:

$$subplda = 2; subpldb = 2$$
$$subp[6] = \{0, 0, 1, 2\}$$
$$subph[2] = \{12, 4\}$$
$$subpw[3] = \{6, 10\}$$

$$subplda = 1; subpldb = 3$$
$$subp[6] = \{0, 1, 2\}$$
$$subph[2] = \{16\}$$
$$subpw[3] = \{8, 5, 3\}$$

The arrays ($subp, subph, subpw$) for partition shapes have to be provided manually. This is not however scalable for large number of processors. We don't consider this to be a serious drawback since the state-of-the-art solutions are proven to be optimal for only three processors and we believe that these arrays can be generated automatically.

Like SUMMA [21], the implementation of SummaGen consists of three main stages:

- Horizontal communications of rows of matrix A.
- Vertical communications of columns of matrix B.
- Local computations.

We now describe these stages.

A. Horizontal Communications for Matrix A

Each processor gathers all the necessary *A* elements required for computation of its own partition. All of these elements are stored locally in a single working matrix *WA*, to be used later during local computations.

A processor first iterates over the sub-partition rows to check if its owns at least 1 sub-partition along the row. After selecting a row (*blocki*), it now iterates over all the columns of that row (*blockj*). Now, given $subp_{blocki,blockj}$, it either broadcasts the sub-partition across the row because it owns it locally or waits to receive the sub-partition from the owner.

There is a special case, however when an entire sub-partition row is owned by a single processor. In this case, no communication is required as no other processor needs the sub-partition row. Therefore one only has to copy locally the elements from *A* into *WA*.

Figure 2) contains the implementation of the horizontal communications. A processor iterates over the sub-partition rows (line 2) using a starting point in *subp* (*myi*) and the number of rows thereafter (*block_lda*). It needs to store the element-wise index within the *A* rows as well. Thus we initialize *Alocali* to 0 beforehand and update after each row with the sub-partition height (line 42).

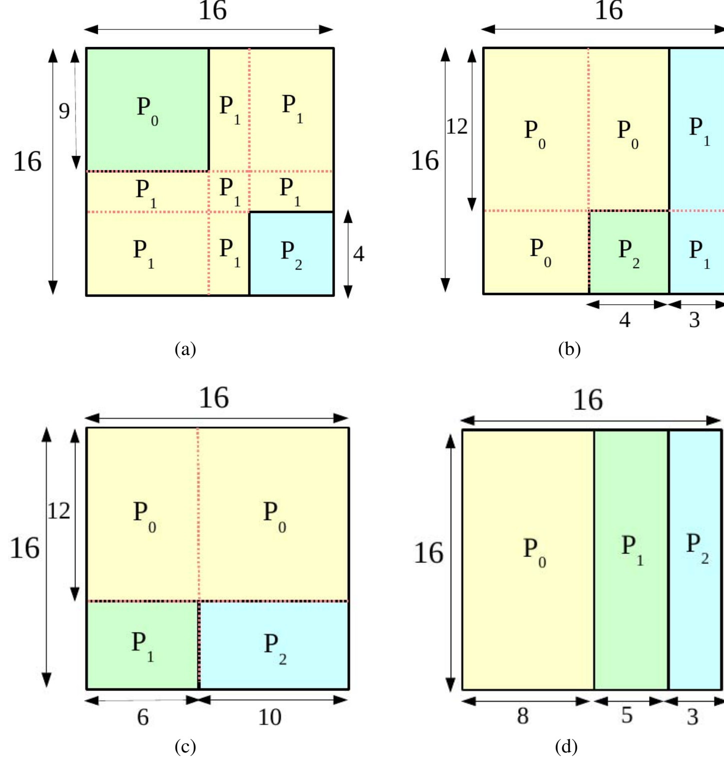


Fig. 1. a). Example: The square corner partition shape for three processors $\{P_0, P_1, P_2\}$ solving PMM of matrices A, B, C of size 16×16. P_0 and P_1 own the square corners. b). Example: The square rectangle partition shape for three processors $\{P_0, P_1, P_2\}$ solving PMM of matrices A, B, C of size 16×16. P_1 and P_2 own the square and rectangle. (c). Example: The block rectangle shape for three processors solving PMM of matrices A, B, C of size 16×16. All the processors own rectangular partitions. (d). Example: Traditional 1D rectangular shape for three processors for three processors solving PMM of matrices A, B, C of size 16×16. All the processors own rectangular partitions.

Next, we must check whether the processor actually owns a sub-partition along the row (line 3) because if it does not, then we can skip the row as it does not need any elements from this row.

The next set of lines initialize the height of temporary arrays (line 4) and the row communicator variables (line 5), which are the comm, the rank of the processor in the comm and an array of the other row ranks (*comm_ranks[new_rank] = global_rank*).

In line 8, we check the special case when the processor owns the entire sub-partition row. In this case, we can simply copy our *A* elements locally into *WA* and move onto the next row (line 9).

If the processor does not own the entire sub-partition row, the row contains a number of other processors and therefore horizontal communications ate incurred. So we start out by iterating over the columns (line 13), again keeping track of the element-wise index of the processor within the *A* columns where we initialize *Alocalj* to 0 beforehand and update after each column (line 38). We also declare the source owner of the sub-partition and initialize the width of the temporary arrays (line 16) and total size (line 17). Finally, we reallocate the temporary array (line 18) using the new total size.

Now we check whether if the processor is the owner of the sub-partition (line 20). If it owns the sub-partition, then the *source* is initialized to its rank (line 22) and the temporary array (*tmp*) is initialized to elements from its local *A* (line 25). If it does not own the sub-partition, we search *comm_ranks* (line 29) looking for the owner, that is, we find the source within the new communicator. The reason why we can't just use the *subp* rank is that we created a new communicator previously (line 6) whose rank may be different from the global communicator rank.

At this point, we are ready for communications and use *MPI_Bcast()* to transfer the elements across to each processor in the row communicator (line 33).

Finally, we copy the temporary array, which now contains the *A* elements of the sub-partition $subp_{blocki,blockj}$, into our working matrix *WA* (line 36) and move onto the next column. Eventually, after all columns have been iterated, all processors in the sub-partition row $subp_{blocki,*}$ will have the complete set of elements for that row.

We repeat for each sub-partition row and once finished, all processors will locally have all *A* elements required for computation of their partition.

B. Vertical Communications for Matrix B

Each processor gathers all the necessary *B* elements required for computation of its own partition. All of these elements are

```
int Alocali = 0;
for (int blocki = myi; blocki < myi+block_lda; blocki++) {
  if (row_contains_rank(rank, blocki, subp)) {
    int tmp_lda = subph[blocki];
    int comm_ranks[subpldb];
    get_subp_comm(&comm, &comm_rank, comm_ranks,
      size, subp, subpldb, 0, blocki * subpldb);
    if (comm_rank == size) {
      copy_matrix(&WA[Alocali * n], n,
        &A[Alocali * ldb], tmp_lda, n, n);
    } else {
      int Alocalj = 0;
      for (int blockj = 0;
          blockj < subpldb; blockj++) {
        int source;
        int tmp_ldb = subpw[blockj];
        int elem_num = tmp_lda * tmp_ldb;
        tmp = (double *)
          realloc(tmp, elem_num * sizeof(double));
        if (subp[blocki * subpldb + blockj]
            == rank) {
          source = comm_rank;
          int start_elem = Alocali * ldb +
            (Alocalj - elem_j_start);
          copy_matrix(tmp, tmp_ldb, &A[start_elem],
            tmp_lda, tmp_ldb, ldb);
        } else {
          source = 0;
          while (comm_ranks[source] !=
              subp[blocki * subpldb + blockj])
            source++;
        }
        MPI_Bcast(tmp, elem_num, MPI_DOUBLE,
          source, comm);
        int start_elem = Alocali * n + Alocalj;
        copy_matrix(&WA[start_elem], n,
          tmp, tmp_lda, tmp_ldb, tmp_ldb);
        Alocalj += subpw[blockj];
      }
    }
  }
  Alocali += subph[blocki];
}
```

Fig. 2. SummaGen: Horizontal communications for matrix A.

```
int Blocalj = 0;
for (int blockj = myj; blockj < myj+block_ldb; blockj++) {
  if (column_contains_rank(rank, blockj, subp)) {
    int tmp_ldb = subpw[blockj];
    int comm_ranks[subplda];
    get_subp_comm(&comm, &comm_rank, comm_ranks,
      size, subp, subplda, subpldb, blockj);
    if (comm_rank == size) {
      copy_matrix(&WB[Blocalj], ldb,
        &B[Blocalj], n, tmp_ldb, ldb);
    } else {
      int Blocali = 0;
      for (int blocki = 0;
        blocki < subplda; blocki++) {
        int source;
        int tmp_lda = subph[blocki];
        int elem_num = tmp_lda * tmp_ldb;
        tmp = (double *)
          realloc(tmp, elem_num * sizeof(double));
        if (subp[blocki * subpldb + blockj]
            == rank) {
          source = comm_rank;
          int start_elem =
            (Blocali - elem_i_start) * ldb + Blocalj;
          copy_matrix(tmp, tmp_ldb, &B[start_elem],
            tmp_lda, tmp_ldb, ldb);
        } else {
          source = 0;
          while (comm_ranks[source] !=
              subp[blocki * subpldb + blockj])
            source++;
        }
        MPI_Bcast(tmp, elem_num, MPI_DOUBLE,
          source, comm);
        int start_elem = Blocali * ldb + Blocalj;
        copy_matrix(&WB[start_elem], ldb,
          tmp, tmp_lda, tmp_ldb, tmp_ldb);
        Blocali += subph[blocki];
      }
    }
  }
  Blocalj += subpw[blockj];
}
```

Fig. 3. SummaGen: Vertical communications for matrix B.

stored locally in a single working matrix *WB*, to be used later during local computations.

The structure of vertical communication is identical with that of its horizontal counterpart. Therefore we will not go into detail but point out the differences in the implementation (Figure 3).

The differences between them revolve around iteration of the sub-partition columns (line 2), identifying the column communicator (line 6), and accessing and initializing the indices of the *B* and *WB* matrices.

At the end of vertical communication, all processors will locally have all the necessary *B* elements required for computation of their partition.

C. Local Computations

Computation of our local partition is the final step of SummaGen and at this stage we have the all the *A* and *B* elements required (located in *WA* and *WB*). Since the partition can be non-regular, $WA \times WB$ would not be a prudent solution.

The reason for this is that multiple processors can end up computing the same sub-partition. This is due to the fact that processors may need sub-partition row *i* and column *j* even though they don't own the sub-partition $subp_{i,j}$. An example for this can be seen in Figure 1a, where P_1 requires sub-partition row 0 and column 0 even though P_0 owns $subp_{0,0}$.

The simple solution to remove this redundancy and compute only our partition, is to compute on a per sub-partition basis. Thereby computing our total *C* partition incrementally.

In the implementation (Figure 4) we iterate over the rows and columns (lines 2 and 5) of the sub-partitions array, *subp*, during which a call to an optimized vendor DGEMM routine (line 8) for each sub-partition owned (line 7) is all that's required. At the end of which all processors will have computed strictly their own resulting *C* partition.

The routine *localDgemm* calls optimized vendor library DGEMM routine multiplying two matrices of sizes $height \times n$ and $n \times width$.

V. Algorithms for Constructing the Partition Shapes

We now describe the algorithm to arrange the partitions for a given shape. The inputs to the algorithm are the size N^2 of the dense square matrices (A, B, C) and the shape type (square corner, square rectangle, block rectangle, 1D rectangular). For the case where the speeds are constants, the procedure

```
int Clocali = 0, Clocalj;
for (int blocki = myi; blocki < myi+block_lda; blocki++) {
  height = subph[blocki];
  Clocalj = 0;
 for (int blockj = myj; blockj < myj+block_ldb; blockj++){
    width = subpw[blockj];
    if (subp[blocki * subpldb + blockj] == rank) {
      localDgemm(&height, &width, &n,
         &WA[Clocali * n], &n,
         &WB[Clocalj], &ldb,
         &C[Clocali * ldb + Clocalj], &ldb,
         &etime);
    }
    Clocalj += width;
  }
  Clocali += height;
}
```

Fig. 4. SummaGen: Local computations.

expects as input an array of three positive real numbers representing the speeds. For the case where the speeds are not constant, the inputs are discrete speed functions. The output from the procedure is a set of arrays representing the partitions: $\{subplda, subpldb, subp, subpw, subph\}$, which is input to SummaGen for execution of PMM. Proving that these algorithms construct the optimal partitions for a given shape is a future research topic.

1) Square Corner: The main steps to determine the layout of the partitions in the square corner shape (example in Figure 1a) are:

Step 1. Partition workload: For the case where the speeds are constants, the workload size N^2 is partitioned using the algorithm described in [2]. For the case where the speeds are not constant, the workload is partitioned using a data partitioning algorithm that employs load imbalancing technique and based on non-smooth FPMs [17]. The output workload distribution is given by $d = \{a_1, a_2, a_3\}$. The partitions represent areas. The areas are sorted in non-increasing order.

Step 2. Bottom left-hand square: Consider the area a_3. Determine the square n_3^2 such that $n_3^2 \approx a_3$. Place the square in the bottom left-hand corner of the square corner shape. This partition is allocated to P_2.

Step 3. Top right-hand square: Consider the area a_2. Determine the square n_2^2 such that $n_2^2 \approx a_2$ and place it in the top right-hand corner of the square corner shape. This partition is allocated to P_0. The remaining non-rectangular area is allocated to P_1.

For all the shapes, step 1 remains the same.

2) Square Rectangle: For the *square rectangle* shape (an example shown in the Figure 1b), steps 2 and 3 are below:

Step 2. Left-most rectangle: The area a_2 is divided by N to give the dimension of the smaller side of the rectangle. The other dimension of the rectangle is N. It is assigned to P_1.

Step 3. Square adjoining the rectangle: The area a_3 is then considered. Determine the square n_3^2 such that $n_3^2 \approx a_3$. Place the square next to the rectangle determined in Step 2. The square is assigned to P_2. The remaining area is allocated to P_0.

TABLE I
HCLSERVER1: SPECIFICATIONS OF THE INTEL HASWELL MULTICORE CPU, NVIDIA K40C, AND INTEL XEON PHI 3120P.

Intel Haswell E5-2670V3	
No. of cores per socket	12
Socket(s)	2
CPU MHz	1200.402
L1d cache, L1i cache	32 KB, 32 KB
L2 cache, L3 cache	256 KB, 30720 KB
Total main memory	64 GB DDR4
Memory bandwidth	68 GB/sec
NVIDIA K40c	
No. of processor cores	2880
Total board memory	12 GB GDDR5
L2 cache size	1536 KB
Memory bandwidth	288 GB/sec
Intel Xeon Phi 3120P	
No. of processor cores	57
Total main memory	6 GB GDDR5
Memory bandwidth	240 GB/sec

3) Block 2D Rectangular: For the *block 2D rectangular* shape (an example shown in the Figure 1c), the steps 2 and 3 are as follows:

Step 2. Top rectangle: The area a_1 is divided by N to give the dimension of the smaller side of the rectangle. The other dimension of the rectangle is N. It is assigned to P_0.

Step 2. Right-most rectangle: The area a_2 is now considered. The dimensions of the sides are determined as follows: $(a \approx \sqrt{a_2}, b = \frac{a_2}{a})$. This rectangle is allocated to P_1. The remaining area is allocated to P_2.

4) Traditional 1D Rectangular: For the *traditional 1D rectangular* shape (an example shown in the Figure 1d), the area a_3 is divided by N to give the dimension of the smaller side of the rectangle and the rectangle is assigned to P_2. The area a_2 is divided by N to give the dimension of the smaller side of the rectangle, which is assigned to P_1. The remaining rectangle is allocated to P_0.

VI. EXPERIMENTAL RESULTS

We perform our experiments on our research server *HCLServer1*, which contains an Intel Haswell multicore CPU, an Nvidia K40c GPU, and an Intel Xeon Phi 3120P. The specifications of the three devices are given in Table I. The OS on the server is CentOS 7.2.1511.

We use three abstract processors ($p = 3$) described earlier in the introduction in the PMM applications. We call the abstract processors AbsCPU, AbsGPU, and AbsXeonPhi to aid our exposition in this section. For the abstract processor AbsCPU, the local computations are performed using DGEMM routine provided in Intel MKL BLAS [26]. For the abstract processors AbsGPU and AbsXeonPhi, we developed two packages that perform out-of-core matrix multiplication of large dense matrices on them. For AbsGPU, ZZGemmOOC out-of-core package [27] reuses CUBLAS [28] for in-core DGEMM invocations. For Xeon Phi, XeonPhiOOC out-of-core package [27] reuses MKL BLAS [26] for in-core DGEMM invocations. The Intel MKL and CUDA versions used are 2017.0.2 and 7.5.

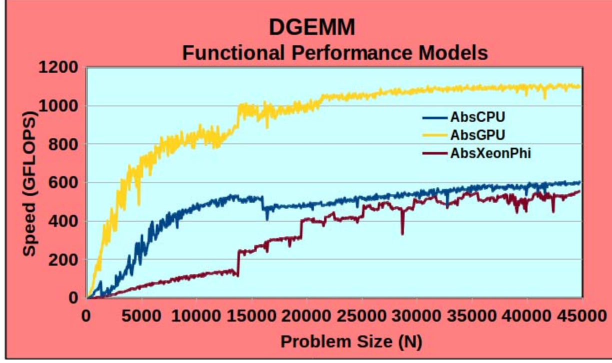


Fig. 5. Speed functions/Performance profiles of the abstract processors (AbsCPU, AbsGPU, AbsXeonPhi) computing matrix multiplication of two square matrices of size $N \times N$.

The MPI implementation used is Intel MPI 5.1.3. One MPI process is mapped to one abstract processor.

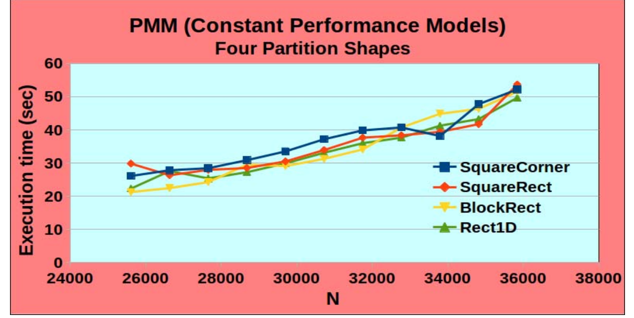
The full speed functions or performance profiles of the three abstract processors are shown in Figure 5. For each data point in the functions, we measure the execution time of each abstract processor when all the other abstract processors are also executing the same workload simultaneously, thereby taking into account the influence of resource contention. The execution time for accelerators includes the time taken to transfer data between the host and devices. For each data point, the workload is a matrix multiplication for two dense square matrices of size $x \times x$ whose performance (speed) is calculated as $\frac{2 \times x^3}{t}$, where t is the execution time.

The full functions are thus constructed using an automated procedure. For problem sizes exceeding $(N = 22592)$, there are memory failures since the problem size allocated to one or more processors does not fit into the main memory of the processor. Therefore, we use out-of-core implementations [27].

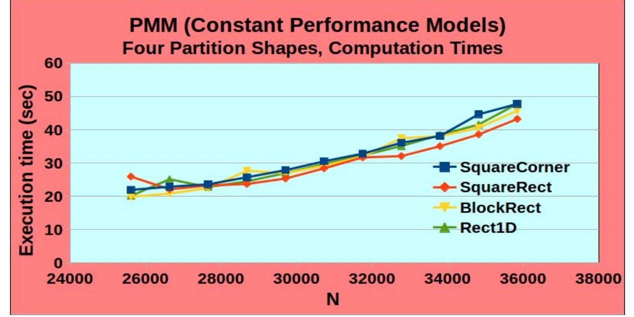
To obtain an experimental data point, the application is executed repeatedly until the sample mean lies in the 95% confidence interval and a precision of 0.025 (2.5%) has been achieved. For this purpose, Student's t-test is used assuming that the individual observations are independent and their population follows the normal distribution. We verify the validity of these assumptions using Pearson's chi-squared test. When we mention a single number such as floating-point performance (in TFLOPs), it is assumed that we are refering to the sample mean determined using the Student's t-test.

To study the performance of our implementation, we use four shapes proven to be optimal for three heterogeneous processors where the speeds of the processors are represented by constant functions of problem size [9], [10]. They are a). Square corner, b). Square rectangular, c). Block 2D rectangular, and d). Traditional 1D rectangular.

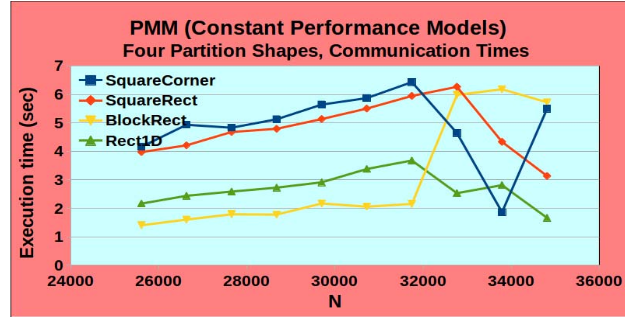
We consider two different cases: a). The speeds of the processors are represented by constant functions of problem size (constant performance models) and b). The speeds of the processors are represented by non-constant functions of problem size (functional performance models).



(a)



(b)



(c)

Fig. 6. a). Execution times of PMM for the four partition shapes multiplying two dense square matrices of size $N \times N$. The speeds of the processors are constant functions of problem size. b). Computation times during the execution of PMM for the four partition shapes. c). Communication times during the execution of PMM for the four partition shapes.

A. Constant Performance Models

We compare the performances of PMM for the four partition shapes where the speeds of the processors are represented by constant functions of problem size. The range of problem sizes (N) tested is $\{25600, \cdots, 35840\}$. Figure 5 shows that the relative speeds of the three abstract processors AbsCPU, AbsGPU, and AbsXeonPhi are nearly constant in this range. Their speeds used in the experiments are $\{1.0, 2.0, 0.9\}$.

The execution times of PMM for the four partition shapes are shown in Figure 6a. They are equal with a maximum percentage difference of 23% for the problem size $N = 25600$ and an average percentage difference of 8%. The peak performance observed in the experiments is 2.10 TFLOPs for

square rectangle shape for problem size $N = 38416$. This is 84% of the theoretical peak floating-point performance of the machine, which is 2.50 TFLOPs obtained by summation of the theoretical peaks of the three abstract processors. The average performance is 70% of the theoretical peak floating-point performance of the machine.

Figures 6b and 6c show the computation and communication times during the execution of PMM for the four partition shapes. The computation and communication times are the maximums of the computation and communication times of the abstract processors. The communication times include only time for the MPI communications and not data transfer times between the host and the accelerators. The parallel execution times are dominated by computation times. The communication times, however, are different. We hope to study further the cause for these differences using realistic and accurate communication models in our future work.

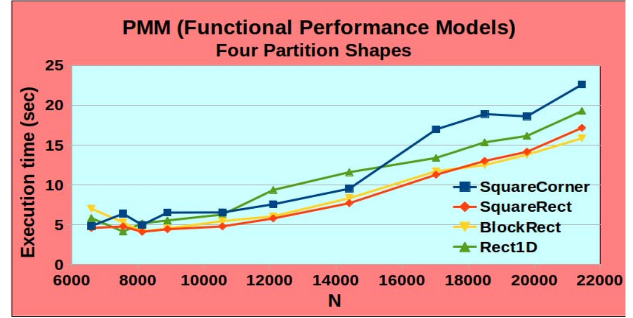
B. Non-constant Performance Models

For each shape, we compare the performances of PMM for a matrix decomposition determined by data partitioning algorithm that employs load imbalancing technique and is based on non-smooth FPMs [16], [17]. Optimal solutions found by this algorithm are uneven workload distributions. They minimize the parallel execution time of computations but may not load balance the application in terms of execution time.

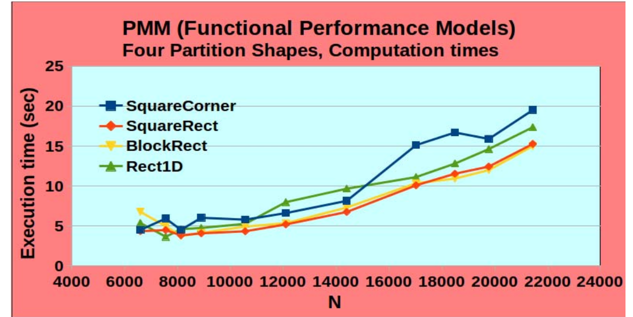
The range of problem sizes (N) tested is $\{1024, \cdots, 20480\}$. For this range, the speed functions of the three abstract processors shown in the Figure 5 are non-constant functions of the problem size. The discrete speed function of AbsXeonPhi is smooth between 64^2 to 13760^2. The maximum variations occur for problem sizes in the range $[12800^2, 19200^2]$. The variations increase however for larger problem sizes ($\geq 13824^2$) where out-of-card computations are invoked. Unlike AbsXeonPhi, the variations decrease for AbsCPU and AbsGPU as problem size increases. The load imbalancing data partitioning algorithm [17] exploits these variations to determine optimal workload distribution that minimizes the time of computations during the execution of PMM.

Figure 7a compares the execution times of PMM for the four partition shapes. Figures 7b and Figure 7c compares the execution time of computations and communications. The peak performance observed in the experiments is 1.80 TFLOPs for square rectangle shape for problem size $N = 35008$. This is 72% of the theoretical peak floating-point performance of the machine.

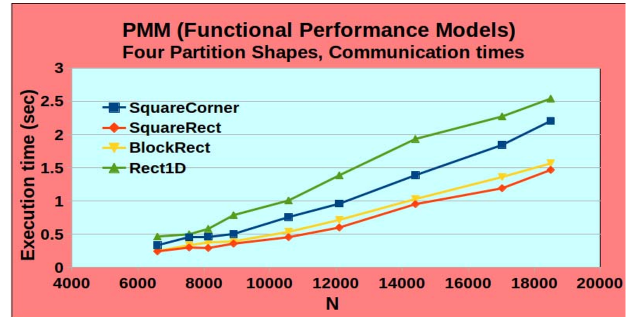
The partition shapes *square rectangle* and *block rectangular* perform better than the other two shapes. Their low parallel execution times are due to their low computation as well as communication times. In our future work, we hope to supplement our experimental findings with a theoretical study of the optimality of these shapes when the speeds of the processors are represented by non-constant functions of problem size.



(a)



(b)



(c)

Fig. 7. a). Execution times of PMM for the four partition shapes multiplying two dense square matrices of size $N \times N$. The speeds of the processors are non-constant functions of problem size. b). Computation times during the execution of PMM for the four partition shapes. c). Communication times during the execution of PMM for the four partition shapes.

C. Study of Energy Consumptions for Partition Shapes Based on Constant Performance Models

In general, two types of energy consumption can be considered, dynamic and static. We define the static energy consumption as the energy consumption of the platform without the given application execution. Dynamic energy consumption is calculated by subtracting this static energy consumption from the total energy consumption of the platform during the given application execution. That is, if P_S is the static power consumption of the platform, E_T is the total energy consumption of the platform during the execution of an application, which takes T_E seconds, then the dynamic energy E_D can be calculated as,

$$E_D = E_T - (P_S \times T_E) \quad (5)$$

Our experimental platform *HCLServer1* is facilitated with one WattsUp Pro power meter that sits between the wall A/C outlets and its input power sockets. The power meter captures the total power consumption of the server. It has data cable connected to one USB port of the server. A script written in Perl collects the data from the power meter using the serial USB interface. The execution of the script is non-intrusive and consumes insignificant power. The power meter is periodically calibrated using an ANSI C12.20 revenue-grade power meter, Yokogawa WT210. The maximum sampling speed of the power meter is one sample every second. The accuracy specified in the data-sheets is $\pm 3\%$. The minimum measurable power is 0.5 watts. The accuracy at 0.5 watts is ± 0.3 watts. The static power consumption of the platform is 230 Watts.

We use *HCLWattsUp* API [29], which gathers the readings from the power meter to determine the dynamic energy consumption during the execution of PMM application. *HCLWATTSUP* has no extra overhead and therefore does not influence the energy consumption of the application execution.

Fans are significant contributors to energy consumption. On our platform, fans are controlled in two zones: a) zone 0: CPU or System fans, b) zone 1: Peripheral zone fans. There are 4 levels to control the speed of fans:

- *Standard*: BMC control of both fan zones, with CPU zone based on CPU temp (target speed 50%) and Peripheral zone based on PCH temp (target speed 50%)
- *Optimal*: BMC control of the CPU zone (target speed 30%), with Peripheral zone fixed at low speed (fixed 30%)
- *Heavy IO*: BMC control of CPU zone (target speed 50%), Peripheral zone fixed at 75%
- *Full*: all fans running at 100%

To rule out the contribution of fans in dynamic energy consumption, we set the fans at full speed before executing the PMM applications. When set at full speed, the fans run constantly at $\sim$13400 rpm until they are set to a different speed level. In this way, energy consumption due to fans is included only in the static power consumption of the platform. We monitor the temperature of our platform and speeds of the fans (with *Full* setting) with the help of Intelligent Platform Management Interface (IPMI) sensors, both with and without the application run. We found an insignificant difference in the speeds of fans in both scenarios.

Figure 8 shows the dynamic energy consumptions for the four partition shapes for the PMM application employing constant performance model. The range of problem sizes (N) tested is $\{25600, \cdots, 35840\}$. One can see that the dynamic energy consumptions are equal. This does not, however, suggest that the shapes are optimal for dynamic energy. We aim to further develop methods to prove whether these shapes are optimal for dynamic energy.

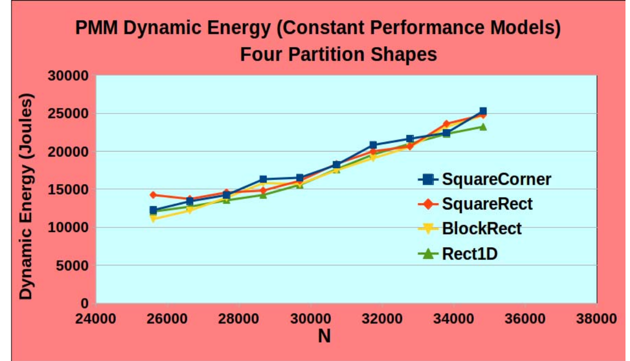


Fig. 8. Dynamic energy consumptions of the PMM applications for the four partition shapes.

VII. Conclusion

Parallel matrix-matrix multiplication (PMM) of dense matrices is a foundational kernel of parallel linear algebra libraries in high performance computing (HPC) domain. The problem of finding the optimal shape of matrices that minimized the computation time and the overall volume of communications in PMM on heterogeneous platforms has been the center of research and can be classified into two distinct threads.

The first thread purely focused on rectangular partitions whereas the second thread relaxed the rectangular partition constraint to allow non-rectangular partitions. While efficient implementations of PMM based on rectangular partitions exist [13], [14], the results in the second thread are entirely theoretical. There is no software implementation for PMM that would facilitate experimental studies of the practical performance and optimality of the proposed partition shapes.

To address this shortcoming, we proposed an implementation of PMM based on non-rectangular partitions on heterogeneous platforms called *SummaGen*. To study its performance, we compared the performances of PMM for four partition shapes proven optimal for three processor case where speeds of the processors are represented by constant function of problem size. We employ for our experiments a hybrid heterogeneous multi-accelerator NUMA node comprising of three heterogeneous computing devices, a dual-socket Intel Haswell multicore CPU, an Nvidia K40 GPU, and an Intel Xeon Phi 3120P. We showed that the four shapes exhibit equal performances (with an average percentage difference of 8%) for a range of problem sizes where the speeds are constant confirming their optimality in practice. We demonstrated that the four shapes exhibit equal dynamic energy consumptions for a range of problem sizes where the speeds are constant. Whether these shapes are optimal for dynamic energy is a subject for our current research.

To understand the behaviour of the shapes when the speeds of the processors are represented by non-constant function of problem size, we compare the performances of PMM employing load imbalancing matrix decomposition method that takes as input non-smooth functional performance models

of the processors. We find that the partition shapes *square rectangle* and *block rectangle* perform better than the other two shapes. In our future work, we hope to formally prove the optimality of these shapes when the speeds of the processors are represented by non-constant functions of problem size

The peak and average performances of the implementation are 80% and 70% of the theoretical peak floating-point performance of the machine.

The software implementation of SummaGen is located at [30]. For our future work, we will study the efficiency of *SummaGen* for distributed-memory nodes and large clusters.

ACKNOWLEDGMENT

This publication has emanated from research conducted with the financial support of Science Foundation Ireland (SFI) under Grant Number 14/IA/2474.

REFERENCES

[1] A. Kalinov and A. Lastovetsky, "Heterogeneous distribution of computations solving linear algebra problems on networks of heterogeneous computers," *Journal of Parallel and Distributed Computing*, vol. 61, no. 4, pp. 520 – 535, 2001.

[2] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert, "Matrix multiplication on heterogeneous platforms," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 10, Oct. 2001.

[3] H. Nagamochi and Y. Abe, "An approximation algorithm for dissecting a rectangle into rectangles with specified areas," *Discrete Appl. Math.*, vol. 155, no. 4, Feb. 2007.

[4] A. Lastovetsky and R. Reddy, "Two-dimensional matrix partitioning for parallel computing on heterogeneous processors based on their functional performance models," in *7th International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar 2009)*, Lecture Notes in Computer Science, vol. 6043, Springer. Lecture Notes in Computer Science, vol. 6043, Springer, 25/9/2009 2010, pp. 112–121.

[5] D. Clarke, A. Lastovetsky, and V. Rychkov, "Column-based matrix partitioning for parallel matrix multiplication on heterogeneous processors based on functional performance models," in *9th International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar'2011)*, Lecture Notes in Computer Science 7155, Springer. Lecture Notes in Computer Science 7155, Springer, August 29, 2011 2012, pp. 450–459.

[6] A. Fügenschuh, K. Junosza-Szaniawski, and Z. Lonc, "Exact and approximation algorithms for a soft rectangle packing problem," *Optimization*, vol. 63, no. 11, 2014.

[7] B. Becker and A. Lastovetsky, "Matrix multiplication on two interconnected processors," in *Proceedings of the 8th IEEE International Conference on Cluster Computing*. IEEE Computer Society, 25-28 Sept 2006 2006.

[8] B. Becker and A. L. Lastovetsky, "Towards data partitioning for parallel computing on three interconnected clusters," in *Proceedings of the 6th International Symposium on Parallel and Distributed Computing (ISPDC)*. IEEE Computer Society, 5-8 July 2007 2007.

[9] A. DeFlumere and A. Lastovetsky, "Searching for the optimal data partitioning shape for parallel matrix matrix multiplication on 3 heterogenous processors," in *23rd International Heterogeneity in Computing Workshop (HCW)*, IEEE Computer Society. IEEE Computer Society, 19 May 2014.

[10] A. DeFlumere and A. L. Lastovetsky, "Optimal data partitioning shape for matrix multiplication on three fully connected heterogeneous processors," in *12th International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar'2014)*, 25 August 2014.

[11] O. Beaumont, L. Eyraud-Dubois, and T. Lambert, "A new approximation algorithm for matrix partitioning in presence of strongly heterogeneous processors," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 474–483.

[12] O. Beaumont, B. A. Becker, A. Deflumere, L. Eyraud-Dubois, T. Lambert, and A. Lastovetsky, "Recent advances in matrix partitioning for parallel computing on heterogeneous platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 1, pp. 218–229, 2019.

[13] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Hérault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. Yarkhan, and J. J. Dongarra, "Distributed Dense Numerical Linear Algebra Algorithms on massively parallel architectures: DPLASMA," in *Proceedings of the 25th IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW'11), PDSEC 2011*, 2011.

[14] D. Clarke, Z. Zhong, V. Rychkov, and A. Lastovetsky, "FuPerMod: a software tool for the optimization of data-parallel applications on heterogeneous platforms," *The Journal of Supercomputing*, vol. 69, pp. 61– 69, 2014.

[15] Z. Zhong, V. Rychkov, and A. Lastovetsky, "Data partitioning on multicore and multi-GPU platforms using functional performance models," *Computers, IEEE Transactions on*, vol. 64, no. 9, pp. 2506–2518, 2015.

[16] A. L. Lastovetsky and R. Reddy, "New model-based methods and algorithms for performance and energy optimization of data parallel applications on homogeneous multicore clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 4, pp. 1119–1133, April 2017.

[17] H. Khaleghzadeh, R. Reddy, and A. Lastovetsky, "A novel data-partitioning algorithm for performance optimization of data-parallel applications on heterogeneous HPC platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 10, pp. 2176–2190, 2018.

[18] A. Lastovetsky and R. Reddy, "Data partitioning with a functional performance model of heterogeneous processors," *International Journal of High Performance Computing Applications*, vol. 21, no. 1, pp. 76–90, 2007.

[19] J. Li, X. Li, G. Tan, M. Chen, and N. Sun, "An optimized large-scale hybrid dgemm design for cpus and ati gpus," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12. ACM, 2012, pp. 377–386.

[20] O. Beaumont, L. Eyraud-Dubois, and T. Lambert, "Cuboid partitioning for parallel matrix multiplication on heterogeneous platforms," in *Euro-Par 2016: Parallel Processing*. Springer International Publishing, 2016, pp. 171–182.

[21] R. A. van de Geijn and J. Watts, "SUMMA: scalable universal matrix multiplication algorithm," *Concurrency: Practice and Experience*, vol. 9, no. 4, pp. 255–274.

[22] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5d matrix multiplication and lu factorization algorithms," in *Euro-Par 2011 Parallel Processing*, E. Jeannot, R. Namyst, and J. Roman, Eds. Springer Berlin Heidelberg, 2011, pp. 90–109.

[23] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz, "Communication-optimal parallel algorithm for strassen's matrix multiplication," in *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '12. ACM, 2012, pp. 193–204.

[24] J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger, "Communication-optimal parallel recursive rectangular matrix multiplication," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '13. IEEE Computer Society, 2013, pp. 261–272.

[25] J. Poulson, B. Marker, R. A. van de Geijn, J. R. Hammond, and N. A. Romero, "Elemental: A new framework for distributed memory dense matrix computations," *ACM Trans. Math. Softw.*, vol. 39, no. 2, Feb. 2013.

[26] Intel MKL BLAS. [Online]. Available: https://software.intel.com/en-us/mkl

[27] H. Khaleghzadeh, Z. Zhong, R. Reddy, and A. Lastovetsky, "Out-of-core implementation for accelerator kernels on heterogeneous clouds," *The Journal of Supercomputing*, vol. 74, no. 2, Feb 2018.

[28] CUBLAS: Dense linear algebra on GPUs. [Online]. Available: https://developer.nvidia.com/cublas

[29] Heterogeneous Computing Laboratory, University College Dublin, "HCLWattsUp: API for power and energy measurements using WattsUp Pro Meter," 2016. [Online]. Available: http://git.ucd.ie/hcl/hclwattsup

[30] S. Patton and R. R. Manumachu, "hclsummagen: Efficient implementation of parallel matrix-matrix multiplication based on non-rectangular partitions for heterogeneous hybrid platforms," 2018. [Online]. Available: https://git.ucd.ie/manumachu/hclsummagen