Performance Prediction for the HTMT: A Programming Example

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

A case study, dense matrix multiply, is used to introduce an analytical methodology to predict the performance of the percolation model on the Hybrid Technology Multi-Threaded (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.

1 Introduction

This paper presents an analytical performance prediction for the implementation of Cannon's matrix multiply algorithm in the Hybrid Technology Multi-Threading (HTMT) architecture [8]. The HTMT subsystems are built from new technologies: super-conducting processor elements (called *SPELLs* [5]), a network based on RSFQ (Rapid Single Flux Quantum) logic devices (called *CNET* [16, 17]), "Processor In Memory" (PIM) technology [10], a high-performance optical packet switched network (called *Data Vortex* [3], optical holographic storage devices (called HRAM [13]), and fine grain multi-threaded computing technology [9].

*Amaral, Gao, Ruiz and Ryan are with the Computer Architecture and Parallel Systems Laboratory, University of Delaware, Newark, DE, USA. http://*www.capsl.udel.edu*, emails: {*amaral,ggao,ryan,ruiz*}*@capsl.udel.edu*

†Merkey is with the 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*

‡Sterling is with the Center for Advanced Computer Research, California Institute of Technology, Pasadena, CA, USA. http://*www.cacr.caltech.edu/∼tron/*.

The development of the *percolation model* is dominated by two characteristics that distinguish HTMT: (1) the latency to fetch data or code from the next level in the memory hierarchy is in 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 transformations closer to the storage devices. This programming model is being developed to help the programmer exploit and tolerate these architectural characteristics [7, 6, 14]. Percolation establishes the following guiding principles: (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 into threads and ensures that a statement that requests a long latency operation is placed in a separate thread from the statements that depend on the results of the operation.

2 The HTMT Architecture

For a detailed description of the HTMT architecture we suggest that the reader review the Caltech HTMT web site and the links therein [8]. Figure 1 presents a simplified view of the HTMT architecture. The super-conducting processors (SPELLs), their associated cryostatic memory (CRAM), and the RSFQ based network (CNET) are in the center of the figure. We often refer to this center region as the cryostatic region because the super-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. This figure represents four SPELLs with the proportional

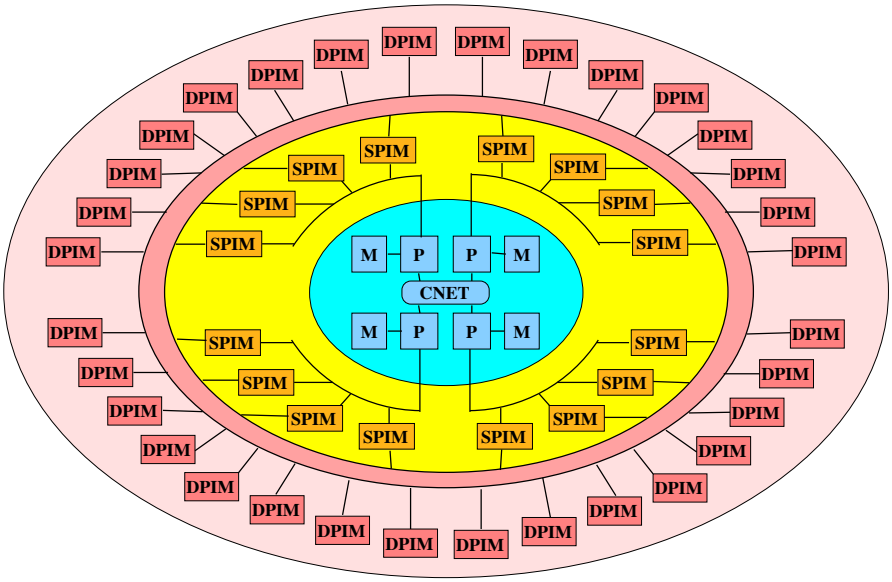

Figure 1. A simplified view of the processing and memory regions in the HTMT architecture.

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 communicates with any SPIM module through the data vortex, and that the DPIM and the SPIM modules 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. The latencies for communication between the DRAMs and the SRAMs and between the SRAMs and CRAMs are handled by the percolation process, while the latencies for communication across the CNET to access remote CRAM modules are mainly handled by EARTH-style two level threads with split phase synchronization operations [9].

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 capacity in the outside, 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 induce the idea 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 for more data or stores away results from previous computations. Depending on the application some of the events above might be not 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 [7, 15]. The HTMT program execution model is described in [6]. A complete specification of HTMT-C can be found in [2].

4 A Programming Example in HTMT-C

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

HTMT is a shared memory architecture, but its program model encourages explicitly controlling the movement of data between the memory levels to overcome the high latencies penalty otherwise incurred. A key strategy is to maximize the re-use of any data that has been moved into the core of the machine. This goes hand-in-hand with minimizing the amount of data that is copied from the DRAM to the SRAM and on into the CRAM. This strategy corre-

```
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.

sponds to maximizing the computation to communication ratio in massively parallel processors.

The percolation of the data from the outside layers of HTMT into its cryostatic core is more easily controlled if we divide the matrices into blocks, and the blocks themselves into sub-blocks. Specifically, matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ of dimension $M \times M$ are divided in t^2 blocks of dimension $b_s \times b_s$; each block is divided into s^2 sub-blocks of dimension $b_c \times b_c$. Thus we have $M = t \times b_s = t \times s \times b_c$. If we denote the mnth block of $\mathbf{C}$ by $\mathbf{C}_{m,n}$ (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:

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

Each block multiplication is performed by s^2 SPELLs, logically organized as a grid, using Cannon's algorithm [4, 11]. Cannon's algorithm is a well-known algorithm for matrix multiplication on a distributed memory machine. We have adopted it in this case because it requires a minimal amount of temporary space within the CRAM and because the uniform communication pattern makes efficient use of the CNET.

Each matrix block is further divided into $s \times s$ sub-blocks of size $b_c \times b_c$. To compute the multiplication of $\mathbf{A}_{m,k}$ by $\mathbf{B}_{k,n}$, the SPELL in position (i, j) of the grid must receive the sub-blocks $\mathbf{a}_{\alpha,\kappa}$ and $\mathbf{b}_{\beta,\gamma}$, where

$$\begin{cases} \alpha &= i + m \times s \\ \kappa &= (i \oplus j) + k \times s \\ \beta &= (i \oplus j) + k \times s \\ \gamma &= j + n \times s \end{cases} \tag{2}$$

and $\oplus$ is the addition module s.

Note that after Cannon's algorithm is used to perform the block multiplication in the grid of SPELLs, the SPELL (i, j) stores a partial result for the sub-block $\mathbf{c}_{ms+i,ns+j}$. There is no need to percolate this partial result outward until an entire row of blocks of $\mathbf{A}$ and an entire column of blocks of $\mathbf{B}$ have percolated to the SPELLs and have been multiplied.

4.1 Percolation from DRAM into SRAM

Figure 3 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 3 is matrix $\mathbf{A}$, the data transformation is the one for SPELL $(2, 3)$. From equations 2, 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 3 we represent the vector that is formed by the data transformation. The sub-blocks are aggregated in SRAM according to the SPELL that is their destination, and not according to their membership to a block. Therefore there is no formation of a monolitic block in SRAM at any time.

We assume that both matrices $\mathbf{A}$ and $\mathbf{B}$ are read into the DRAM from the HRAM in a row first order. It is not necessary for any DRAM to store the entire matrix at a time. For each row of blocks, the DRAM only has to read one row of sub-blocks from HRAM, as illustrated in Figure 3. This reading is performed in lines 5-8 of the algorithm presented in Figure 4. Each DRAM PIM only needs to percolate one sub-block from each block of the matrix. The position of this sub-block is the same in all blocks of the matrix. The for loop starting in line 9 in Figure 4 spans over the blocks in this row. The data corresponding to each sub-block is copied into the SRAM buffer for $\mathbf{A}$ by the for loops in lines 10-12. See [1] for the BDATATRANSFORMATION algorithm that is very similar to the $\mathbf{A}$ transformation, except that the elements of $\mathbf{B}$ are stored in a column first order within a sub-block in the Bbuffer. The time complexity of each data transformation is dominated by the time to copy the rows of sub-blocks in the temporary buffer in line 8. Thus each data transformation takes

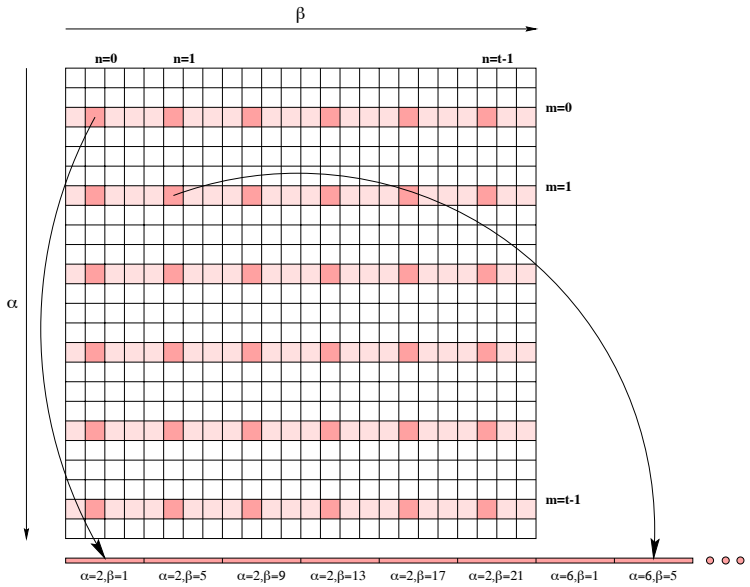

Figure 3. Data transformation performed in the DRAM corresponding to SPELL (2,3) in matrix A to obtain the desired data layout for percolation.

```
ADATATRANSFORMATION(i, j, s, t, b_c, b_s)
1.  a ← 0
2.  ψ ← i + j
3.  if ψ ≥ s
4.     then ψ ← ψ − s
5.  for n ← 0 to t − 1
6.      do for x ← 0 to b_c − 1
7.             do for y ← 0 to (t × b_s) − 1
8.                    do temp[x][y] ← A[n × b_s + i × b_c + x][y]
9.         for m ← 0 to t − 1
10.            do for f ← 0 to b_c − 1
11.                   do for g ← 0 to b_c − 1
12.                          Abuffer[a] ← temp[f][m × b_s + ψ × b_c + g]
13.                          a ← a + 1
```

Figure 4. Algorithm for the Data Transformation of Matrix A in the CRAM.

$O(t^2 \times b_s \times b_c) = O(s \times (t \times b_c)^2) = O(\frac{1}{s}\ M^2)$ time steps for matrices of dimension $M \times M$.

4.2 Communication and Computation Interleaving

Figure 5 summarizes the computation to produce one sub-block of **C** in a SPELL. We want allow significant overlap between the steps of the computation 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** with two columns of sub-blocks of **B** to produce a pipeline effect.

The multiplication of a sub-block in an odd column of **A** by a sub-block in an odd row of **B** is interleaved with the multiplication of a sub-block in an even column of **A** and an even row of **B**. To implement this interleaving we reserve space for three sub-blocks or **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. See [1] for the multi-threaded algorithm that implements the computation interleaving and for the algorithm that implements the percolation of sub-blocks from SRAM to CRAM.

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

Figure 5. Outline of the computation to generate one sub-block of C in a SPELL.

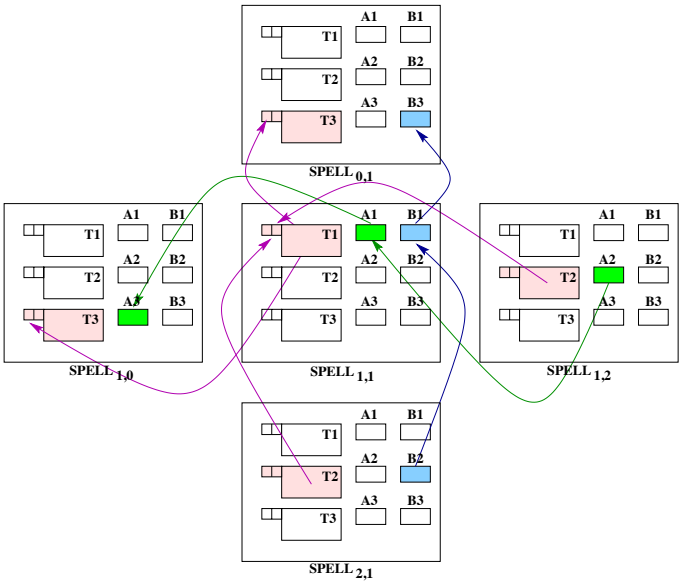

Figure 6. Data movements and exchange of synchronization signals among neighboring SPELLs in the grid.

5 Performance Estimation

Table 1 lists the complexity of each one of the algorithm phases for a single execution of that phase. To obtain overall execution time we need to consider the portion of time spent in each phase, hardware performance for each phase, the number of times that each phase has to be performed and the overlapping among phases.

Since CRAM has the tightest constraint, we will first determine b_c. Assuming CRAM system software and user code consumes 180 Kbytes of the 1 Mbyte CRAM, we set b_c to be as large as possible while allowing seven sub-blocks to be resident in an CRAM as described in Section 4.2. The largest b_c such that $7\,b_c$ words fit in the available CRAM is $b_c = 125$.

The number of sub-blocks per block, t, is determined by the amount of memory available in the SRAM associated with each SPELL. The algorithm described in Section **??** requires 3 blocks per SRAM, and we are assuming that the system software, including RTS, OS and user program space uses 4 Mbytes. The largest t that allows $3(t\,b_c)^2$ words to fit in SRAM is $t = 26$.

Hardware Parameter	Value
CRAM size	128 Kwords
SRAM size	32 Mwords
DRAM size	512 Mwords
DRAM access time	0.97 ns/word
SRAM access time	0.42 ns/word
CRAM access time	30 ps/word
Number of FPU/SPELL	5
SPELL FPU Cycle	15 ps
Vortex port Bandwidth	10 Gwords/s
CRAM-SRAM Bandwidth	64 Gwords/s
CNET bandwidth	16.7 Gwords/s

Table 2. Hardware parameters assumed for the design point for the year 2007

Table 2 summarize the hardware parameters pertinent to our performance estimation. A detailed discussion of this parameters is provided in [1]. For instance, our defini-

Algorithm Phase	Complexity
Data transformation in DRAM	$2\,t^2\,(b_c)^2\,(s+1)$ 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 1. Complexity in terms of amount of information exchanged of operations performed for each phase of the computation.

tion of average access time in, say the DRAM, is

$$\begin{aligned} \mathrm{DRAM}_{at} &= \frac{\mathrm{CPA}}{\mathrm{WPA}} \times (\text{clock rate})^{-1} \\ &= (7/16)(450\times 10^6)^{-1} \\ &= 0.97\,\mathrm{ns/word} \end{aligned} \quad (3)$$

where CPA is the number of cycles per access, WPA is the number of words read/written in one access [1].

If we let T be the total execution time of the dense matrix multiply algorithm in the HTMT machine, and we consider that the task at hand is the multiplication of matrices of dimension $q\ M \times q\ M$, where q is a positive integer [2], then

$$\begin{aligned} T &= 3\,D_t + q^3[IP_{DS} + t\,OP_{CS} + OP_{SD} \\ &\quad + t^3\,(2\,IP_{SC} + s\,\max(M_S\,,\,D_S))], \end{aligned} \quad (4)$$

where D_t is the DRAM data transformation time, IP_{DS} is the time required for the inward percolation of a block from DRAM to SRAM, IP_{SC} is the time required for the inward percolation of a sub-block from SRAM to CRAM, M_S is the time required to compute a sub-block multiplication in the SPELL, D_S is the time to exchange the sub-blocks of matrices **A** and **B** among the SPELLs, OP_{CS} is the time for the outward percolation of a sub-block from CRAM to SRAM, and OP_{SD} is the time for the outward percolation of a block from SRAM to DRAM. If we assume that $IP_{DS} = OP_{DS}$ and $IP_{SC} = OP_{CS}$, and that $M_S > D_S$, then equation 4 simplifies to [3]:

$$\begin{aligned} T &= 3\,D_t + q^3(t^3\,s\,M_S + 2\,IP_{DS} + \\ &\quad t^2\,(2t+1)\,IP_{SC}) \end{aligned} \quad (5)$$

The values for each of the components of equation 5, using $b_c = 125$ and $t = 26$, are presented in Table 3. The

[1] According to the roadmap in [12], for the 2007 design point, we have CPA = 7 and a clock rate of 450 MHz for the DRAM PIM.

[2] In section 4 we described the percolation process to multiply matrices of dimension $M \times M$, the multiplication of matrices of dimensions $q\ Mq\ M$ will require q^3 multiplications of size $M \times M$. Except for the first one, the data transformations in DRAM are overlapped with the remaining of the percolation process.

[3] See [1] for the justification that the time required for a sub-block multiplication in the SPELLs, M_S is larger than the time required for a sub-block shift in the SPELLs, D_S.

Time	Expression	Value
D_t	$2\,(t\,b_c)^2(s+1)\,\mathrm{DRAM}_{at}$	$1.33\,s$
M_S	$2\,(b_c)^3\,\frac{SF_t}{\#\,\mathrm{FPU/SPELL}}$	$11.7\mu s$
D_S	$\frac{(b_c)^2}{CNETBandwidth}$	0.94 s
IP_{SC}	$(b_c)^2\,\mathrm{SRAM}_{at}$	$6.6\,\mu s$
IP_{DS}	$(t\,b_c)^2\,\mathrm{DRAM}_{at}$	$10.3ms$

Table 3. Execution time components.

dominant components are the times required for the data transformations in DRAM, D_t, and the time required to compute the sub-block multiplication in the SPELL, M_S. For a fixed number of SPELLs and fixed block sizes, D_t is determined by DRAM_{at}, and for a fixed number of FPUs per SPELL, M_S is determined by the average cycle time of a SPELL floating point unit, SF_t. Therefore considering all other parameters constant, we have

$$\begin{aligned} T &= 2.8\times 10^9\,\mathrm{DRAM}_{at} + q^3(0.88\times 10^{12}\,SF_t \\ &\quad + 24ms) \end{aligned} \quad (6)$$

Thus the performance, expressed in petaFLOPs, is given by:

$$\begin{aligned} P &= \frac{\#\text{ of FLOPs}}{T\,[s]\times 10^{15}} \\ &= \frac{2\,(q\,t\,s\,b_c)^3\times 10^{-15}}{4.12\,\mathrm{DRAM}_{at} + q^3\,(0.88\,SF_t + 0.024)} \end{aligned}$$

The graph in Figure 7 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, for $q = 10$, $q = 1$ and for two different values for the average access time in the DRAM, expressed in ns/word. For the hardware parameters in Table 2, $\mathrm{DRAM}_{at} = 0.97\mathrm{ns}/word$, $SF_t = 15ps$/FLOP, and $q = 1$, this model predicts performance of 1.1 petaFLOPS for the HTMT architecture.

Acknowledgements

We are thankful to Mikhail Dorojevets and Jay Brockman for the useful discussions and prompt reply to our requests for information about the CNET and SPELLs. This

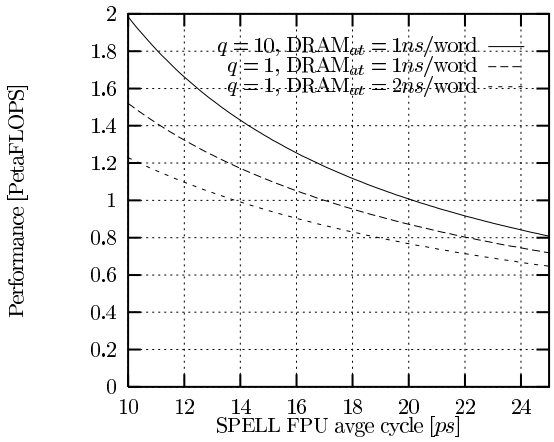

Figure 7. Performance, measured in petaFLOPS, in function of the SPELL Floating Point Unit cycle, SF_t, and of the DRAM PIM access time DRAM_{at}.

analitycal study of performance was made possible by numerous discussions with the HTMT applications and technology experts. 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. We would like to acknowledge support of DARPA, NSA and NASA through a subcontract with JPL/Caltech. The current EARTH research is funded partly by the NSF.

References

[1] J. N. Amaral, G. Gao, P. Merkey, T. Sterling, Z. Ruiz, and S. Ryan. An HTMT performance prediction case study: Implementing Cannon's dense matrix multiply algorithm. Technical Report TM26, University of Delaware, Newark, DE, February 1998. CAPSL Technical Memo.

[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] 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.

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

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

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.