University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

An HTMT Performance Prediction Case Study: Implementing Cannon's Dense Matrix Multiply Algorithm

José Nelson Amaral, Guang R. Gao,
Phillip Merkey†, Thomas Sterling‡,
Zachary Ruiz, Sean Ryan

CAPSL Technical Memo 26
February 17, 1999

†Center of Excellence in Space Data and Information Sciences, Goddard Space Flight Center, NASA, Greenbelt, MA. http:*//cesdis.gsfc.nasa.gov/people/merk/merk.html*, email: *merk@cesdis.gsfc.nasa.gov*
‡Center for Advanced Computer Research, California Institute of Technology, Pasadena, CA, USA. http:*//www.cacr.caltech.edu/∼tron/*.

University of Delaware • 140 Evans Hall • Newark, Delaware 19716 • USA
http://www.capsl.udel.edu • ftp://ftp.capsl.udel.edu • capsladm@capsl.udel.edu

Abstract

This technical memo presents a case study of performance prediction for the Hybrid Technology Multi-Threaded architecture. We use a dense matrix multiply to introduce an analytical methodology to predict the performance of the percolation model on the HTMT architecture. HTMT introduces a percolation program and execution model that (1) is explicitly multi-threaded; (2) incorporates global memory address space; and (3) explicitly exposes the HTMT memory hierarchy to the programmer. The percolation model extends dynamic prefetching to allow the management of contexts that include data, program instructions, and control states.

An analytical study of our algorithm and the percolation process is used to determine the number of operations that are performed in each memory region and the amount of data that is exchanged between regions. Current estimates for the processing power, network performance and storage capacity in each memory region are injected into the analytic study to predict the performance of this algorithm on HTMT. The resulting calculations indicate that with current design parameters, it is possible to multiply dense matrices of dimensions $208,000 \times 208,000$ in 16.2 seconds, resulting in an estimation of 1.1 petaFLOPS.

The contributions of this work include a detailed description of the mapping of Cannon's matrix multiply algorithm into the SPELLS; the specification of the percolation process that is required to move data into and out of the cryostatic core of the HTMT architecture in order to execute Cannon's algorithm; the introduction of an analytical methodology to estimate the complexity of the different phases of the percolation process; the use of current estimates for the hardware performance of the subsystems of HTMT to produce an actual performance prediction for the execution of the algorithm.

Contents

List of Figures

1 Introduction

This document presents an analytical performance prediction for the implementation of a dense matrix multiply algorithm in the Hybrid Technology Multi-Threading (HTMT) architecture [11]. The HTMT subsystems are built from new technologies: super-conducting processor elements (called *SPELLs* [7]), a network based on RSFQ (Rapid Single Flux Quantum) logic devices (called *CNET* [19]), "Processor In Memory" (PIM) technology [13], a high-performance optical packet switched network (called *Data Vortex* [5], optical holographic storage devices (called HRAM [16]), and fine grain multi-threaded computing technology [12].

From a computer architecture point of view, two characteristics distinguish HTMT: (1) the latency to fetch data or code from the next level in the memory hierarchy is of the order of tens of thousands of cycles of the fastest processors in the machine; (2) processing is distributed across the memory hierarchy, enabling the execution of data transformation closer to the storage devices.

The percolation programming model has been under development at the University of Delaware for a year [10, 17, 9]. This model establishes the following guiding principles to take advantage of the architectural characteristics above: (1) the instructions of a program segment must be paired with the data required by those instructions before they are shipped to the vicinity of the fastest processors in the machine; (2) the programmer has a view of a global address space for the entire architecture, but data movements should be coded explicitly and should use split phase transactions to prevent processors from wasting cycles while waiting for long latency operations to complete; (3) the program explicitly divides the computation to be performed into threads of computation and introduces a thread switching when a long latency operation is required.

The remainder of the document is organized as follows. Section 2 gives a high level description of the HTMT architecture. Section 3 summarizes the percolation model. Section 4 describes how the Cannon matrix multiply algorithm is mapped to the HTMT architecture using the percolation model and identifies the components of the analytic performance model. Section 5 presents an analytical study of the time complexity of each phase of the percolation process based on the sizes of blocks and sub-blocks used in the matrix multiplication. Section 6 discusses the hardware parameters of the HTMT architecture that are relevant to the performance prediction of the algorithm. Section 7 combines the algorithm complexity information with hardware estimations to produce an overall performance estimation for the year 2007 implementation of HTMT.

2 The HTMT Architecture

Figure 1 presents a simplified view of the HTMT architecture. In the center are the super-conducting processors, their associated cryostatic memory (CRAM), and the RSFQ based network CNET. We often refer to this center region as the cryostatic region because the super-

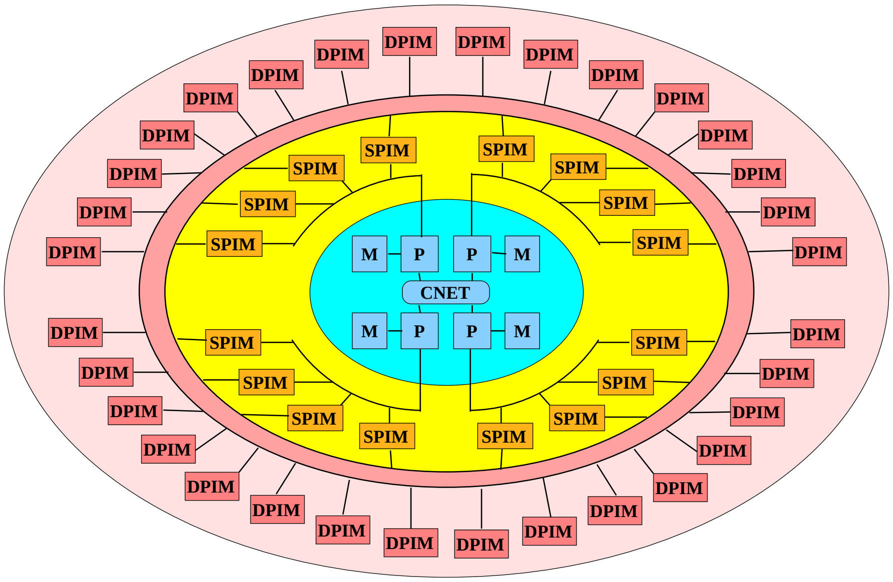

Figure 1: A simplified view of the processing and memory regions of the HTMT architecture.

conducting elements require it to be kept at a very low temperature. Outside the cryostatic region are the SRAM memory modules and the corresponding processors in memory (SPIM). The ring outside the SPIM area represents the data vortex that is an optical interconnection network used for communication between the SRAM region and the DRAM region. The DRAM region outside of the data vortex is also formed by memory modules associated with processors in memory (DPIM). Not represented in the figure are the holographic storage devices, the farms of disks and the tape robots that will be used for massive storage of data. Figure 1 represents four SPELLs with the proportional number of SPIM and DPIM modules. The final configuration of the HTMT machine is expected to have up to 4096 SPELLs.

Observe in Figure 1 that any DPIM module can communicate with any SPIM module through the data vortex, and also the DPIM and the SPIM modules can communicate among themselves using the data vortex. Also any two SPELLs can communicate with each other through the CNET. However a given SPELL can only communicate directly with a local set of SPIM modules.

3 The Percolation Model

Considering the machine model presented in Figure 1 the percolation process starts in the DRAM region. Data and code are selected and prepared in that region and sent to the SRAM region. Under the percolation model of execution, the entire computation is explicitly threaded. A unit formed by the data plus the code is designated a *parcel*, therefore the function that specifies such a unit is called a *parcel threaded function*. Because of the way the machine is laid out with the fast processors in the center and the memory modules with higher storage

```
BASEPERCOLATION
1.  Data transformations are performed in DRAM;
2.  The transformed data percolates inward to SRAM;
3.  while SRAM parcel retiring condition not true;
4.          A parcel threaded function f( ) becomes enabled in SRAM;
5.          Data is split in smaller pieces (and transformed again) in SRAM;
6.          The transformed data percolates inward to CRAM;
7.          The code of f( ) percolates inward to CRAM;
8.          while CRAM parcel retiring condition not true;
9.                  f( ) is executed in a SPELL;
10.                 More data is requested from SRAM;
11.                 Results are percolated outward to SRAM;
12.         f( ) is retired from CRAM;
13.         Results are arranged in larger blocks (and transformed) in SRAM;
14.         Results are percolated outward to DRAM;
15. Dual transformations are performed in the outward percolated data in DRAM
```

Figure 2: Outline of steps in the base percolation process.

capacity in the outer regions, we call the movement of parcels towards the fast processors *inward percolation* and the movement of parcels towards the large memory storage units *outward percolation*. The base percolation model can be illustrated by examining the exchange of parcels between the SRAM and the CRAM regions as illustrated in the steps listed in Figure 2.

The linear listing of events in Figure 2 might give an impression that the percolation is a sequential process. It is not! New data transformations might take place in the DRAM while the previously transformed data is used by the SRAM/CRAM processors. A pipeline structure can be coded to allow the SPELLs to process blocks of data while the SRAM replies to requests of more data or stores away results from previous computation. Depending on the application some of the events above might not be necessary. For instance, in many applications it is not necessary to percolate results outward to the SRAM after every execution of $f(\,)$ in the CRAM.

The programming language used to express the percolation model is an extension of Portable Threaded-C, a fine grain explicitly multi-threaded language [10, 18]. The HTMT program execution model is described in [9]. A complete specification of HTMT-C can be found in [2].

4 A Programming Example

Consider the problem of multiplying two large, dense matrices on the HTMT machine. Assume the $M \times M$ matrices $\mathbf{A}$ and $\mathbf{B}$ are initially stored in HRAM in row first order. We wish to compute and store their product, the matrix $\mathbf{C}$, in HRAM, also in row first order.

The HTMT architecture offers limited storage space in the CRAM region and a high latency to fetch data from SRAM. Also, although the memory address space is shared, the latency to access a remote CRAM might be six times higher (or more if there is contention in the network) than the access to a local CRAM module. Therefore it seems appropriate to use a distributed memory algorithm that requires a minimal amount of temporary space within the CRAM. A well-known algorithm with these characteristics is Cannon's dense matrix multiply algorithm [6, 14].

The percolation of the data from the outside layers of the HTMT architecture into its cryostatic core is more easily controlled if we treat the matrices as if they were block matrices where the block themselves consist of sub-blocks. We assume that each one of these matrices are divided into $t \times t$ blocks of size b_s. If we denote the ij^{th} block of $\mathbf{C}$ by $\mathbf{C}_{i,j}$ (likewise for $\mathbf{A}$ and $\mathbf{B}$), then the computation of $\mathbf{C}_{m,n}$ requires t, $b_s \times b_s$ block multiplications according to:

$$\mathbf{C}_{m,n} = \sum_{k=0}^{t} \mathbf{A}_{m,k} \times \mathbf{B}_{k,n} \tag{1}$$

In this section we discuss Cannon's dense matrix multiply algorithm that is used to perform the sub-block multiplications in the SPELLs. We proceed to specify the percolation process to move data from HRAM to CRAM and to move results produced in the SPELL all the way back to HRAM.

Our percolation strategy consists of (1) establishing the operations that are to be performed in the SPELLs to implement Cannon's algorithm; (2) determining which sub-blocks of $\mathbf{A}$ and $\mathbf{B}$ need to percolate to each SPELL; (3) designing data transformations to be performed at the DRAM level that will enable the formation of *bundles* [1] of sub-blocks to be percolated from the SRAM to the SRAM; (4) establishing a mechanism to allow the interleaving of communication and computation in the SPELLs; (5) computing the largest sub-blocks that can fit in CRAM according to Cannon's algorithm and the interleaving mechanism; (6) computing the largest bundles that can fit in SRAM; (7) designing the algorithm to perform the percolation from SRAM to CRAM.

4.1 Cannon's Algorithm

Although the SPELLs and CRAMs have a shared address space, the use of a distributed memory algorithm such as Cannon's is justified by the difference in the latency incurred for data access in a local CRAM versus accessing data in a remote CRAM, and by the data re-use in a dense matrix multiplication. Moreover the uniform communication pattern in Cannon's algorithm makes efficient use of the CNET.

We divide the blocks of the matrices in collections of sub-blocks to make it adequate for the percolation of the data from the outside layers of HTMT into its cryostatic core and for the percolation of the results produced by the multiplication from the core to the outside layers. The

[1] The *bundle* of sub-blocks that percolates from DRAM to SRAM does not form a logical matrix block. Each block is distributed across all the DRAMs — and later across the SRAMs and CRAMs. In other words, the sub-blocks are bundled together at the DRAM according to their SPELL destinations and not according to their membership to a logical block

maximization of computation to communication ratio in massively parallel processors translates to the maximization of temporal locality in HTMT.

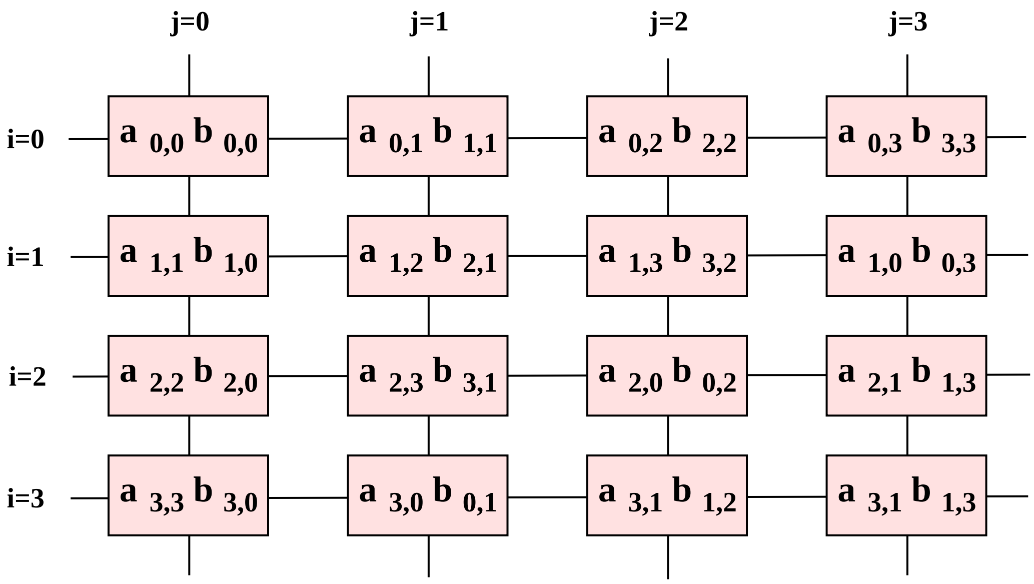

Figure 3: Sub-blocks initially percolated to each SPELL in the grid.

The computation of a block of **C** requires t block multiplications. Each block multiplication is performed by s^2 SPELLs using Cannon's algorithm [6, 14]. Following Cannon's algorithm, the SPELLs are logically organized in a two dimensional grid. Each SPELL in the grid communicates with its four neighbors[2]. Each matrix block has dimension $b_s \times b_s$ and is further divided into $s \times s$ sub-blocks of size $b_c \times b_c$; thus $b_s = s\ b_c$; $\mathbf{a}_{\alpha,\beta}$ is the sub-block in row α and column β of the matrix.

To implement Cannon's algorithm the sub-blocks of **A** and **B** must percolate to the grid of SPELLs in a skewed fashion, as displayed in Figure 3.

The first sub-block multiplication is performed by each SPELL as soon as the sub-blocks arrive in the grid of SPELLs. When this multiplication is complete, the sub-blocks of **A** are shifted leftward, and the sub-blocks of **B** are shifted upward. Figure 4 illustrates this sub-block movement. The figure shows the position of the sub-blocks after the first movement.

While the sub-blocks are been shifted in the grid of SPELLs, each SPELL accumulates a partial result for a sub-block of **C**. The partial result computed by the SPELL (i, j) when the blocks $\mathbf{A}_{0,0}$ and $\mathbf{B}_{0,0}$ are percolated to the grid of SPELLs [3] is given by the following equation:

$$C_{i,j} = \sum_{k=0}^{s-1} A_{i,k} \times B_{k,j} \tag{2}$$

[2]One can imagine this grid forming a toroid, such that there are no edges. However in this discussions we will always present the grid flattened out.

[3]The percolation of the block $\mathbf{A}_{i,j}$ to the grid of SPELLs implies the percolation of one sub-block of $\mathbf{A}_{i,j}$ from each SRAM to each CRAM in the architecture.

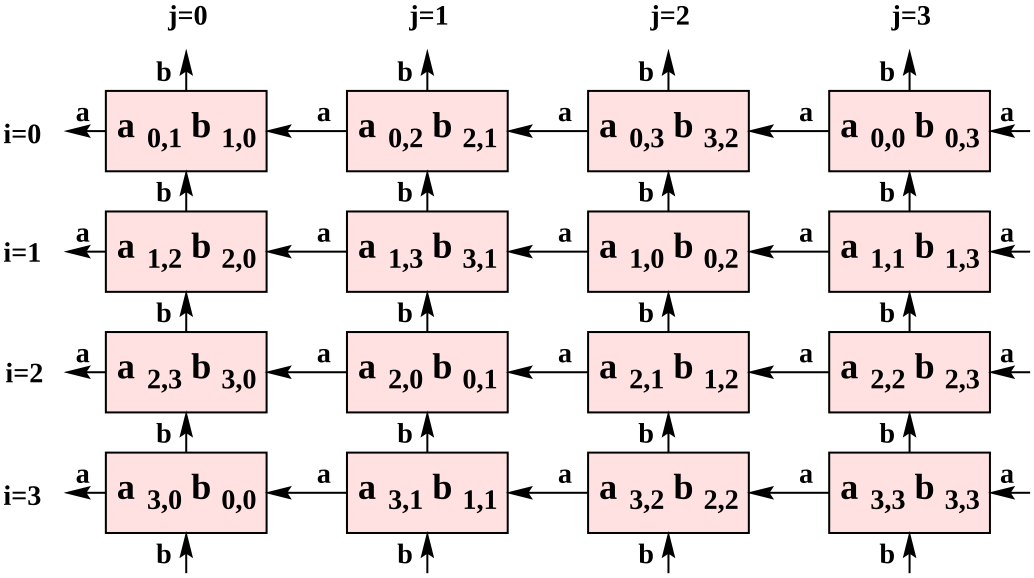

Figure 4: Sub-blocks in each SPELL after the first movement of blocks of **A** and sub-blocks of **B** between neighbor processors in the grid.

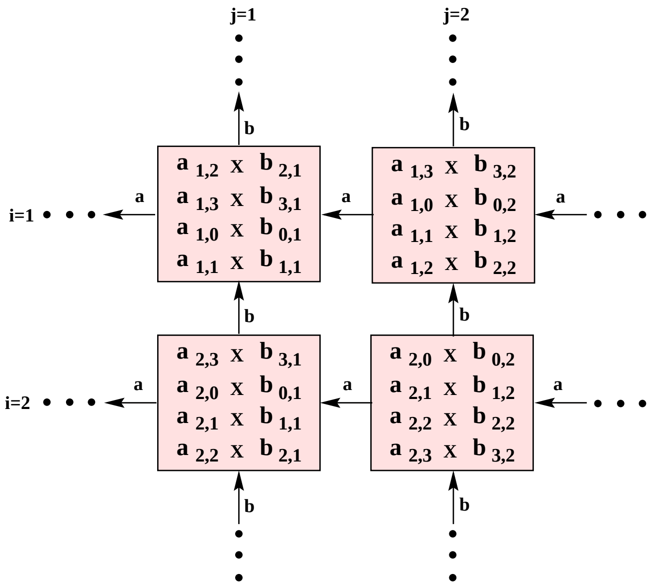

Figure 5: Sequence of sub-block multiplications that are performed by each SPELL in the grid.

Figure 5 shows the sub-block multiplications that are performed by four SPELLs in a 4×4 grid. Each rectangle in Figure 5 represents one SPELL. The sub-block multiplication listed on the top of the rectangle is the first multiplication to be performed and the one listed on the bottom of the rectangle is the last one to be performed.

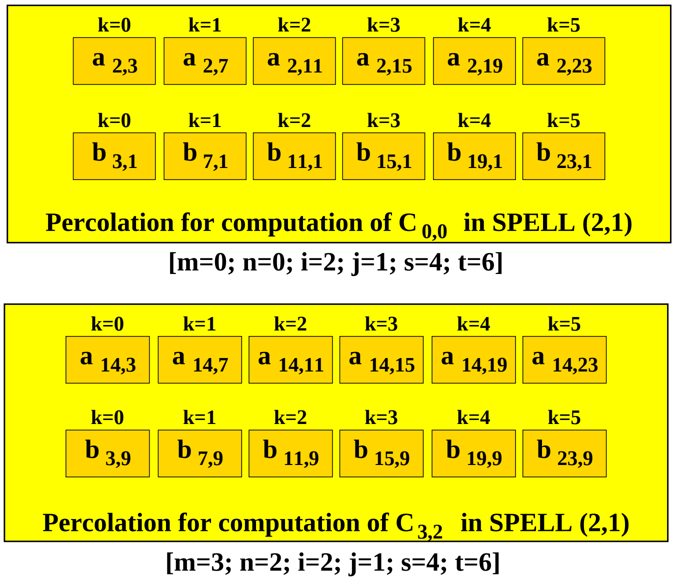

Figure 6: Sequences of sub-blocks of $\mathbf{A}$ and $\mathbf{B}$ that percolate to SPELL $(2,1)$. On the top are the sub-blocks required to compute the block $\mathbf{C}_{0,0}$. On the bottom the sub-blocks to compute the block $\mathbf{C}_{3,2}$.

4.2 Percolation from SRAM to CRAM

Having established the blocking algorithm for the computation in the SPELLs, we can determine which sub-blocks need to percolate through each SRAM. For simplicity, first we ignore the data interleaving in the SRAM and specify the algorithm as if there was a single SRAM PIM for each SPELL. Because we assumed that $M = t \times b_s$, each SRAM PIM has to percolate t sub-blocks of $\mathbf{A}$ and $\mathbf{B}$ to its SPELL before a sub-block of $\mathbf{C}$ is ready to be percolated outward to the SRAM.

Figure 6 illustrates, for an architecture with 16 SPELLs and bundles formed by 36 sub-blocks ($s = 4$ and $t = 6$), the sub-blocks of $\mathbf{A}$ and $\mathbf{B}$ that percolate to the SPELL $(2,1)$ in order to compute the block $\mathbf{C}_{0,0}$, and to compute the block $\mathbf{C}_{3,2}$. In general, the sub-blocks $\mathbf{a}_{\alpha,\kappa}$ and $\mathbf{b}_{\beta,\gamma}$ have to percolate to SPELL (i,j) to compute the multiplication of $\mathbf{A}_{m,k}$ by $\mathbf{B}_{k,n}$, where

$$\left\{\begin{array}{lcl} \alpha & = & i + m \times s \\ \kappa & = & (i \oplus j) + k \times s \\ \beta & = & (i \oplus j) + k \times s \\ \gamma & = & j + n \times s \end{array}\right. \tag{3}$$

and $\oplus$ is the addition module s.

To obtain all sub-blocks that have to percolate from the SRAM of SPELL (i,j), we make $m = 0, 1, \ldots, s-1$; $n = 0, 1, \ldots, s-1$; and $k = 0, 1, \ldots, t-1$ in equations 3. For instance, Figure 7 shows the bundle of sub-blocks that the SRAM of the SPELL $(2,1)$ must be able to percolate.

$a_{2,3}$	$a_{2,7}$	$a_{2,11}$	$a_{2,15}$	$a_{2,19}$	$a_{2,23}$
$a_{6,3}$	$a_{6,7}$	$a_{6,11}$	$a_{6,15}$	$a_{6,19}$	$a_{6,23}$
$a_{10,3}$	$a_{10,7}$	$a_{10,11}$	$a_{10,15}$	$a_{10,19}$	$a_{10,23}$
$a_{14,3}$	$a_{14,7}$	$a_{14,11}$	$a_{14,15}$	$a_{14,19}$	$a_{14,23}$
$a_{18,3}$	$a_{18,7}$	$a_{18,11}$	$a_{18,15}$	$a_{18,19}$	$a_{18,23}$
$a_{22,3}$	$a_{22,7}$	$a_{22,11}$	$a_{22,15}$	$a_{22,19}$	$a_{22,23}$

$b_{3,1}$	$b_{3,5}$	$b_{3,9}$	$b_{3,13}$	$b_{3,17}$	$b_{3,21}$
$b_{7,1}$	$b_{7,5}$	$b_{7,9}$	$b_{7,13}$	$b_{7,17}$	$b_{7,21}$
$b_{11,1}$	$b_{11,5}$	$b_{11,9}$	$b_{11,13}$	$b_{11,17}$	$b_{11,21}$
$b_{15,1}$	$b_{15,5}$	$b_{15,9}$	$b_{15,13}$	$b_{15,17}$	$b_{15,21}$
$b_{19,1}$	$b_{19,5}$	$b_{19,9}$	$b_{19,13}$	$b_{19,17}$	$b_{19,21}$
$b_{23,1}$	$b_{23,5}$	$b_{23,9}$	$b_{23,13}$	$b_{23,17}$	$b_{23,21}$

Figure 7: Bundle of sub-blocks that the SRAM corresponding to the SPELL in position $(2, 1)$ of the grid must be able to percolate, for $t = 6$ and $s = 4$.

$c_{2,1}$	$c_{2,5}$	$c_{2,9}$	$c_{2,13}$	$c_{2,17}$	$c_{2,21}$
$c_{6,1}$	$c_{6,5}$	$c_{6,9}$	$c_{6,13}$	$c_{6,17}$	$c_{6,21}$
$c_{10,1}$	$c_{10,5}$	$c_{10,9}$	$c_{10,13}$	$c_{10,17}$	$c_{10,21}$
$c_{14,1}$	$c_{14,5}$	$c_{14,9}$	$c_{14,13}$	$c_{14,17}$	$c_{14,21}$
$c_{18,1}$	$c_{18,5}$	$c_{18,9}$	$c_{18,13}$	$c_{18,17}$	$c_{18,21}$
$c_{22,1}$	$c_{22,5}$	$c_{22,9}$	$c_{22,13}$	$c_{22,17}$	$c_{22,21}$

Figure 8: Bundle of sub-blocks of $\mathbf{C}$ that are percolated back to the SRAM corresponding to the SPELL $(2, 1)$, for $t = 6$ and $s = 4$.

The percolation of the sub-blocks of matrices $\mathbf{A}$ and $\mathbf{B}$ from SRAM to CRAM follows a standard matrix multiply algorithm. We assume that the blocks of $\mathbf{C}$ are computed in a row first order. Thus initially row 0 of $\mathbf{A}$ and column 0 of $\mathbf{B}$ are percolated by each SRAM PIM. The block $\mathbf{C}_{0,0}$ is percolated back to the SRAMs, and then the row 0 of $\mathbf{A}$ and the column 1 of $\mathbf{B}$ are percolated to the CRAMs for the computation of $\mathbf{C}_{0,1}$. Observe that the percolation of a row of $\mathbf{A}$ requires the individual percolation of t sub-blocks of $\mathbf{A}$. Likewise the percolation of a column of $\mathbf{B}$ implies the percolation of t sub-blocks of $\mathbf{B}$.

Figure 8 shows the bundle of sub-blocks of $\mathbf{C}$ that are stored in the SRAM corresponding to SPELL $(2, 1)$.

4.3 Percolation from DRAM into SRAM

In section 4.2 we determined the *desired layout* for the bundle of sub-blocks matrices $\mathbf{A}$ and $\mathbf{B}$ in SRAM. We also established the layout of the bundle of sub-blocks of the matrix $\mathbf{C}$ that will be produced in each SRAM. We now have to specify the data transformations that are required in the DRAM level to produce the desired data layout, and to properly store the results in HRAM.

The data transformation performed in DRAM stores the sub-blocks of $\mathbf{A}$ in row first order in vectors. These sub-block vectors are concatenated together to form a longer vector containing the bundle of sub-blocks to be percolated to the SRAMs. The sub-blocks of $\mathbf{B}$ are transformed and stored in column first order in vector form and are also concatenated to form the longer vector containing the bundle of sub-blocks to be percolated to the SRAMs. The storage of the sub-blocks of $\mathbf{A}$ in row first order and of the sub-blocks of $\mathbf{B}$ in column first order increases the data locality during the computation in the SPELLs.

After the data transformation, the bundle of sub-blocks illustrated in Figure 7 is stored as a vector in DRAM. Thus the percolation from DRAM to SRAM is performed by a single data transfer. The data layout created by the data transformation in DRAM allows the percolation of a sub-block from SRAM to CRAM to be specified by a sub-block address and its size. There is no need for further transformations in the data layout in the SRAM level.

Figure 9 illustrates the data transformation that each DRAM must perform in order to obtain the *desired data layout* to start the percolation process. If we consider that the matrix represented in Figure 9 is matrix $\mathbf{A}$, the data transformation is the one for SPELL $(2, 3)$. From equations 3, for $t = 6$ and $s = 4$, SPELL $2, 3$ must receive the sub-blocks $\mathbf{a}_{2,1}, \mathbf{a}_{2,5}, \mathbf{a}_{2,9}, \ldots, \mathbf{a}_{2,21}, \mathbf{a}_{6,1}, \mathbf{a}_{6,5}, \ldots$. On the bottom of Figure 9 we represent the vector that contains the bundle of sub-blocks to be percolated to the SRAM.

We assume that both matrices $\mathbf{A}$ and $\mathbf{B}$ are read into the DRAM from the HRAM in a row first order. Each DRAM only needs to copy to its local buffer one sub-block from each block of the matrix. Moreover, the position of this sub-block is the same in all blocks of the matrix.

Observe that it is not necessary for any DRAM to store the entire matrix at one time. For each row of blocks, the DRAM only has to read one row of sub-blocks from HRAM, as illustrated in Figure 9. This reading is performed in lines 5-8 of the pseudo algorithm presented in Figure 10. The for loop starting in line 9 in Figure 10 spans over a row of blocks. The data corresponding to each sub-block is copied into the SRAM buffer for $\mathbf{A}$ by the for loops in lines 10-12.

The BDataTransformation presented in Figure 11 is very similar, except for the replacement of i for ψ in line 8, and of ψ for j in line 12, and for the reversed order of the for

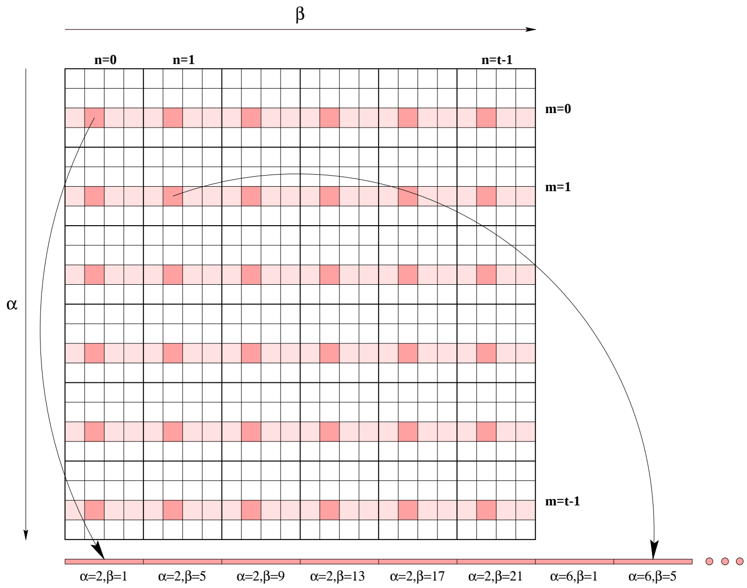

Figure 9: Data transformation performed in the DRAM corresponding to SPELL (2,3) in matrix **A** to obtain the desired data layout for percolation.

ADataTransformation(i, j, s, t, b_c, b_s)
1. $a \leftarrow 0$
2. $\psi \leftarrow i + j$
3. **if** $\psi \geq s$
4.   **then** $\psi \leftarrow \psi - s$
5. **for** $n \leftarrow 0$ **to** $t - 1$
6.   **do for** $x \leftarrow 0$ **to** $b_c - 1$
7.   **do for** $y \leftarrow 0$ **to** $(t \times b_s) - 1$
8.   **do** temp$[x][y] \leftarrow \mathbf{A}[n \times b_s + i \times b_c + x][y]$
9.   **for** $m \leftarrow 0$ **to** $t - 1$
10.   **do for** $f \leftarrow 0$ **to** $b_c - 1$
11.   **do for** $g \leftarrow 0$ **to** $b_c - 1$
12.   Abuffer$[a] \leftarrow$ temp$[f][m \times b_s + \psi \times b_c + g]$
13. $a \leftarrow a + 1$

Figure 10: Algorithm for the Data Transformation of Matrix **A** in DRAM.

loops in lines 10, and 11 that cause the sub-blocks of **B** to be stored in a column first order in the Bbuffer. We are still copying sub-blocks in linear arrays and organizing this sub-blocks in a row first fashion in the DRAM buffer for **B**.

The transformation of the vector containing a bundle of sub-blocks of **C** in DRAM is presented in the algorithm CDataTransformation in Figure 12. This data transformation is simpler than the ones for matrices **A** and **B** because there is no data skewing involved and there is no need to buffer the data in DRAM before writing it out to HRAM. In all three data transformation algorithms we are assuming that **A**[], **B**[], and **C**[] are addresses in HRAM.

```
BDataTransformation(i, j, s, t, b_c, b_s)
b ← 0
ψ ← i + j
if ψ ≥ s
  then ψ ← ψ − s
for n ← 0 to t − 1
    do for x ← 0 to b_c − 1
           do for y ← 0 to (t × b_s) − 1
                do temp[x][y] ← B[n × b_s + ψ × b_c + x][y]
       for m ← 0 to t − 1
           do for g ← 0 to b_c − 1
                  do for f ← 0 to b_c − 1
                         Bbuffer[b] ← temp[f][m × b_s + j × b_c + g]
                         b ← b + 1
```

Figure 11: Algorithm for the Data Transformation of Matrix **B** in the DRAM.

```
CDataTransformation(i, j, s, t, b_c, b_s)
c ← 0
for n ← 0 to t − 1
    do for m ← 0 to t − 1
           do for x ← 0 to b_c − 1
               do for y ← 0 to b_c − 1
                      do C[m b_s + i b_c + x][n b_s + j b_c + y] ← Cbuff[(m b_c + x) t b_c + n b_c + y]
```

Figure 12: Algorithm for the Data Transformation of the vector Cbuff percolated back to DRAM to produce matrix **C** in HRAM.

4.4 Communication and Computation Interleaving

Figure 13 summarizes the computation to produce one sub-block of **C** in a SPELL. We would like to allow significant overlap between the steps of the computation, in order to reduce the execution time. If we consider that the SPELL computation units work independently of the communication of data among the CRAMs, we can interleave the multiplication of two rows of sub-blocks of **A** by two columns of sub-blocks of **B** to produce a pipeline effect.

The multiplication of a sub-block in an odd row of **A** by a sub-block in an odd column of **B** is interleaved with the multiplication of a sub-block in an even row of **A** by a sub-block in an even column of **B**. To implement this interleaving mechanism we reserve space for three sub-blocks of **A** and three sub-blocks of **B** in each CRAM. While data is received in one sub-block, data is transmitted from a second one, and the data from the third sub-block is used in the current computation. Because we interleave sub-block multiplications that contribute to the result of the same **C** sub-block, only one such **C** sub-block is required in each CRAM. Therefore the data storage requirement in each CRAM is seven sub-blocks.

The interleaving is implemented by three threads in each SPELL. For instance, Figure 14 illustrates the synchronization signals and the data movements concerning Thread T_1, The

```
for k ← 0 to t − 1
    do Percolate sub-blocks of A and B inward to CRAM
        for r ← 0 to s − 1
            do Multiply sub-block of A by sub-block of B
                Shift sub-blocks to neighbors
Percolate sub-block of C outward to SRAM
```

Figure 13: Outline of the computation to generate one sub-block of **C** in a SPELL.

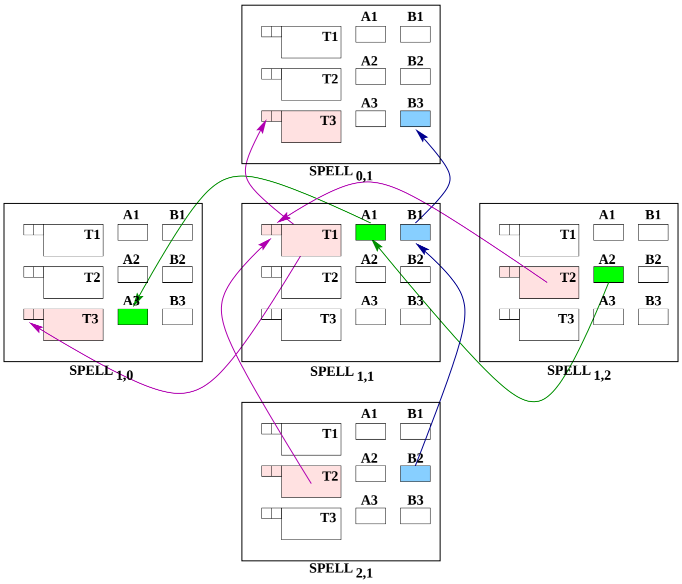

Figure 14: Data movements and exchange of synchronization signals among neighboring SPELLs in the grid.

pseudo-code for thread T_1 is presented in Figure 15. When the DATA_MOVE_SYNCs initiated by the thread T_2 of the bottom neighbor and by the thread T_2 of the right neighbor are completed, thread T_1 is enabled and uses the data just received in A_1 and B_1 to compute the sub-block multiplication. When the multiplication is completed, thread T_1 issues two DATA_MOVE_SYNCs to move the data from its A_1 buffer in the A_3 buffer of its left neighbor, and the data from its B_1 buffer in the B_3 buffer of its top neighbor. The thread T_3 of these neighbors receive a sync signal when these data transfers are completed. This mechanism allows for the sub-block multiplication to overlap with the data exchange among the SPELLs.

Observe that the pseudo-code in Figure 15 has two entry points to the thread, the label **THREAD_START_T1** is the beginning of the thread that is synchronized by the end of the percolation process from SRAM to CRAM. The second entry point is the label **THREAD_T1** in line 13 that is synchronized by the thread T_3 of the SPELL to the right and thread T_3 of the SPELL to the bottom of the current SPELL.

Table 1 displays, for each thread T_i, the data buffers that are used by each T_i to perform

```
THREAD_START_T1:
        left ← j − 1
        if left < 0
          left ← s − 1
        top ← i − 1
        if top < 0
          top ← s − 1
        for k ← 0 to s − 1
            do      C ← C + (A_1 × B_1)
                    BLOCK_MOVE_SYNC(A_1, (S_{i,left}, A_3), sync(S_{i,left}, T_3))
                    BLOCK_MOVE_SYNC(B_1, (S_{top,j}, B_3), sync(S_{top,j}, T_3))
                    END_THREAD()
                THREAD_T1:
```

Figure 15: Pseudo-code for thread T2 in a SPELL.

Thread	Uses Data From:	Receives Sync From:	Sends Data To:	Sends Sync To
$(S_{i,j};T_1)$	$(S_{i,j};A_1)$	$(S_{i\oplus 1,j};T_2)$	$(S_{i\ominus 1,j};A_3)$	$(S_{i\ominus 1,j};T_3)$
	$(S_{i,j};B_1)$	$(S_{i,j\oplus 1};T_2)$	$(S_{i,j\ominus 1};B_3)$	$(S_{i,j\ominus 1};T_3)$
$(S_{i,j};T_2)$	$(S_{i,j};A_2)$	$(S_{i\oplus 1,j};T_3)$	$(S_{i\ominus 1,j};A_1)$	$(S_{i\ominus 1,j};T_1)$
	$(S_{i,j};B_2)$	$(S_{i,j\oplus 1};T_3)$	$(S_{i,j\ominus 1};B_1)$	$(S_{i,j\ominus 1};T_1)$
$(S_{i,j};T_3)$	$(S_{i,j};A_3)$	$(S_{i\oplus 1,j};T_1)$	$(S_{i\ominus 1,j};A_2)$	$(S_{i\ominus 1,j};T_2)$
	$(S_{i,j};B_3)$	$(S_{i,j\oplus 1};T_1)$	$(S_{i,j\ominus 1};B_2)$	$(S_{i,j\ominus 1};T_2)$

Table 1: Communication pattern among threads in neighboring SPELLS. $(S_{i,j};T_k)$ is the Thread T_k in the SPELL (i,j). $(S_{i,j};B_k)$ is the buffer B_k of the same SPELL.

the computation, the buffers where T_i sends data is sent at the end of the computation, the threads that are synchronized at the end of the data movement, and the threads that send synchronization signals to start T_i. In the table, T_1, T_2, and T_3 are the three threads in each SPELL; $S_{i,j}$ is the SPELL (i,j) of the grid; and A_1, A_2, A_3, B_1, B_2, and B_3 are buffers in each CRAM to store sub-blocks of matrices **A** and **B**. Thus $(S_{i,j};T_2)$ is the thread T_2 of SPELL (i,j) of the grid.

4.5 Algorithm for Percolation from SRAM to CRAM

The algorithm for the percolation of sub-blocks of **A** from the SRAM to the CRAM is presented in pseudo-code in Figure 16. For **A** it is necessary to percolate each row of sub-blocks t times before moving to the next row. In the pseudo code, γ is the index spanning over the rows of sub-blocks, and θ is the index spanning over the columns of sub-blocks of **A**. The for loop in line 4 that uses σ as an index is necessary to ensure that each row is percolated t times before the percolation of the next row starts. rbase is the row base address and corresponds to the

ASramPercolation
1. **THREAD_START_A_PERCOLATION:**
2. **for** $\gamma \leftarrow 0$ **to** $t-1$
3. **do** $rbase \leftarrow \gamma \times t \times b_c \times b_c$
4. **for** $\sigma \leftarrow 0$ **to** $t-1$
5. **do** **for** $\theta \leftarrow 0$ **to** $t/2-1$
6. **do** **SC_DATA_DISPATCH** (Abuffer + rbase + $(\theta \times b_c \times b_c)$, $A1$, $sync$(THREAD_START_T1))
7. **SC_DATA_DISPATCH** (Abuffer + rbase + $((\theta+1) \times b_c \times b_c)$, $A2$, $sync$(THREAD_START_T2))
8. **END_THREAD()**
9. **THREAD_NEXT_SUBBLOCK:**
10. **END_THREAD()**
11. **THREAD_NEXT_ROW:**

Figure 16: Algorithm for the percolation of sub-blocks of matrix **A** from SRAM to CRAM.

address in the buffer Abuffer of the first sub-block in row γ. Notice that this code has three thread starting points: in line 1 the percolation process is started; to request the percolation of a new sub-block, CRAM synchronizes the slot corresponding to the thread in line 9; and to start the percolation of a new row, the thread in line 11 must receive a synchronization signal.

5 Algorithm Complexity

In section 4 we proposed the algorithms for the percolation and computation in all three regions: DRAM, SRAM, and CRAM. In this section we analyze the time and space complexity of these algorithms. In section 6 we will study the hardware parameters for HTMT to allow us to develop a performance prediction for the execution of the blocking dense matrix multiply algorithm in HTMT.

The size of the sub-blocks is determined by the amount of storage space available in CRAM. Assuming that we are interested in the multiplication of very large matrices, we will try to make the sub-blocks as large as possible to favor the computation/communication ratio. Each sub-block is formed by $(b_c)^2$ elements, b_c is computed by

$$b_c = \left\lfloor \sqrt{\frac{\text{CRAM size} - \text{CRAM system space}}{\text{num of subblocks per CRAM} \times \text{word size}}} \right\rfloor \tag{4}$$

where $\lfloor x \rfloor$ is the floor of x, i.e., the largest integer n such that $n \leq x$. In section 4.4 we established that the algorithm requires the simultaneous storage of 7 sub-blocks in each CRAM module. We assume a word size of 8 bytes. Each CRAM module has capacity to store 1 Mbytes. Assuming that 160 Kbytes are necessary to store the code to perform the RTS functions, and the user code, we obtain:

$$b_c = \left\lfloor \sqrt{\frac{(1024 - 160) \times 1024}{7 \times 8}} \right\rfloor = 125 \tag{5}$$

The number of sub-blocks per bundle is t^2; t is determined by the amount of memory available in the SRAM associated with each SPELL:

$$t = \left\lfloor \frac{1}{b_c} \sqrt{\frac{\text{SRAM size} - \text{SRAM system space}}{\text{num of bundles per SRAM}}} \right\rfloor \tag{6}$$

The algorithm described in Section 4.2 requires 3 bundles per SRAM. Assuming that the SRAM associated with each SPELL can store 32 Mwords [15] and 4MB = 0.5 Mwords are required for the runtime system, operating system and user code, we obtain

$$t = \left\lfloor \frac{1}{125} \sqrt{\frac{(32 - 0.5) \times (1024)^2}{3}} \right\rfloor = 26 \tag{7}$$

A sub-block multiplication in the SPELL requires that $2\,(b_c)^3$ floating point operations be performed. To produce one sub-block of the matrix **C**, it is necessary to percolate and multiply t sub-blocks of **A** by t sub-blocks of **B**. An important question in the HTMT architecture is whether it is possible to transmit enough data to the CRAMs to keep the SPELLs busy. In this algorithm, the ratio of floating point operations per byte transmitted is given by:

$$\frac{\text{FLOPs}}{\text{bytes percolated}} = \frac{2\,(b_c)^3}{2\,(b_c)^2 \times 8 + \frac{1}{t} \times (b_c)^2 \times 8} = \frac{1}{4}\,\frac{t\,b_c}{(2\,t+1)} = 15.3\frac{\text{FLOPs}}{\text{byte}} \tag{8}$$

Another important question for the interface between the SRAM and the CRAM, is the ratio between bytes transmitted into the CRAM, and bytes transmitted out of the CRAM. Considering only the data traffic, this ratio is given by:

$$\frac{\text{bytes\_in}}{\text{bytes\_out}} = \frac{2\,t\,(b_c)^2 \times 8}{(b_c)^2 \times 8} = 2t = 52 \tag{9}$$

The time complexity of the data transformations performed in DRAM is determined by the number of words accessed for each transformation. For the transformations for matrices **A** and **B** the algorithms in Figure 10 and 11 perform $t^2 \times b_c \times b_s + t^2 \times (b_c)^2 = t^2 \times b_c^2 \times (s+1)$ word copies. The algorithm for the data transformation of **C** presented in Figure 12 performs $t^2 \times (b_c)^2$ word copies. Remember that a word copy requires two word accesses: a read and a write access.

Table 2 summarizes the complexity of each one of the algorithm phases. The complexity listed in Table 2 is for a single execution of the phase. To obtain the portion of time spent in each phase we also need to take into consideration the hardware speed, the number of times that each phase has to be performed, and the overlapping among phases.

Algorithm Phase	Complexity	Measuring Unit
A and **B** DRAM transformation	$2\,t^2\,(b_c)^2\,(s+1)$	word accesses
C DRAM transformation	$2\,t^2\,(b_c)^2$	word accesses
DRAM to SRAM percolation	$t^2\,(b_c)^2$	word transfers
SRAM to CRAM percolation	$(b_c)^2$	word transfers
Sub-block Multiplication in SPELL	$2\,(b_c)^3$	FLOPS
Sub-block Shift in CRAM	$(b_c)^2$	word transfers per SPELL pair

Table 2: Complexity in terms of amount of information exchanged or operations performed for each phase of the computation.

Hardware Parameter	Symbol	Value
Storage Space per SRAM		128 Kwords
Storage Space per CRAM (aggregate)		32 Mwords
Storage Space per DRAM (aggregate)		512 Mwords
Average access time in DRAM	DRAM_{at}	0.97 ns/word
Average access time in SRAM	SRAM_{at}	0.42 ns/word
Average access time in CRAM	CRAM_{at}	30 ps/word
Number of FPU/SPELL		5
SPELL FPU Average Cycle	SF_t	15 ps
Bandwidth for each VORTEX port	DV_{bw}	10 Gwords/s
Bandwidth between each CRAM and each SRAM	CS_{bw}	64 Gwords/s
CNET bandwidth (peer-to-peer)		16.7 Gwords/s

Table 3: Hardware parameters assumed for the design point for the year 2006/2007

6 Hardware Parameters

To estimate how long a sub-block multiplication in a SPELL takes, we have to multiply the number of floating point operations given in Table 2 by the average period of a SPELL floating point unit and divide by the number of units available in the SPELL:

$$M_S \quad = \quad 2\,(b_c)^3 \times \frac{SF_t}{\#FPU/SPELL} \tag{10}$$

where M_S is the average time required for one sub-block multiplication, and SF_t is the average cycle of the SPELL FPU. A SPELL processor has 5 floating-point functional units (FPUs) operating with an average cycle time, SF_t, of about 15 ps [8] [4]. These FPUs are connected to the remainder of the processor through the intra-processor network (PNET). This estimation

[4]In [8] Dorojevets et al. write that the average cycle time for the FPUs is 30 ps. However our personal correspondence with the authors revealed that this design parameter has been revised since and that the current estimate is 15 ps per cycle.

assumes that there is no contention for registers and other conflicts that could slow down the computation. Therefore we obtain:

$$M_S \quad = \quad 2\,(125)^3 \times \frac{15\,ps}{5} = 11.7\ \mu s \tag{11}$$

We also want to estimate the time necessary to shift the sub-blocks among the SPELLs. For a first approximation we assume that the credit-based control flow is used to build the CNET. Thus the sustained rate across a micro-pipeline can be up to 8.2 Gbits/second [20]. The aggregation of these micro-pipelines provides an aggregated rate for peer-to-peer peak bandwidth of 8 bytes/30 ps [8]. With the CNET loaded, this bandwidth reduces to 4 bytes/30ps = 16.7 Gwords/s. Each one of the sub-blocks contains $(b_c)^2$ words. Thus the delay to shift a sub-block from a SPELL to another, D_S, is

$$D_S = (b_c)^2 \times \frac{1}{\text{CNET peer to peer Bandwidth}} = (125)^2 \times \frac{s}{16.7 \times 10^9 \text{words}} = 0.94\ \mu s \tag{12}$$

For the 2007 design point, it is predicted that the SRAM will operate at a clock speed of 900 MHz, and that each access will require 3 machine cycles [15]. We assume that 8 words are read/written in each access. The access time in the SRAM is given by

$$\text{SRAM}_{at} \quad = \quad \frac{\text{CPA}}{\text{WPA}} \times \frac{1}{\text{clock rate}} = \frac{3}{8} \times \frac{1}{900 \times 10^6} = \frac{0.42\ ns}{\text{word}} \tag{13}$$

$$\tag{14}$$

where CPA is the average number of machine cycles per access, and WPA is the average number of words accessed in each access.

Similarly, for the DRAM, the projected clock speed is 450 MHz, each access requires 7 machine cycles, and we assume that 16 words are read/written in each access [15]. Thus, the average access time in DRAM is computed by:

$$\text{DRAM}_{at} \quad = \quad \frac{\text{CPA}}{\text{WPA}} \times \frac{1}{\text{clock rate}} = \frac{7}{16} \times \frac{1}{450 \times 10^6} = \frac{0.97\ ns}{\text{word}} \tag{15}$$

7 Performance Estimation

The computation of the matrix **C** requires: (1) a data transformation for **A** and a data transformation for **B** in DRAM; (2) the inward percolation of a bundle of sub-blocks of **A** and a bundle of sub-blocks of **B** from each DRAM to each SRAM; (3) the computation of t^2 blocks of **C**; (4) the outward percolation of a bundle of sub-blocks of matrix **C** from each SRAM to each DRAM; and (5) the data transformation of **C** in DRAM. This sum of times is represented in the following equation:

$$T \quad = \quad D_t[A] + D_t[B] + 2\,IP_{DS} + t^2\,T_B + OP_{SD} + D_t[C] \tag{16}$$

where T is the time to compute the multiplication of two matrices of dimension $M \times M$, with $M = t \times s \times b_c$; $D_t[A]$ and $D_t[B]$ are the time complexity of the data transformation of matrices

A and **B** in DRAM; IP_{DS} is the time required for the inward percolation of one bundle of sub-blocks from DRAM to SRAM; t^2 is the number of blocks in each matrix, T_B is the time required to compute one block of **C** and store the result in SRAM; and OP_{SD} is the time required for the outward percolation of one bundle of sub-blocks from SRAM to DRAM.

The computation of one block of C requires t block multiplications(see equation 1 in section 4. Therefore the time required to compute one block of **C**, T_B, can be expressed by the following equation:

$$T_B \quad = t\,[2\,IP_{SC} + s \max(M_S, D_S)] + OP_{CS} \tag{17}$$

where IP_{SC} is the time required for the inward percolation of one sub-block from SRAM to CRAM; M_S is the time necessary to perform one sub-block multiplication in a SPELL; D_S is the time required to shift the data in the SPELL grid; and OP_{CS} is the time required for the outward percolation of a sub-block of **C** from CRAM to SRAM. Notice that the multiplication of a block of **A** by a block of **B** is performed in parallel by s^2 SPELLs. Thus each SPELL performs s sub-block multiplications. Because we allow the overlap of the sub-block multiplication with the shifting of sub-blocks among the SPELLs, the computation of T_B only needs to include the maximum between M_S and D_S. The computation of an entire block of **C** only requires the outward percolation of a single sub-block of **C** because each one of the s^2 SPELLs will percolate one of these sub-blocks.

If we assume that $M_S > D_S$, $D_t[A] = D_t[B]$, $IP_{DS} = OP_{SD}$, and $IP_{SC} = OP_{CS}$, we can rewrite equations 16 and 17 as

$$T \quad = \quad 2\,D_t[A] + 3\,IP_{DS} + t^2\,T_B + D_t[C] \tag{18}$$

$$T_B \quad = \quad t\,s\,M_S + (2t+1)\,IP_{SC} \tag{19}$$

replacing equation 19 in equation 18 we obtain

$$T \quad = \quad 2\,D_t[A] + D_t[C] + 3\,IP_{DS} + t^3\,s\,M_S + t^2\,(2t+1)\,IP_{SC} \tag{20}$$

We now want to estimate each one of the components of the execution time. Table 2 has the expression for the number of word accesses required for the transformation of matrices **A**, **B** and **C** in DRAM. The data transformation in DRAM is performed in parallel by s^2 DPIMs. According to the algorithms shown in Figures 10 and 11, and the illustration in Figure 9 for the data transformation of matrices **A** and **B**, each DPIM needs to transform t^2 sub-blocks and each sub-block has $(b_c)^2$ elements, resulting in $t^2\,(b_c)^2$ word copies. However, for each row of blocks, the DPIM has to copy an entire row of sub-blocks into a buffer in order to perform the data transformation. A row of sub-blocks has $t\,s\,(b_c)^2$ elements, and the DPIM will handle t rows. Therefore the complexity of this first step (lines 5-8 in the pseudo code of Figure 11) is $t^2\,s\,(b_c)^2$ word copies. Each word copy requires two DRAM accesses, one to read the word and another to write the word. Thus the time required for a data transformation is given by:

$$D_t[A] \quad = \quad D_t[B] = 2\,t^2\,b_c^2\,(s+1)\,\mathrm{DRAM}_{at} \tag{21}$$

$$\quad = \quad 2\,(26)^2\,(125)^2 \times 65 \times 0.97\,ns = 1.33\,s \tag{22}$$

On the other hand, the data transformation for **C** only requires the scattering of t^2 sub-blocks. Thus its time complexity is given by:

$$
\begin{aligned}
D_t[C] &= 2\, t^2\, (b_c)^2\, \mathrm{DRAM}_{at} && (23)\\
&= 2\,(26)^2 \times (125)^2 \times 0.97\ ns = 20.5\ ms && (24)
\end{aligned}
$$

The amount of time that it takes to percolate each sub-block from SRAM to CRAM is given by the access time for SRAM and CRAM and by the time required to send a block of data between them. Assuming that, using Dally's technology, the data ratio for communication between SRAM and CRAM is 4 Tbits/sec = 64 Gwords/sec [15]. With $\mathrm{SRAM}_{at} = 0.42\ ns/\mathrm{word}$ we can access 2.4 Gwords/sec in SRAM. The CRAM has a cycle of 30 ps and thus can access 33 Gwords/sec [8] Both the inward percolation of a sub-block from SRAM to CRAM, IP_{SC}, and the outward percolation of a sub-block from CRAM to SRAM, OP_{CS}, are given by:

$$
\begin{aligned}
IP_{SC} &= \max(\text{SRAM Access Time}\ ,\ \text{Fly Time}\ ,\ \text{CRAM Access Time}) && (25)\\
IP_{SC} &= \text{subblock size} \times \max\left(\frac{1}{2.4}, \frac{1}{64}, \frac{1}{33.3}\right)\ \frac{s}{\text{Gwords}}\\
IP_{SC} &= (125)^2 \times \frac{1}{2.4}\ \frac{s}{\text{Gwords}} = 6.5\mu s
\end{aligned}
$$

(26)

In general, if the access time in the SRAM is the limiting factor for the percolation between the SRAM and the CRAM, IP_{SC} can be expressed as:

$$IP_{SC} = (b_c)^2 \times SRAM_{at} \tag{27}$$

Next we want to estimate the time to percolate a bundle of sub-blocks from DRAM, through the data vortex, into SRAM. This time is given by:

$$IP_{DS} = \max(\text{DRAM Access Time}\ ,\ \text{Data Vortex Fly Time}\ ,\ \text{SRAM Acess Time}) \tag{28}$$

We established in Section 6 that the DRAM access time is $0.97 ns/\mathrm{word}$ or 1.03 Gwords/sec, and that the SRAM bandwidth is 2.4 Gwords/sec. Each input (and output) port of the data vortex can deliver 640 Gbits/sec= 10 Gwords/sec [4], with a total latency turn-around of 107 ns [3]. A bundle of sub-blocks is formed by t^2 sub-blocks. Thus the percolation time from DRAM to SRAM is given by:

$$
\begin{aligned}
IP_{DS} &= t^2\, (b_c)^2\ \max\left(\frac{1}{1.03}, \frac{1}{10}, \frac{1}{2.4}\right)\ \frac{s}{\text{Gwords}}\\
&= (26)^2 \times (125)^2 \times \frac{1}{1.03 \times 10^9} = 10.3\ ms && (29)
\end{aligned}
$$

In general, if the DRAM access time is the limiting factor in the transfer of data from DRAM to SRAM, $IP_D S$ is computed by:

$$IP_{DS} = t^2\, (b_c)^2\, \mathrm{DRAM}_{at} \tag{30}$$

Component of Execution Time	Estimation	Symbol
A DRAM data transformation	$1.33s$	$D_t[A]$
B DRAM data transformation	$1.33s$	$D_t[B]$
C DRAM data transformation	$20.5ms$	$D_t[C]$
Inward Percolation of block from DRAM to SRAM	$10.3\ ms$	IP_{DS}
Inward Percolation of sub-block from SRAM to CRAM	$6.6\ \mu s$	IP_{SC}
Sub-block multiplication	$11.7\ \mu s$	M_S
Sub-block exchange among SPELLs	$0.94\ \mu s$	D_S
Outward Percolation of **C** sub-block from CRAM to SRAM	$6.6\ \mu s$	OP_{CS}
Outward Percolation of **C** block from SRAM to DRAM	$10.3\ ms$	OP_{SD}

Table 4: Principal components of the execution time for the matrix multiply percolation algorithm.

The components of the total execution time are summarized in Table 4. Replacing equations 21, 23, 30, 10, and 27 in equation 20, and performing some algebraic manipulations, we obtain:

$$T = t^2\ (b_c)^2\ \left[(4\,s + 9)\ \mathrm{DRAM}_{at} + 2\ t\ s\ b_c\ \frac{SF_t}{F_S} + (2t+1)\mathrm{SRAM}_{at}\right] \tag{31}$$

where F_S is the number of floating point units per SPELL. Equation 31 expresses the total execution time as a function of the DRAM average access time, the SRAM average access time and the average time to perform a floating point operation in a SPELL. For $b_c = 125$, $t = 26$, $s = 64$, and 5 FPUs per SPELL, we obtain:

$$T = 10.6 \times 10^6\ (265\ \mathrm{DRAM}_{at} + 83200\ SF_t + 53\ \mathrm{SRAM}_{at}) \tag{32}$$

Substituting the values from Table 3 in equation 32, we find that the estimated execution time for the matrix multiply algorithm, including percolation required 16.2 seconds. The algorithm performed $2\ (t\ s\ b_c)^3 = 2\ (26 \times 64 \times 125)^3 = 2\ (208000)^3 = 18 \times 10^{15}$ floating-point operations. Thus for the current hardware estimates we have a performance prediction of 1.1 petaFLOPS.

However, most of our estimations were quite optimistic. For instance it is unlikely that we will be able to keep all the floating point unit performing useful work all the time in the SPELLs. Thus we want to express the execution time in terms of the parameters that dominate its variation.

$$T = (2.8\ \mathrm{DRAM}_{at} + 881.92\ SF_t + 0.56\ \mathrm{SRAM}_{at}) \times 10^9 \tag{33}$$

$DRAM_{at}$ and SF_t are the dominant components of the execution time. Assuming that all other parameters remain constant, we can express the performance, measured in petaFLOPS, as

$$P\ [petaFLOPS] \quad = \quad \frac{\#\text{ of FLOP}}{T\ [s] \times 10^{15}} = \frac{18}{2.8\ \mathrm{DRAM}_{at} + 0.88\ SF_t + 0.24} \tag{34}$$

where the average DRAM access time, DRAM_{at}, must be expressed in ns/word, and the average SPELL floating-point execution cycle must be expressed in ps/FLOP.

The graph in Figure 17 plots the performance for the blocking matrix multiply percolation algorithm expressed in petaFLOPS as a function of the average execution time in the SPELL floating-point units, expressed in ps/FLOP. Each curve represent a different value for the average access time in the DRAM, expressed in ns/word. The assumptions used in the calculation presented earlier are of 0.97 ns/word for the DRAM access, and 15 ps/FLOP for the SPELL FPUs, resulting in the 1.1 petaFLOPS performance.

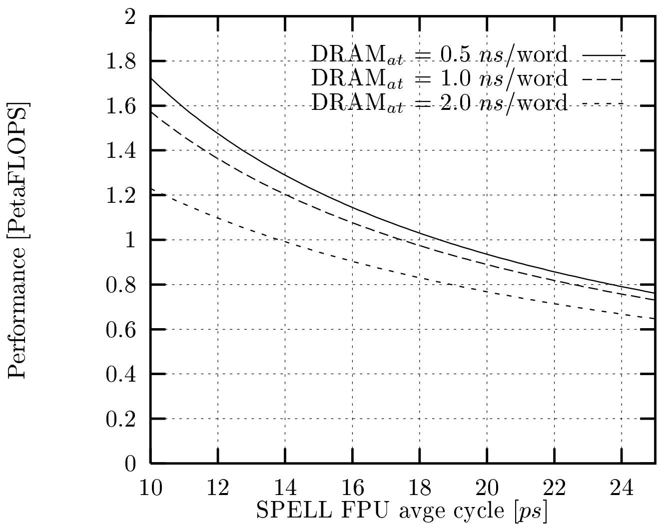

Figure 17: Performance, measured in petaFLOPS, as function of the SPELL Floating Point Unit cycle, SF_t, and of the DRAM PIM access time DRAM_{at}.

The estimation for the execution time in equation 20, and for performance in equation 34 assumes that the task at hand is the computation of the multiplication of matrices of size $t\ s\ b_c \times t\ s\ b_c = 208\ K \times 208\ K$ elements. If the task is to compute the multiplication of much larger matrices, and if there is enough storage space in DRAM to store a second block of each matrix, the data transformation in DRAM can overlap with the SRAM to CRAM percolation and with the sub-block multiplication in the SPELL. For instance, if we are multiplying matrices of dimension $q\ t\ s\ b_c \times q\ t\ s\ b_c$, equation 20 can be rewritten as:

$$T = 2\ D_t[A] + D_t[C] + q^3\ \{3\ IP_{DS} + t^3\ s\ M_S + t^2\ (2t+1)\ IP_{SC}\} \tag{35}$$

For this new dimension for the matrix, we perform $2\ (q\,t\,s\,b_c)^3$ operations. Therefore the performance is now given by:

$$P\ [petaFLOPS] = \frac{\#\text{ of FLOP}}{T\ [s] \times 10^{15}}$$

$$P\ [petaFLOPS] = \frac{2\ (q\,t\,s\,b_c)^3 \times 10^{-15}}{2\ D_t[A] + D_t[C] + q^3\ \{3\ IP_{DS} + t^3\ s\ M_S + t^2\ (2t+1)\ IP_{SC}\}}$$

$$P\ [petaFLOPS] \quad = \quad \frac{2\ (t\, s\, b_c)^3 \times 10^{-15}}{\frac{1}{q^3}\{2\ D_t[A] + D_t[C]\} + 3\ IP_{DS} + t^3\ s\ M_S + t^2\ (2t+1)\ IP_{SC}} \tag{36}$$

Replacing equations 21, 23, 30, 27, and 10 in equation 36, we obtain:

$$P\ [petaFLOPS] \quad = \quad \frac{2\, t\, b_c\, s^3 \times 10^{-15}}{\{\frac{1}{q^3}[4s+6] + 3\}\mathrm{DRAM}_{at} + 2\, t\, s\, b_c\, \frac{SF_t}{F_S} + (2t+1)\mathrm{SRAM}_{at}} \tag{37}$$

Using $t = 26$, $b_c = 125$, $s = 64$, and $F_S = 5$, i.e., 5 floating point units per SPEEL, in equation 37 we obtain:

$$P\ [petaFLOPS] \quad = \quad \frac{1.7 \times 10^{-6}}{\{\frac{1}{q^3}\, 0.262 + 0.003\}\mathrm{DRAM}_{at} + 0.832\, SF_t + 0.053\, \mathrm{SRAM}_{at}} \tag{38}$$

where DRAM_{at} and SRAM_{at} should be expressed in ns/word, and SF_t should be expressed in ps/FLOP.

Figure 18 plots the performance expressed in petaFLOPS as a function of the average floating point unit cycle in the SPELL for an average DRAM access time of 1 ns/word, for $q = 10$. Now that the data transformation in DRAM is overlapped with computation in the SPELLs, the average DRAM access time is not a critical parameter for performance. Observe that with this overlapping of floating point computation and data transformation, the 1 PetaFLOPS performance can be achieved even with an average FPU cycle of 20 ps in the SPELLs.

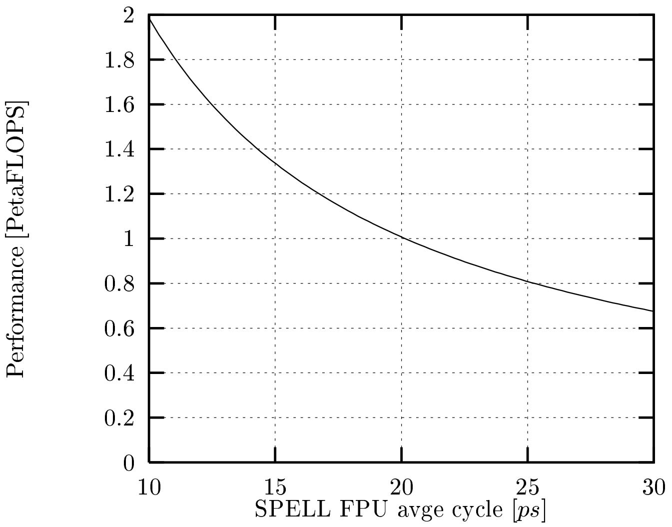

Figure 18: Performance, measured in petaFLOPS, as function of the SPELL Floating Point Unit cycle, SF_t, for an average DRAM PIM access time of 1 ns/word, and for $q = 10$.

8 Future Work

The analytical model used for the performance prediction in this document is greatly simplified. For instance we do not take into consideration the effect of long latencies for the memory accesses in the overall performance of the algorithms. Also we neither investigate how many floating point operations can be actually dispatched to the SPELL FPUs, nor we take into consideration any contention in the various networks in the machine.

There is much work to be done in improving the predictions that we stated here. We foresee two main lines of investigations: (1) generating actual code and running the code in an emulator to obtain a more refined model upon which to base our performance predictions; (2) to study other applications that require the design of algorithms in which the percolation between SRAM and DRAM also needs to be crafted to improve the re-use of data percolated to the SRAM.

Acknowlegments

We are thankful to Mikhail Dorojevets for the useful discussions and prompt reply to our requests for information about the CNET and SPELLs. This analitycal study of performance was made possible by numerous discussions with the HTMT applications and technology experts, including Mark Arend, Keren Bergman, Jay Brockman, Loring Craymer, Mark Herald, Ivan Judson, Peter Kogge, Konstantin Likharev, John Lou, Vince Freeh, Charles Norton, Coke Reed, John Salmon, Herb Siegel, Burton Smith, Rick Stevens, and Dmitri Zinoviev. Members of the HTMT Delaware team that participated in the discussions leading to this study include James Durbano, Thomas Geiger, Gerd Heber, Andres Marquez, Christopher Morrone, Kevin Theobald, Ruppa Thulasiram.

This work was performed for the Jet Propulsion Laboratory, California Institute of Technology, and was sponsored by the Defense Advanced Research Projects Agency (DARPA) and the National Security Agency (NSA) through an agreement with the National Aeronautics and Space Administration.

References

[1] CAPSL technical memo, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware. In ftp://ftp.capsl.udel.edu/pub/doc/memos.

[2] J. N. Amaral, Z. Ruiz, S. Ryan, G. Gao, and K. Theobald. Overview of HTMT-C. Technical report, University of Delaware, 1999. in preparation.

[3] M. Arend, K. Bergman, and C. Reed. Data vortex network packaging. Technical Report HTMT Tech. Note 040, Princeton University, Princeton, NJ, September 1998.

[4] M. Arend, C. Reed, and K. Bergman. Physical design and specifications for the data vortex network. Technical Report HTMT Tech. Note 033, Princeton University, Princeton, NJ, 1998.

[5] K. Bergman and C. Reed. Hybrid technology multithreaded architecture program design and development of the data vortex network. Technical report, Princeton University, 1998. Technical Note 2.0.

[6] L. E. Cannon. *A cellular computer to implement the Kalman filter algorithm*. PhD thesis, Montana State University, Bozeman, MT, 1969.

[7] M. Dorojevets, P. Bunyk, D. Zinoviev, and K. Likharev. Petaflops RSFQ system design. In *Applied Superconductivity Conference*, Palm Desert, Ca., Sept. 1998.

[8] M. Dorojevets, P. Bunyk, D. Zinoviev, and K. Likharev. Petaflops RSFQ system design. In *Applied Superconductivity Conference*, Palm Desert, CA, September 1998. http://gamayun.physics.sunysb.edu/RSFQ/Research/PetaFLOPs.

[9] G. R. Gao, J. N. Amaral, A. Marquez, and K. Theobald. A refinement of the HTMT program execution model. Technical Report TM22, University of Delaware, Newark, DE, July 1998. CAPSL Technical Memo.

[10] G. R. Gao, K. B. Theobald, A. Marquez, and T. Sterling. The HTMT program execution model. CAPSL Technical Memo 09, University of Delaware, Newark, Delaware, jul 1997. In ftp://ftp.capsl.udel.edu/pub/doc/memos.

[11] HTMT. Hybrid technology multi-threaded architectures. http://htmt.caltech.edu, 1998.

[12] Herbert H. J. Hum, Olivier Maquelin, Kevin B. Theobald, Xinmin Tian, Guang R. Gao, and Laurie J. Hendren. A study of the EARTH-MANNA multithreaded system. *International Journal of Parallel Programming*, 24(4):319–347, August 1996.

[13] P. M. Kogge, J. B. Brockman, T. Sterling, and G. Gao. Processing-in-memory: Chips to petaflops. Technical report, International Symposium on Computer Architecture, Denver, Co., June 1997.

[14] H.-J. Lee, J. P. Robertson, and J. A. B. Fortes. Generalized Cannon's algorithm for parallel matrix multiplication. In *International Conference on Supercomputing*, pages 44–51, Viena, Austria, July 1997.

[15] PIM Development Group. PIM technology projections for the HTMT project. Technical report, University of Notre Dame, January 1999.

[16] D. Psaltis and G. W. Burr. Holographic data storage. *Computer*, 31(2):52–60, Febuary 1998.

[17] S. Ryan, J. N. Amaral, G. Gao, Z. Ruiz, A. Marques, and K. Theobald. Coping with very high latencies in petaflop computer systems. Submited to International Symposium on High Performance Computing 99, Kyoto, Japan, 1998.

[18] Kevin B. Theobald, José Nelson Amaral, Gerd Heber, Olivier Maquelin, Xinan Tang, and Guang R. Gao. Overview of the Threaded-C language. CAPSL Technical Memo 19, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, March 1998. In ftp://ftp.capsl.udel.edu/pub/doc/memos.

[19] L. Wittie, D. Zinoviev, G. Sazaklis, and K. Likharev. CNET: Design of an RSFQ switching network for petaflops-scale computing. *IEEE Trans. on Appl. Supercond.*, June 1999. In press.

[20] S. Yorozu and D. Zinoviev. Design and implementation of an RSFQ switching node for petaflops networks. *IEEE Trans. on Appl. Supercond.*, June 1999. In press.