Energy Avoiding Matrix Multiply

Kelly Livingston[1](✉), Aaron Landwehr[1], José Monsalve[1], Stéphane Zuckerman[1], Benoît Meister[2], and Guang R. Gao[1]

[1] Computer Architecture and Parallel Systems Laboratory, Electrical and Computer Engineering Department, University of Delaware, Newark, DE, USA
{kelly,aron,josem,szuckerm}@udel.edu, ggao.capsl@gmail.com

[2] Reservoir Labs, 632 Broadway, New York, NY 10012, USA
meister@reservoir.com

Abstract. As multi and many core chips steadily increase their core count, we observe a phenomenon we call memory hierarchy capacity per capita inversion. To overcome this inversion while remaining energy-efficient, we present a dynamic tiling scheme which we apply to solve the classic Matrix Multiply algorithm. The tiling scheme follows a Hilbert-Inspired Curve strategy to minimize data movement energy, while still allowing for slack and variance within the computation and memory usage of a chip. Our algorithm is energy-conscious: it uses a machine model which does not require symmetric memory (in size or addressing) anywhere in the hierarchy. It only concerns itself with the energy consumption of all memories. This property makes it very robust to chip variance and allows all possible resources to be utilized, which is necessary for future near-threshold voltage designs. Initial results, obtained on a future many-core simulator targeting the Traleika Glacier architecture, give initial estimates of memory reads and writes to all parts of the chip as well as relative energy consumption.

1 Beyond Traditional Tiling: Targeting Exascale

Matrix Multiply (MM) has been studied for decades. Early works presented algorithmic improvements for asymptotic reduction of operations of MM to $O(N^{log_2(7)})$ by trading multiplications for simpler addition and applying recursively [21]. More recent work has looked at communication avoidance by seeking to minimize bytes read per floating point operation and attempting to reach the known lower bound which can provide more locality and less communication [2,11,17]. Other previous works have taken the traditional algorithm and looked within the context of architecture and memory subsystems.

C. Ding et al. (Eds.): LCPC 2016, LNCS 10136, pp. 55–70, 2017.
DOI: 10.1007/978-3-319-52709-3_5

Projects like ATLAS [23] looked to apply auto-tuning techniques so that optimal tiling is created for each memory level, which produced excellent results. As multicore solutions evolved, these solutions and others [1, 10] evolved to better leverage parallelism and solve problems that arise from shared cache structures. Traditionally, lower level data in a cache required replication to higher levels of caches. While we see efforts to advance the efficiency of complex cache hierarchies to loosen this constraint [18] the principle of having larger cache capacity at levels farther from the processor is still true today. However, we see a shift for future architectures starting with GPUs.

We are targeting the Traleika Glacier (TG) architecture, a prototype design chip for exploring Near Threshold Voltage (NTV) computing and an extension of the Runnemede many-core processor architecture [6]. TG is highly hierarchical: execution engines are grouped into blocks; blocks are grouped into units; and units are grouped under a single chip as shown in Fig. 1. The sizes of memory are very unconventional as well. Figure 2 compares the memory hierarchy of a CPU, the 10 core Intel Xeon Processor E5-2470 v2 with a GPU, the NVidia Tesla K80, and TG [6]. As the figure illustrates, for

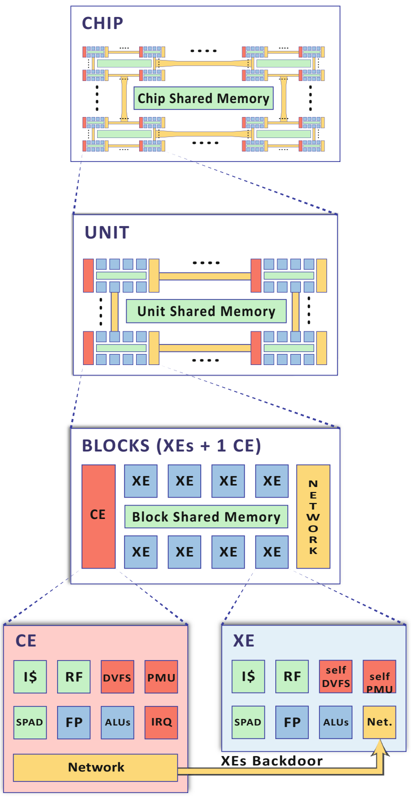


Fig. 1. Traleika glacier strawman architecture

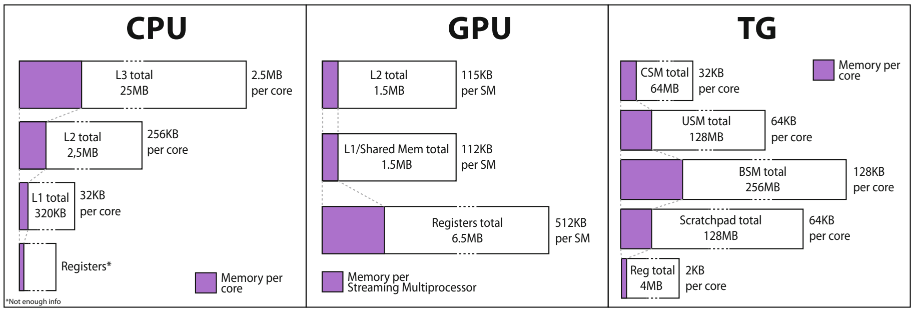


Fig. 2. Graphic representation of capacity per capita inversion

chip designs with dense amounts of compute, the higher level memory would occupy far too much area on die and thus is reduced. This reduction creates a *memory capacity per capita inversion* (CPCI) for the levels of the memory hierarchy. Unfortunately, this inversion violates many of the assumptions made in classical cache analysis algorithms. And it is difficult to analyze the chip as a distributed memory machine since there is still significant locality associated with every memory in the hierarchy. Thus, TG supports configuring all levels of memory as scratchpad or potentially as incoherent cache [16] in order to research the best way to utilize the hierarchy. Our solution for TG similarly follows how GPUs, leveraging the shared memory, permanently store results in the lower levels of memory, leaving the higher level cache for read-only accesses of A and B [19,20,22]. Further, this trend can extend every level of programmer controlled shared memory in a CPCI hierarchy. This opens many possibilities for unique and interesting techniques for utilizing this space including tiling which this paper will leverage.

Section 2 extends tiling, specifically looking at tiling for energy efficiency. Section 3 introduces a novel method for dynamically generating tile shapes using a hilbert inspired ordering. Section 4 combines these two techniques to provide a methodology for creating a tiling scheme for any memory layout and explain how to use asynchronous tasks to build a robust MM algorithm. Section 5 provides specific details about our experimental testbed using the FSIM simulator and the results.

2 Energy Efficient Tiling

2.1 Tiling Principles—The Matrix Multiplication Example

At the core of numerous numerical packages such as LU factorization, MM is an ideal candidate for tiling. In fact, it is a common benchmark or the core routine of benchmarks used to test hardware due to its large reuse of data which can test memory and caching subsystems. It can be computed with a triple nested loop, making the asymptotic computational complexity $O(N^3)$. In this paper, MM is defined as $C_{M,N} = A_{M,K} \times B_{K,N}$, $A, B, C \in \mathbb{R}^2$, $M, N, K \in \mathbb{N}^*$.

There are three traditional ways to tile MM: inner product (*i.e.*, dot product), outer product (*i.e.*, cross product), and a combination of the former two. Inner product ordering reduces accesses to C; outer product ordering reduces accesses to A and B, but requires additional local memory and synchronization. A hybrid combination will perform a trade-off to reuse A, B, and C. Traditionally, a new tile, static in both size and shape, will be used for each level of memory since more temporary space is available at farther memory levels and thus can provide more reuse of A, B, and C. The remainder of this section introduces a novel hybrid method of distributing a tile amongst multiple levels of a CPCI hiearchy with a dynamic shape that can better utilize memory and reduce data movement.

2.2 Energy Efficient Tiling

As previously mentioned, outer product tiling is the only way to provide reuse of the A and B matrix at the expense of more temporary storage and strict synchronization. The resulting energy consumption during computing can be divided up into energy to do compute and energy to move data. As we shrink lithography processes more, data movement and leakage will begin to dominate energy consumption [5]. Since leakage occurs regardless of executing tasks, an algorithm must keep all processors busy with little scheduling downtime. Thus, we also rely on asynchronous fine-grained scheduling in order to keep processors busy where synchronization is occuring, and double buffering to create slack in the synchronization, in way similar to Garcia et al. [14]. For reducing data movement, we propose a method to model the energy consumed by a tiling scheme to quickly determine a near-optimal tile size for a given amount of memory. This method creates a machine model using a few assumptions:

1. Accessing data (read or write) from any kind of memory can be approximated as a particular static cost composed of dynamic access, leakage, and communication energy for both a farther memory and a closer memory.
2. The static cost is the average for all the values of that memory level regardless of variances in location, temperature, or circuit performance.
3. The shared memory structure is physically near all neighbors and the distance travelled dominates the static energy cost function.

The total energy consumed for a subtiling according to these assumptions is modelled in Eq. 1. E is the static energy cost per access to either a memory higher (HM) in both capacity and access energy or a lower memory (LM) in which we are tiling. Matrices are $A_{M,K}$ and $B_{K,N}$ in HM with sub-tiles in LM with dimensions $m \times n$ for outer product and k for inner product.

$$\begin{aligned} HM_{Total} &= 2MN \cdot E_{HM} + \left(\frac{NMK}{n} + \frac{NMK}{m}\right) \cdot E_{HM} \\ LM_{Total} &= \frac{MN}{mn} \cdot \frac{K}{k}(2 + 2k) \cdot mn \cdot E_{LM} \\ E_{Total} &= HM_{Total} + LM_{Total} \end{aligned} \tag{1}$$

In the HM energy consumption, every C result is read and written once because of the inner product ordering of the tile. m and n accesses for the A and B tiles are reduced by using outer product ordering of the smaller subtile. These reductions require increases in access to the lower memory (LM). First, a subtile must read in a partial sum from the LM subtile, then read k values from the A input buffer and k values from the B input buffer, perform k computes, and finally write back the partial sum to the result subtile. This operation is performed for the $m \cdot n$ values for each result tile every $\frac{K}{k}$ synchronization points at the energy cost of LM. Then the final results are written back out to HM, and the procedure will be repeated for the $\frac{MN}{mn}$ number of result tiles needed to complete the matrix in HM. To optimize the energy consumed by data movement, we make several changes of variables and a memory constraint. Let $R = \frac{E_{HM}}{E_{LM}}$ define the ratio of energy consumption from higher to lower memory, and let $S = \frac{m}{n}$ define the ratio of the longest side to the shortest side of the subtile (for this derivation, we assume m is longer). When $S = 1$, the tile is square, and as the tile becomes more rectangular, the squareness factor increases. Equation 1 can then be simplified to Eq. 2.

$$E_{Total} = E_{LM} \cdot \left(MNK \cdot \left(\frac{2}{k} + 2 \right) + \frac{(1+S) \cdot MNK}{Sn} \cdot R + 2MNR \right) \tag{2}$$

Next, we make a memory constraint and thus define Q as the quantity of memory available for tiling in LM. We also will constrain our equation to a tiling scheme which will double buffer the A and B input vectors in order to loosen synchronization requirements which results in a memory constraint definition in Eq. 3.

$$Q = Sn^2 + (1+S) \cdot 2kn \qquad \rightarrow \qquad k = \frac{Q - Sn^2}{(1+S) \cdot 2n} \tag{3}$$

Substituting k in our original expression and simplifying, we derive the total energy consumed as a function of higher tile dimensions, ratios, quantity of memory, and a single variable n to define the subtiling in Eq. 4.

$$E_{Total} = (1+S) \cdot \tfrac{4Sn^2 + (Q - Sn^2) \cdot R}{(Q - Sn^2) \cdot Sn} \cdot MNK \cdot E_{LM} + (2MNK + 2MNR) \cdot E_{LM} \tag{4}$$

And lastly to find the minimum energy, we differentiate and set to 0 in Eq. 5. Solving the quadratic for n^2 we obtain the final equation, Eq. 6.

$$\tfrac{dE_{Total}}{dn} = 0 = -1 \cdot \tfrac{((1+S) \cdot (Q^2 R - 2QS \cdot (R+2)n^2 + n^4 \cdot (R-4) \cdot S^2)}{Sn^2 \cdot (Q - Sn^2)^2} \cdot MNK \cdot E_{LM} \tag{5}$$

The proper amount of memory that should be dedicated to the outer product result tile is a function of the energy access ratio between HM and LM regardless of the shape of the tile. We denote this function as the *fill factor*: it is designated as FF in Eq. 6. It is important to understand that this model is based on the three assumptions where the energy is static, which is not necessarily true. Where the inner product length is extremely short, there will be potential startup overheads that are not amortized such that the energy factor does not properly relate to the real energy cost. Similarly, in the case where the inner product is very long due to a low fill factor, the bandwidth requirements will increase to the HM which typically requires more energy per operation when accessed at higher bandwidths. Thus, this model should only be utilized as a first order approximation strategy for an overall tiling scheme.

$$\begin{gathered} Sn^2 = Q \left(\frac{(R+2) - \sqrt{8R+4}}{R-4} \right) \\ FF = \begin{cases} \frac{(R+2) - \sqrt{8R+4}}{R-4} & R \neq 4 \\ \frac{1}{3} & R = 4 \end{cases} \end{gathered} \tag{6}$$

Other limits and checks should be imposed as well to ensure this is the optimal tiling. One such requirement is that $k \geq 1$ and $n \geq 1$ can be violated by the non-discrete fill factor calculation and by using Eqs. 3 and 6, additional constraints to the tiling shown in Eq. 7 can be added to make sure enough memory is available.

$$1 \leq \frac{Q - Sn^2}{(1+S) \cdot 2n} \qquad \rightarrow \qquad Q \geq \frac{4(S+1)^2 FF}{S(FF-1)^2} \tag{7}$$

Lastly, we previously defined S as $\frac{m}{n}$, where m and n are sides of a full rectangle tile. S' is defined as an imperfectly filled tile which contains work equivalent to S. To do this, the outer product work of the partial tile and the A and B input buffer width requirements of the partial tile are matched to determine what the full tile equivalent would be. After simplification and derivation, it yields Eq. 8.

$$S' = \frac{Inputs^2 - 2Work + Inputs\sqrt{Inputs^2 - 4Work}}{2Work} \tag{8}$$

... with $Work = Sn^2$, and $Inputs = (1+S)n$. We will see in future sections how these constraints and S' can be applied to coarsen tiles and lower runtime overhead, and still ensure that sufficient memory is left for input buffers.

3 Hilbert Inspired Global Layout

Beyond the mathematical modelling used to obtain basic rectangular tiling to assign the proper amounts of A, B, and C tiles in memory level (in the abstract sense), we need an automatic method for explicitly aggregating tiles which creates a tile shape that has the least projected surface area for both dimensions (thus a low S'). Explicit aggregation is important since recursive implicit aggregation like in cache-oblivious algorithms would fail to expand memory consumption in the lower memories. Our method must also be able to adapt to any memory layout, and be robust for any problem dimension. This makes a space filling curve an excellent candidate since these curves map a higher order space into a one dimensional space perfect for linearly enumerating as asynchronous tasks while also ensuring a good amount of locality. Some space-filling curves like Morton curves are computationally very inexpensive, but they have unbounded Hölder continuity and thus if used recursively could lead to large jumps within the matrix. Better candidates are Peano or Hilbert curves. Figure 3 provides examples of Hilbert and Morton curves.

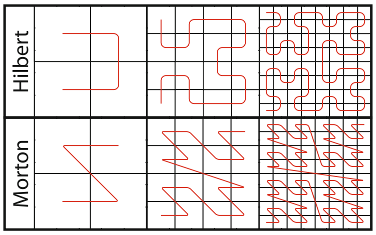


Fig. 3. Order 1 through 3 Hilbert and Morton curves

Once the requirement to replicate data down a cache-like hierarchy is removed, the freedom to pin tiles anywhere in the hierarchy is possible. However, there is no obvious strategy to get the best layout. We present a data layout and an asynchronous scheduling technique which maximizes memory utilization, adapts to different memory sizes, preserves locality even during dynamic throughput changes in processors, and is based on energy optimal tiling principles. This produces tiles in certain memory locations in the method shown in Fig. 8. In order to achieve the properties described, the aggregations are not perfectly square or perfectly filled, which will incur some performance penalty that must be quantified before describing our curve technique.

3.1 Measuring S' Empirically

Hungershöfer and Wierum [15] show that for all sections of a Morton and a Hilbert curve, Hilbert curves have slightly lower average surface area to volume but also contain a higher worst case surface area to volume ratio.

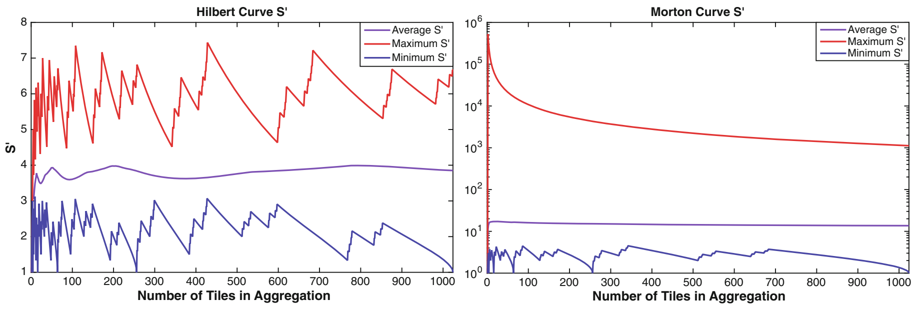
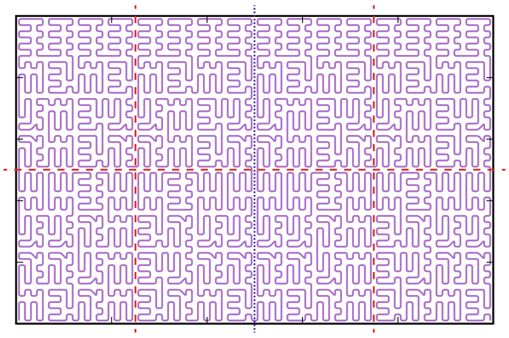


Fig. 4. S': Equivalent full tile aspect ratio matching work: inputs of a partial tile

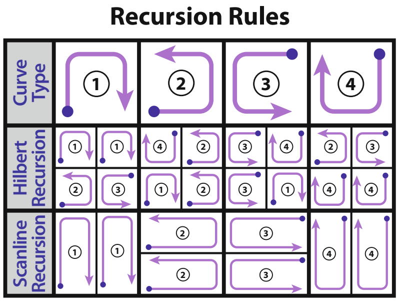
Figure 4 shows our calculations for worst case, average, and minimum S' values for every possible aggregation that follows the curve order for each curve length using a 1024×1024 Morton and Hilbert curves. For S', the Hilbert curve outperforms Morton by a factor of four on average and has a bounded maximum below 8 whereas the Morton curve produces large maximum aspect ratios. This is because S' is more related to projected surface areas than standard surface areas, giving an even larger penalty to Morton curves and making the choice of Hilbert inspired curves as the most reasonable choice.

3.2 Decomposition Rules for Layout

Figure 5a gives an example curve for any arbitrarily dimensioned problem which provides good locality for tiling, which we call *Hilbert Inspired Curve* (HIC). To this end, we implement a pseudo-Hilbert curve algorithm influenced by the works of Zhang *et al.* [24] and Chung *et al.* [9], which will be close to a Hilbert curve in S' performance. Unlike Zhang *et al.* where divisions create splits with sections having power of 2 dimensions on the outer portions of the matrix, our algorithm makes simple divisions by 2, split in both dimensions until we reach a base case. While Zhang's technique generates more regular patterns at the expense of different aspect ratios throughout the matrix, our technique ensures a Hilbert order with as close to the overall aspect ratio of the matrix at the expense of a more complex base case ordering.



(a) An example 50 × 80 Curve



(b) Recursion Rules

Fig. 5. HIC: Hilbert Inspired Curve. (Color figure online)

In order to lower the aspect ratio of the tiles and reduce the expected S', HIC will make scanlines of tiles following Chung *et al.*'s work, rather than using Hilbert recursion.

This is because dividing a rectangular tile into a more square tile occurs only until the longer length switches axes and is no longer smaller than the current S value. This creates the condition shown in Eq. 9.

$$S > \frac{1}{\frac{S}{2}} \qquad \rightarrow \qquad S > \sqrt{2} \tag{9}$$

This is equivalent to always dividing the longest dimension of the tile, similarly to many cache oblivious algorithms, except the single dimension split is only performed when the curve types allow a scanline recursion. This results in the recursion rules laid out in Fig. 5b.

In the base case where either tile dimension goes below 7, HIC terminates recursion and specifies every possible scanline order in a look-up table similar to Zhang. We ensure that a split in the base case cannot result in two odd tiles by shifting the split as necessary. This reduces our look-up table to 4 cases for each curve type, resulting in 64 total scanline orders. Figure 6 illustrates the scanline order for all cases of base tiles for a curve traversing from lower

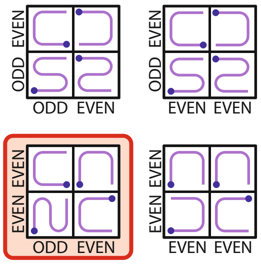


Fig. 6. Portion of scanline look up table

left to lower right. The other three curve orientations are not presented. Dots indicate start points for each scanline and the highlighted case does not have a contiguous end/start connection between the two upper subtiles as previously mentioned.

This is exactly what we see in our example tile from Fig. 5a, with the ratio $8 : 5 > \sqrt{2}$, and so a single dimension cut on the X axis is made in the middle (shown in blue dashes). This produces two tiles with $S = 5 : 4$. Hence Hilbert recursion begins with the first 2 cuts (shown in red dashes). Several aspect ratios were tested to ensure S' values were still reasonable to evaluate the impact of these changes and allow arbitrary matrix dimensions, instead of the traditional Hilbert curve. Results are presented in Fig. 7.

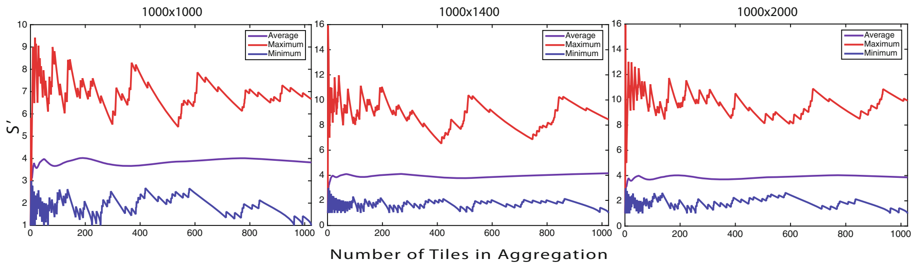


Fig. 7. S' for 3 different aspect ratios

While the maximum value has reached as high as 16 for smaller tile sizes, the overall S' values remain nearly the same as the original Hilbert curve, which will bode well in Sect. 4 when utilizing this curve to aggregate tiles in a CPCI hierarchy.

4 Tiling Up and Down a Hierarchy Efficiently

4.1 Aggregating Tiles

The first step to implement our algorithm is to query the runtime for all program available memory in the chip memory hierarchy. Then a tree is built where the smallest memory closest to the processor is a leaf and the memory shared between different groups of processors are inner nodes. Next the base tile size and inner product length n and k are determined, (see Sect. 2). Of course, a tile size of 1 could work but the overhead of runtime queues, curve pointer calculations, and synchronization would be cost prohibitive. If the base tile size is made too coarse, then smaller regions of memory will be unusable: fragment pieces of memory during tiling create excessive work stealing, and (for small problem sizes) expose too little concurrency. Hence the importance of determining the proper base tile size. For the purposes of our experiments, we picked our base tile sizes empirically, but this process is autotunable.

Bottom-up tile formation starts by attempting to aggregate base blocks together into larger tiles that can form outer products while still having enough memory available for the input buffers. All aggregations must follow a global layout dictated by the HIC. Thus, the task is simply to divide what portions of memory will be A input buffers, B input buffers, and C result tiles for that memory block using the FF equation from Fig. 6 and the HIC curve function to project what inputs will be needed. Once the children of a subtree have finished, the subtree attempts to partition the shared memory using the FF equation with one exception: it regards the value Q in the equation as not only the size of its memory but also the result tiles from all its children in the calculation. This exception is made due to assumption 3 of our machine model from Sect. 2. It is intuitive: any child could steal work from another child at the cost of the shared memory access when gross imbalance occurs. The rest of the memory in the subtree is divided and utilized for the A and B tiles according to the dimensions created by the HIC. Additionally, we insure that an upper level input buffer can hold a large enough buffer (product length of k) to support lower level input buffer reads. This assignment continues sequentially all the way through the memory tree until the root finishes by initiating the first data movement of matrices from DRAM. After all nodes of the memory tree are initialized, data layout is finished and the spawning of tasks for computation can begin.

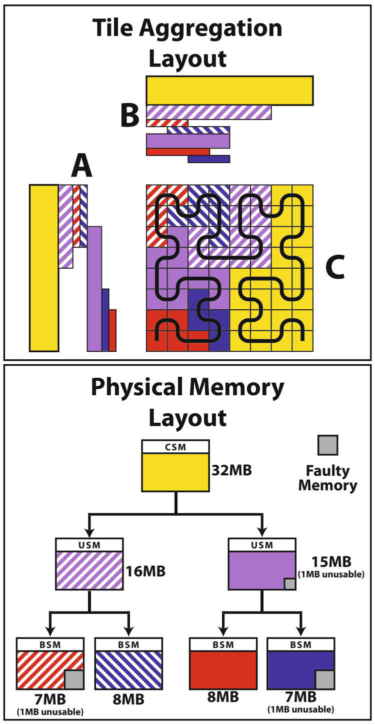


Fig. 8. Example tiling and memory layout (These memory capacities are for illustrative purposes only)

4.2 Creating Tasks

As mentioned earlier, outer product operations have synchronization requirements if multiple operations are occuring in parallel. In order to perform these operations, yet still maintain high performance, we implement a hierarchically double-buffered and load-balanced asynchronous computation. This is similar in style to Garcia et al. [14].

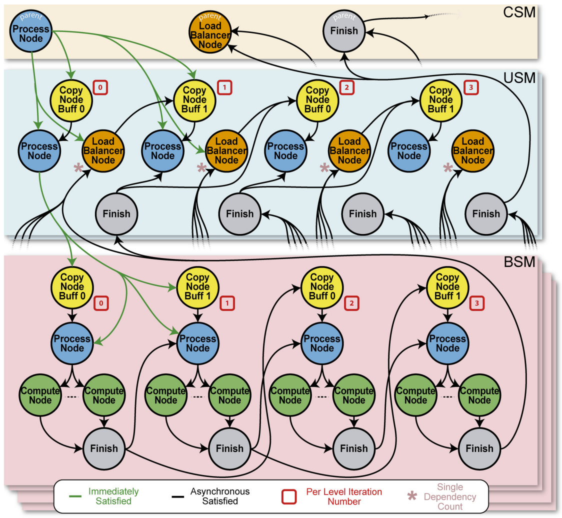


Fig. 9. Data dependency graph

As shown in Fig. 9, there are 4 kinds of nodes that we give to the runtime. Each node has a set of dependencies that must be satisfied before it can placed in the running queue. Similarly, once a node finishes, it will satisfy its future dependencies by making calls to the runtime with a globally unique identifier for each dependency.

Once higher memory level work is available, each execution engine performs direct DMA transfers, bypassing all other memory structures. While we could have provided additional data reuse by recruiting groups of XEs in the same block or unit to perform a similar input broadcast into the lower level buffer just as the lower level tiles did, this would add more synchronization and potentially affect performance.

5 Experimental Results

5.1 Testbed

We experiment using FSim, which is a heavily multithreaded and multi-process functional simulator created by Intel. It models the TG architecture: execution and control engines, load-store queues, memory controllers and memory banks at each level of the TG hierarchy, are all implemented as individual threads. The runtime we used on FSim only allows up to $\frac{1}{8}$ of the targeted 2048-core TG chip to be simulated: up to 4 units of 8 blocks each, with 8 execution engines and one control engine in each block ($\approx$256 cores). Because we are only simulating part of the chip, we reduce the chip area to 64 mm^2 for performing on-chip network energy calculations, and modify the amount of memory in the Unit and Chip shared memories in order to maintain a hierarchy inversion ratio of 2:1 as seen in Table 1.

Table 1. Simulation parameters

Chip shared memory	16 MB
Units/chip	4
Unit shared memory	8 MB
Blocks/unit	8
Block shared memory	2 MB
Base tile size → n	30
Base tile size → k	30

We trace and count all matrix data movements from any memory module in the hierarchy using our runtime; in addition we report relative energy consumption provided by FSim that includes dynamic tiling computations and runtime overheads. However, FSim is not cycle accurate: we are unable to estimate static power consumption or the actual performance of the MM, but all dynamic energy consumption is measured using approximations developed from architectural designs. For this paper, since we are more interested in data movement, we fix the voltage to be in superthreshold operation so all dynamic energy consumption is on that order.

5.2 Tiling Related Results

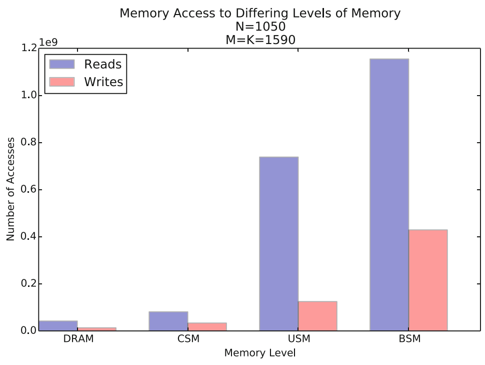


Fig. 10. Memory accesses

Figure 10 shows the number of memory accesses to all shared memories on the chip. Our energy-aware algorithm gives a clear preference for the closer memory operations, preferring to access BSM 20 times more than DRAM. In fact, the algorithm favors the local operations so strongly that *the number of DRAM operations is exactly the lower bound on the number of accesses to do the MM operation*. This is in spite of the C matrix being 83% of the size of the CSM showing that our algorithm can easily operate on working sets larger than the highest capacity of memory in the hierarchy.

It is not necessary to compare this method to other standard cache-oblivious algorithms, since they follow the inclusion property.

This is because any algorithm that only uses the CSM would certainly be unable to fit all 3 matrices in memory, necessitating that at least one of them be accessed a second time. Since all our energy consumption in the on-chip memories is less than 25%, it is already clear that we would consume less energy than any competitor that does not have some kind of explicit outer product layout. This is why we specifically chose this single case to illustrate our point.

5.3 Machine Related Results

The preference for on-chip memory operations over DRAM accesses is very helpful for off-chip bandwidth utilization as well. Given that a $1050 \times 1590 \times 1590$

requires a total of 5.3 GFLOP and our tiling scheme is able to only require 60 MB of loads or stores to DRAM, with throughput levels of 1.75 TFLOP/S which we would expect that $\frac{1}{8}$ of a chip could perform, it would still only require a DRAM bandwidth of 20 GB/s.

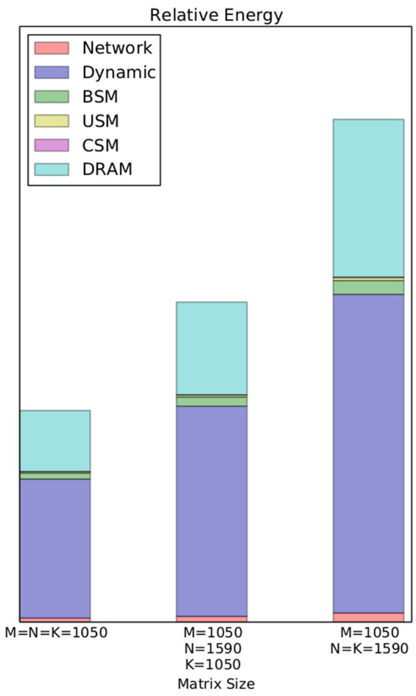


Fig. 11. Relative energy consumption

This could potentially be an even larger reduction in off-chip bandwidth requirements if the full memory capacity were simulated. This comes at a cost with large increases in on-chip accesses which we would expect an on-chip network could handle the added requirements.

Figure 11 shows the relative energy consumption (without static energy) to the 4 different shared memory regions of the chip as well as the dynamic energy consumption of the processors for three different MM sizes. Here we notice that even though the BSM, USM, and CSM are read and written orders of magnitude more than the DRAM, the energy consumed by the DRAM is still much more than the more local memory operations.

6 Related Work

Space-Filling Curves. Chatterjee *et al.* [7] studied recursive data layouts for multiple kinds of Morton curves as well as Hilbert curves in the context of Matrix Multiply, while Bader and Zenger [3] created an algorithm using Peano curves. More recently, Ballard *et al.* [4] used a Morton inspired ordering in which they divide by the largest dimension which in a square matrix resolves to Morton order. These works solely looked at the locality properties of space filling curves in order to provide cache friendly ordering. In addition, our work incorporates a hierarchy of scratchpad memories and ensures the tiling scheme provides energy optimal data movement. Furthermore, this technique also leverages the curve in a scheduler for more choreographed data movement to increase locality.

Cache Oblivious and Communication Avoiding Algorithms. Frigo *et al.* [13] define an algorithm as being cache-oblivious when the algorithm is cache optimal without requiring any parameters defining the cache. They do this by using the inclusion property of caches to simplify the problem into a 2 memory space problem: fast cache memory and slow system memory, similar to our formulation. They then can infer cache optimality for any algorithm that provably minimizes communication between these two memories so long as the algorithm is not a function of the sizes. However, this means exclusive caches or noncoherent caches or scratchpads like the CPCI hierarchies we target can not apply to a cache-oblivious algorithm or if so a complex analysis of the coherence algorithm is necessary to determine what the maximum working set the cache can hold and under what conditions of memory operations that maximum working set can exist. We only require the energy cost to be inclusive and let the capacity be a variable we define in our model. The downside to our algorithm is that it naturally operates using a machine model where all data movement is explicit and formulating an algorithm within a traditional cache hierarchy would be difficult if not impossible for some caches.

More recent work includes communication avoiding (CA) classes [2,11,17]. They extend the cache oblivious concept to networks. CARMA [12] utilizes a breadth-first\depth-first hybrid algorithm that leverages additional available memory to reduce communication across distributed-memory and NUMA machines. It is not obvious how CARMA would handle an inverted memory hierarchy such as TG since it is usually applied to distributed memory systems. Additionally, we assume that energy consumption will be a dominating and limiting factor within a chip in future architectures rather than bandwidth. Because CA algorithms are cache-oblivious, they place equal weight on memory accesses regardless of the energy liabilities they generate which could limit overall performance when thermal constraints are considered.

7 Conclusion

This paper has presented a novel energy-aware algorithm targeting future many-core architectures. It relies on the memory capacity inversion property and applies a custom space-filling curve to implement our tiling method and achieve energy efficient matrix multiplication execution. We provide a demonstrative simulation experiment to show the advantages of our techniques and predict an energy-optimal bandwidth to flop ratio absent of other bottlenecks in the TG design. While this work provides a precise account of dynamic energy expenditure and makes every effort to amoritize overheads properly, a not-yet implemented cycle-accurate simulator would quantify the scheduling overheads of our algorithm, which would allow for the computation of the total estimate of energy per operation. This would inform computer architects in how inverted memory hierarchies could be utilized. Likewise, our machine model is extensible: bandwidth consumption can be modelled, following Chen *et al.*'s work [8]. From a compiler perspective, our proposed algorithm can be integrated in a more general framework, taking advantage of polyhedral models to extend our dynamic space filling curves and energy model. From a runtime standpoint, there is the potential for using runtime information to guide custom schedulers for optimal locality using our framework. Lastly, initial confirmation of the energy model can be empirically made on systems like KNL which have scratchpad modes for the in-package memory.

Acknowledgments. Authors would like to thank Shekhar Borkar, Joshua Fryman, Romain Cledat, Ivan Ganev, Bala Seshasayee and others on the Intel XStack team for information on memory energy ratios, use of FSim, and computing resources. This material is based upon work supported by the Department of Energy [Office of Science] under Award Number DE-SC0008717. This research is also based upon work supported by the National Science Foundation, under award XPS-1439097.

References

1. Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., Tomov, S.: Numerical linear algebra on emerging architectures: the plasma and magma projects. J. Phys.: Conf. Ser. **180**(1), 012037 (2009)
2. Baboulin, M., Donfack, S., Dongarra, J., Grigori, L., Rémy, A., Tomov, S.: A class of communication-avoiding algorithms for solving general dense linear systems on CPU/GPU parallel machines. Procedia Comput. Sci. **9**, 17–26 (2012). Proceedings of the International Conference on Computational Science, ICCS 2012
3. Bader, M., Zenger, C.: Cache oblivious matrix multiplication using an element ordering based on a Peano curve. Linear Algebra Appl. **417**(23), 301–313 (2006). Special Issue in Honor of Friedrich Ludwig Bauer
4. Ballard, G., Demmel, J., Lipshitz, B., Schwartz, O., Toledo, S.: Communication efficient Gaussian elimination with partial pivoting using a shape morphing data layout. In: SPAA 2013, Montréal, Québec, Canada. ACM (2013)
5. Borkar, S.: Role of interconnects in the future of computing. J. Lightwave Technol. **31**(24) (2013). ISSN: 0733-8724
6. Carter, N.P., Agrawal, A., Borkar, S., Cledat, R., David, H., Dunning, D., Fryman, J.B., Ganev, I., Golliver, R.A., Knauerhase, R.C., et al.: Runnemede: an architecture for ubiquitous high-performance computing. In: HPCA (2013)
7. Chatterjee, S., Lebeck, A.R., Patnala, P.K., Thottethodi, M.: Recursive array layouts and fast parallel matrix multiplication. In: SPAA, Saint Malo, France. ACM (1999)
8. Chen, G., Anders, M., Kaul, H., Satpathy, S., Mathew, S., Hsu, S., Agarwal, A., Krishnamurthy, R., Borkar, S., De, V.: 16.1 a 340mv-to-0.9v 20.2tb/s source-synchronous hybrid packet/circuit-switched 16 × 16 network-on-chip in 22nm tri-gate CMOS. In: 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC) (2014)
9. Chung, K.-L., Huang, Y.-L., Liu, Y.-W.: Efficient algorithms for coding Hilbert curve of arbitrary-sized image and application to window query. Inf. Sci. **177**(10), 2130–2151 (2007). Including Special Issue on Hybrid Intelligent Systems
10. D'alberto, P., Bodrato, M., Nicolau, A.: Exploiting parallelism in matrix-computation kernels for symmetric multiprocessor systems: matrix-multiplication and matrix-addition algorithm optimizations by software pipelining and threads allocation. ACM Trans. Math. Softw. **38**(1) (2011)
11. Demmel, J.: Communication-avoiding algorithms for linear algebra and beyond. In: IPDPS 2013 (2013)
12. Demmel, J., Eliahu, D., Fox, A., Kamil, S., Lipshitz, B., Schwartz, O., Spillinger, O.: Communication-optimal parallel recursive rectangular matrix multiplication. In: IPDPS (2013)
13. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: Proceedings of the 40th Annual Symposium on Foundations of Computer Science, FOCS 1999, Washington, DC, USA. IEEE Computer Society (1999)

14. Garcia, E., Orozco, D., Khan, R., Venetis, I., Livingston, K., G. Gao.: A dynamic schema to increase performance in many-core architectures through Percolation operations. In: HiPC 2013, Bangalore, India. IEEE Computer Society (2013)
15. Hungershöfer, J., Wierum, J.-M.: On the quality of partitions based on space-filling curves. In: Sloot, P.M.A., Hoekstra, A.G., Tan, C.J.K., Dongarra, J.J. (eds.) ICCS 2002. LNCS, vol. 2331, pp. 36–45. Springer, Heidelberg (2002). doi:10.1007/3-540-47789-6_4
16. Intel: Strawman system architecture and evaluation (2004). http://tinyurl.com/j6xxg22. Accessed 10 July 2016
17. Irony, D., Toledo, S., Tiskin, A.: Communication lower bounds for distributed-memory matrix multiplication. J. Parallel Distrib. Comput. **64**(9), 1017–1026 (2004)
18. Jaleel, A., Borch, E., Bhandaru, M., Steely Jr., S.C., Emer, J.: Achieving non-inclusive cache performance with inclusive caches: temporal locality aware (TLA) cache management policies. In: MICRO 2010, MICRO '43, Washington, DC, USA. IEEE Computer Society (2010)
19. Juega, J., G'omez, J., Tenllado, C., Verdoolaege, S., Cohen, A., Catthoor, F.: Evaluation of state-of-the-art polyhedral tools for automatic code generation on GPUs (2012)
20. Leung, A., Vasilache, N., Meister, B., Baskaran, M., Wohlford, D., Bastoul, C., Lethin, R.: A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction. In: GPGPU-3, March 2010
21. Strassen, V.: Gaussian elimination is not optimal. Numer. Math. **13**(4), 354–356 (1969)
22. Verdoolaege, S., Carlos Juega, J., Cohen, A., Ignacio Gómez, J., Tenllado, C., Catthoor, F.: Polyhedral parallel code generation for CUDA. ACM Trans. Archit. Code Optim. **9**(4) (2013)
23. Whaley, R.C., Dongarra, J.J.: Automatically tuned linear algebra software. In: SuperComputing 1998, San Jose, CA. IEEE Computer Society (1998)
24. Zhang, J., Kamata, S., Ueshige, Y.: A pseudo-Hilbert scan algorithm for arbitrarily-sized rectangle region. In: Zheng, N., Jiang, X., Lan, X. (eds.) IWICPAS 2006. LNCS, vol. 4153, pp. 290–299. Springer, Heidelberg (2006). doi:10.1007/11821045_31