Millipede: Die-Stacked Memory Optimizations for Big Data Machine Learning Analytics

Nitin[†], Mithuna Thottethodi[*], and T. N. Vijaykumar[*]
[†]NVIDIA (work done at Purdue), [*]ECE, Purdue University
nitin@nvidia.com, {mithuna, vijay}@ecn.purdue.edu

***Abstract*—The technology-push of die stacking and application-pull of Big Data machine learning analytics (BMLA) have created a unique opportunity for processing-near-memory (PNM). This paper makes four contributions: (1) While previous PNM work explores general MapReduce workloads, we identify three application characteristics of most BMLAs: (a) *irregular-and-compute-light* (i.e., perform only a few operations per input word which include data-dependent branches and indirect memory accesses); (b) *compact* (i.e., the relevant portion of the input data and the intermediate live data for each thread are small); and (c) *memory-row-dense* (i.e., process the input data without skipping over many bytes). These characteristics, except for irregularity, are necessary for bandwidth- and energy-efficient PNM, irrespective of the architecture. (2) Based on these characteristics, we propose memory optimizations for a "sea of simple MIMD cores (SSMC)" PNM architecture, called *Millipede*, which (pre)fetches and operates on entire memory rows to exploit BMLAs' row-density. Instead of this *row-oriented* access and compute-schedule, traditional multicores opportunistically improve row locality while fetching and operating on cache blocks. (3) Millipede employs well-known MIMD execution to handle BMLAs' irregularity, and sequential prefetch of input data to hide memory latency. In Millipede, however, one corelet prefetches a row for all the corelets which may stray far from each other due to their MIMD execution. Consequently, a leading corelet may prematurely evict the prefetched data before a lagging corelet has consumed the data. Millipede employs *cross-corelet flow-control* to prevent such eviction. (4) Millipede further exploits its flow-controlled prefetch for frequency scaling based on *coarse-grain compute-memory rate-matching* which decreases (increases) the processor clock speed when the prefetch buffers are empty (full). Using simulations, we compare PNM architectures to show that Millipede improves performance and energy, by 135% and 27% over a GPGPU with prefetch, and by 35% and 36% over SSMC with prefetch, when all three PNM architectures use the same resources (i.e., number of cores and on-processor-die memory) and identical die-stacking.**

I. Introduction

The technology-push of die stacking and the application-pull of Big Data have created a unique opportunity for processing-near-memory (PNM). Die stacking (e.g., Hybrid Memory Cube [1], High Bandwidth Memory [2]) provides unprecedented high-bandwidth connection between memory and processor dies. At the same time, prevalent Big Data machine learning analytics (BMLA) applications process vast amounts of data, are abundantly parallel, and require massive memory bandwidths (e.g., clustering, classification, dimensionality reduction, anomaly detection, and aggregation statistics). BMLAs are important for many industries such as telecommunications, healthcare, banking, insurance, and social media [3], [4]. BMLAs are for unstructured Big Data whereas traditional database analytics are for structured data. This paper identifies and exploits BMLAs' characteristics through memory optimizations to fully utilize die-stacking bandwidth while remaining energy-efficient.

While processing-in-memory (PIM) has been around for decades [5], [6], [7], [8], [9], [10], [11], [12], [13], there have been three problems. The first is the mismatch between DRAM and logic processes, which some past proposals have addressed by advocating PIM with SRAM [14], whereas die stacking offers a higher-density solution. The second, more fundamental, problem is that two-input-one-output operations with more than one large operand pose the difficulty that the processor can be near only one of the operands, requiring massive data movement for the other operand(s) like non-PNM architectures and thereby losing PNM's bandwidth advantage. We show that only one of the input operands is large in most BMLAs. The third problem is the lack of applications with the right characteristics which BMLAs have.

We identify three key characteristics, which most BMLAs either naturally have or can be transformed to have, that fit PNM: *irregular-and-light-compute*, *compact*, and *memory-row-dense*. First, BMLAs often perform data-dependent computation to differentiate among the input data which is fundamental to learning. Such computation involves data-dependent branches and/or irregular memory access to intermediate program state but not the input data (e.g., `counter[label]` for 100 randomly-occurring labels). Further, BMLAs perform only a few operations per input data word (e.g., under 10) requiring low compute bandwidth, so that simple, energy-efficient pipelines suffice. Conversely, compute-heavy applications would be compute-bound and not benefit much from PNM's bandwidth, irrespective of the architecture. Second, BMLAs perform an acute data reduction via summarization so that the output is much smaller than the input (e.g., calculating the cluster centroids through *kmeans*). Consequently, BMLAs access, at a time, often just one input record and a small amount of intermediate program state, the partially-reduced output, which fits in a small local memory (e.g., 8 KB per core). This compact nature avoids PIM's second problem above. Finally, BMLAs process almost all the input data without skipping over bytes, resulting in dense accesses to the memory rows holding the data. This density implies efficiency of memory bandwidth which is PNM's key advantage. General spatial locality (of Map [13], [11]) does not necessarily imply the lack of gaps which is key for bandwidth efficiency.

1530-2075/18/$31.00 ©2018 IEEE
DOI 10.1109/IPDPS.2018.00026

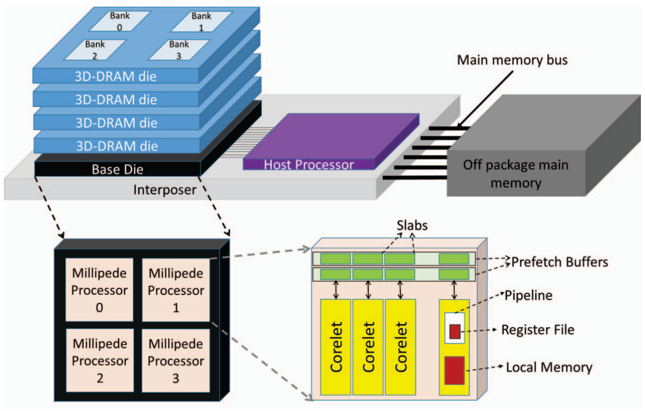

Fig. 1. Millipede Architecture

Except for irregularity, these characteristics are *necessary* for bandwidth- and energy-efficient PNM, *irrespective of the architecture* (Section III-D). Traditional database analytics (e.g., [15]) do not share all these characteristics (Section III).

Following widespread practice for programmability reasons, we use MapReduce [16] (or Spark [17]) to implement BMLAs. While recent PNM work (e.g.,[11], [10]) considers general MapReduce workloads with a broad set of characteristics (Section II), we identify the specific characteristics that fit PNM, which is our first contribution.

Exploiting the high bandwidth of die stacking via PNM architectures based on vector (SIMD), GPGPU (SIMT), or conventional multicore faces difficulties. BMLAs' irregularity makes SIMD and SIMT execution inefficient. While conventional multicore's MIMD execution can handle irregularity, the cores stray far from each other in execution, interleave accesses to many memory rows, degrade row locality, and squander die-stacking bandwidth. Accordingly, we propose memory optimizations for a "sea of simple MIMD cores (SSMC)" PNM architecture, called *Millipede*, targeting BMLAs' characteristics. While conventional multicores employ power-hungry superscalar cores and deep, coherent cache hierarchies, SSMC uses simple cores and no data caches. We envision Millipede to augment the host processor (Figure 1).

Before describing Millipede's novel features, we list the well-known features in its SSMC skeleton: For each (or a few) memory array, there is a Millipede processor comprising a wide set of cores, called *corelets*, to exploit BMLAs' parallelism. Millipede employs MIMD execution for BMLAs' irregularity. BMLAs' light-compute nature affords simple corelets, To hide memory latency of input data, Millipede employs sequential prefetch which exploits BMLAs' row-dense nature; MapReduce's automatic data partitioning obviates cache hierarchies and coherence.

Millipede's novelty stems from *memory-row-orientedness* where a processor's corelets collectively but asynchronously (pre)fetch and operate on entire rows of die-stacked memory before moving on to the next row (i.e., row-centric access and compute-schedule). This row-orientedness, our second contribution, exploits BMLAs' row density to achieve Millipede's goal of utilizing the full die-stacking bandwidth. Such deliberate access-schedule coupling differs from best-effort row locality in conventional multicores which fetch and operate on cache blocks. Recent PNM work does not target row-orientedness, except for considering row locality in *joins* [18] which are fundamentally not compact as we discuss in Section III-D. Millipede implements row prefetching using simple full-empty bits for the prefetch buffers.

Millipede's MIMD execution also incurs the straying problem of conventional multicores and plain SSMC. Because one corelet prefetches for all the other corelets in a Millipede processor, a leading corelet may prematurely re-allocate a prefetch buffer to a new memory row while some lagging corelets have not yet fully consumed the previous memory row. Millipede employs *cross-corelet flow control*, our third contribution, to prevent such premature re-allocation and thus preserves prefetch efficiency despite MIMD. While prefetching is well-studied, the main concerns have been accuracy and timeliness but not premature re-allocation in either self prefetching (i.e., each core prefetches for itself) or cross-core prefetching [19], [20], [21]. Addressing accuracy and timeliness are easy in BMLAs due to sequential input data accesses and loops that can overlap the next row prefetch with the current row computation. While the flow control imposes a global barrier across the corelets, such a barrier occurs only when the prefetch buffers overflow and not at every instruction as in SIMT. Millipede implements the flow control using simple counters for prefetch buffers.

Finally, BMLAs, being compute-light, are memory-bound whose energy can be reduced. To that end, we leverage the prefetch flow control to rate-match the Millipede processor and die-stacked memory via frequency scaling, our fourth contribution. The rate-matching increases (decreases) the processor clock speed whenever a leading corelet finds the prefetch buffers to be full (empty). While the corelets may diverge from each other at fine time granularity, they perform statistically similar amount of work over the full application execution (e.g., 10 billion records). Further, because the same computation is repeated for billions of records, BMLAs' behavior does not change during execution. Accordingly, We employ rate-matching at the coarse granularity (in space) of the processor and not the individual corelets, and (in time) of the full application and not smaller code sections. While rate-matching is well-known, we perform coarse-grained compute-memory rate-matching. Previous work explores rate-matching in hardware at the fine granularity of pipeline sub-components (space) and program phases (time) [22], [23], in the compiler [24], or by trading off accuracy [25]. Our rate-matching needs only a simple incrementer to adjust the clock speed.

To summarize, the key contributions of this paper are:

- identifying irregular-and-light-compute, compact, and row-dense as key application characteristics that fit PNM;
- row-orientedness for SSMC-based PNM architecture;
- flow-controlled cross-corelet row prefetching; and
- coarse-grain compute-memory rate-matching.

Using software simulations running BMLAs, we compare PNM architectures to show that Millipede improves performance and energy by 135% and 27% (198% energy-delay) over a GPGPU with prefetch, and by 35% and 36% (84% energy-delay) over SSMC with prefetch when all three PNM architectures use the same resources (i.e., number of cores and on-processor-die memory) and identical die-stacking.

The rest of the paper is organized as follows. We contrast Millipede to related work in Section II. Section III discusses application characteristics. Section IV describes Millipede's architecture. Section V describes our evaluation methodology. In Section VI, we present our experimental results. Finally, we conclude in Section VII.

II. Related Work

We discuss previous work related to our key contributions.

Application characteristics: As discussed in Section I, previous work [11] explores general workloads, including MapReduce, which include applications without and with inter-thread communication (which inhibits parallelism) or row locality (which degrades memory bandwidth efficiency). In contrast, we identify the key characteristics that fit PNM.

Row-orientedness: Past PNM architecture papers have explored vectors [6], VLIW [8], GPGPUs [26], uniprocessors [7], multicores [6], [9], [27] and SSMC [10], [11]. However, vectors (SIMD), GPGPUs (SIMT), and VLIW perform poorly in the face of data-dependent branches and irregular memory accesses. GPUs employ heavy multithreading to tolerate the latency of unpredictable memory accesses, but the interleaving of numerous contexts degrades cache locality [28] and row locality. While GPGPU's multithreading degree can be lowered (e.g., fewer warps per SM) and supplemented with prefetching for the predictable BMLAs, even 100%-accurate cache-block prefetching does not address GPGPU's difficulty with control-flow irregularity. While conventional multicores' MIMD execution can handle branches, the cores stray from each other due to the unavoidable variability in the record-processing work (as do Millipede corelets without flow-controlled prefetch), interleaving accesses to different rows and degrading row locality. Again, 100%-accurate cache-block prefetching does not address conventional multicores' poor row locality. DIVA [9] targets irregular applications by supporting address translation and coherence. Centip3de [27] exploits die-stacking using a conventional multicore. None of these architectures nor the NDP workshop 2014-2015 papers [29] address row-orientedness (i.e., row-centric access and compute-schedule).

There are several recently-proposed PNM accelerators for various computational patterns. NDA [13] maps dataflow programs to coarse-grain reconfigurable architecture (CGRA) MIMD nodes connected by a network. BMLAs have abundant parallelism and little communication and do not need a general network. Further, NDA is not row-oriented. AC-DIMM [30], based on STT-MRAM, combines ternary associative search with PNM by co-locating key-value pairs in a TCAM. Millipede (a) captures inter-record parallelism whereas AC-DIMM exploits intra-record bit parallelism, and (b) exploits row-orientedness instead of just co-location. While accelerating data reorganization for die-stacked memory [31] is orthogonal to our work, Millipede can leverage this work for the interleaved layout (Section III-B). While Tesseract [12] targets graph workloads via MIMD and inter-core communication, such workloads are not row-dense or compact. Further, Tesseract is not row-oriented and would incur straying similar to conventional multicores and plain SSMC. Other work [32] offloads work to PNM cores upon cache misses. Because BMLA datasets are much larger than caches, PNM execution would be unavoidable.

Accelerators in other compute-intensive domains are specialized for their specific purpose (e.g., [33], [34], [35], [36], [37]). While these accelerators target compute-intensive applications, Millipede targets data-intensive applications.

Flow-controlled cross-corelet row prefetching: While prefetching accuracy and timeliness are well-studied, we focus on cross-core prefetching where one core prefetches for others, as do our corelets, using helper threads [19], in GPUs [38], and in conventional multicores [20], [21]. All but the last paper focus on accuracy or timeliness whereas our concern is cross-core coordination to avoid premature eviction of prefetched data. The last paper regulates each core's prefetches into a shared LLC to ensure equitable sharing of the cache capacity. In contrast, Millipede's flow control ensures cross-core use of prefetched data.

Coarse-grain compute-memory rate-matching: Millipede achieves dynamic compute-memory rate-matching in hardware. While rate-matching is well-known, we target the coarse granularities of entire cores (space) and full applications (time). Previous work has proposed compute-compute rate-matching in hardware in globally-synchronous, locally-asynchronous (GALS) designs [22], [23]. These papers rate-match fine-grained pipeline sub-components running typical sequential programs with fine-grained program phases of variable instruction-level parallelism. Other work employs the compiler and profiling for static, compute-compute rate-matching in streaming applications [24]. Finally, work on compute-pin-I/O rate-matching for multimedia workloads (e.g., h.264) trades-off accuracy for energy by using application hints [25] or heterogeneous cores with varying power, performance, or reliability [39]. In contrast, Millipede saves energy without affecting accuracy.

III. BMLA Characteristics

BMLAs are written commonly as MapReductions [40].

A. MapReduce programming model

BMLA MapReductions process a stream of records. Each Map task sequentially processes a series of records and partially reduces each record's Map output into a local intermediate state. This partial Reduce typically reduces only the records processed by a Map task. In some cases due to local memory limitation, the intermediate state is partially-reduced across a subset of Map tasks that are local to a corelet. Like other MapReductions, BMLA input data is sharded across a cluster (or datacenter) where each node performs its Map and partial Reduce. The data is assumed to be resident in the die-stacked memory, similar to Spark [17], as explained in Section IV-E.

TABLE I
WALK-THROUGH EXAMPLE OF NAIVE BAYES

Pseudocode (Comments in gray)

```
// Single N-dimensional record with associated year
typedef struct {
     int year;
     int X[NUM_DIMENSIONS];
} bayes-struct;

// Dataset – Large collection of records
bayes-struct bayes-struct-array[100000000]

// Live state – Aggregated conditional probabilities (Cprob) of the two classes
int Cprob[NUM_DIMENSIONS][K][2]
int classCount[2]
const int threshold

// PNM code – Map task and combine/partial-reduce
for each record in bayes-struct-array {
     int class
     if (record.year > threshold) class = 1;
     else class = 0;
     for each dim in NUM_DIMENSIONS {
          Cprob[dim][record.X[dim]][class] ++
     }
     classCount[class]++;
}

// Host code – Final reduction
Sum classCount arrays of all corelets.
Sum Cprob matrix of all corelets.
```

The per-node Reduce (using within-node Shuffle) reduces the corelets' partially-reduced outputs. The final Reduce (using cross-cluster Shuffle) reduces the nodes' Reduce outputs to compute the final result.

While the above description holds for any MapReduction in general, our contribution is in identifying the characteristics of irregular-and-compute-light, compact, and row-dense to be suited for PNM architectures. The Map and partial Reduce functions require only a few operations per word but involve data-dependent branches and memory accesses making BMLAs' compute irregular-and-light. Only the input data, and not any other computed data, is large in most BMLAs. The input data access is naturally, or as we show below can be made to be, row-dense and compact. As discussed in Section I, BMLAs naturally accomplish the severe reduction of the huge input data, and therefore, maintain only small amounts of intermediate program state. This small state is in contrast to datacenter-scale MapReductions' intermediate state which can be so large as to spill to the disk (i.e., before being shuffled to the reduce tasks). BMLAs' intermediate state includes any constant data and each Map's partially-reduced output accumulated at any point in execution.

Table I shows the memory organization, the local state and the map/reduce operations needed for *Naive Bayes*, a prominent *supervised classification* BMLA (despite being named "naive"). The code assumes n-dimensional records with an additional year field. Each record is logically in one of two classes depending on whether its associated year exceeds a threshold. The key computation is the counting of conditional probabilities depending on the class of each record. The computation makes row-dense and compact accesses to each record's coordinates in each dimension and its related year (nested loops in PNM code). The computation per dimension is light-weight (single increment of conditional probabilities per dimension and a single increment of record label frequency per record). The computation is irregular because of (1) the branch to identify the records of interest and (2) the indirect data-dependent access of the conditional probability matrix. (Alternately, replacing the indirect accesses with if-then-else constructs, to increment the appropriate counters, would lead to more control-flow irregularity.) Accumulating the counts into a small local state effectively acts as a partial reduction. Finally, the Shuffle and reduction across all corelets and PNM processors occurs at the host processor [10], [13].

B. Layout issues

Because BMLAs' parallelism is primarily inter-record, a "row-major"-like or an "array of structs" layout in memory cannot efficiently capture inter-record parallelism. In this layout, parallel accesses to consecutive records, which would likely fall in different memory rows, would destroy row locality. This layout issue is common to *all* the PNM applications and architectures. A better option is the well-known "column major"-like *interleaved* "array of structs of arrays" layout, where each record is striped across rows and the same field of consecutive records fall in the same row. However, because the words in a row typically outnumber the cores, each core has to process many records. Fortunately, the live state of the records processed by a core can be partially-reduced to prevent state expansion. Nevertheless, because this layout implies that a core (a GPGPU lane, or a Millipede corelet) processes full records, the state needs to fit in the core's resources (else some die-stacking bandwidth is spent on spilling the state to the DRAM). Fortunately, this state can fit in 4-8 KB of local memory for most BMLAs. Because the interleaved layout is well-known, our evaluation uses this layout for *all* three PNM architectures we compare – GPGPUs, SSMC, and Millipede.

C. BMLAs

Table II summarizes the BMLAs we consider. We show that these BMLAs are irregular-and-compute-light, compact, and row-dense. While all applications are light (i.e., no super-linear compute complexity), *PCA* and *GDA* have relatively more compute than the others. Recall our stipulation that the applications are naturally or, with some modifications, can be made compact and row-dense. Some of these BMLAs are naturally compact and row-dense. For example, *kmeans* involves computing the distance from each datapoint in a multi-dimensional space to a set of centroids. Because each datapoint is a simple set of coordinates, the computation is inherently compact. Because every coordinate is used in the distance computation, the computation is dense. The centroids are part of the live state that persists across datapoints and do not affect the computation density or the compactness. However, the distance computation from each of the k centroids may require proportional effort and not constant effort like most of the other applications (i.e., O(k) instead of O(1)).

TABLE II
SUMMARY OF APPLICATION BEHAVIOR

Application	**Input record**	**Per-node live state**	**Operations per byte**
Count	Movie rating	Bin Count	O(1)
Sample Selection	Movie rating	(count, elements) per bin	O(1)
Statistics – variance	Movie rating	Bin count Bin sum of squares	O(1)
Supervised classification (discrete) - Naive Bayes (NB)	N-dim. point + Bin-id	Conditional probability per bin	O(1)
Supervised classification via Euclidean distance	N-dim. point	N-dim. centroids	O(1)- new centroid, O(k)- nearest centroid
Unsupervised clustering via Kmeans (1-iteration)			
Dimensionality reduction via Principal Components Analysis (PCA)	N-dim. point	Mean and covariance	O(1)- mean, O(N)- covariance
Supervised classification (continuous) via Gaussian Determinant Analysis (GDA)	N-dim. point + Bin-id	Per-bin mean, co-variance	

Other BMLAs can be made compact with appropriate data layout. For example, *NB* and *GDA* (Table II) typically process a training set that includes: (1) the coordinates of each datapoint in a multidimensional space, and (2) the bin/class to which the datapoint belongs. Maintaining two separate arrays, one each for data-points and classification, would lead to non-compact, discontiguous accesses. Instead, an array of structs, in which the coordinates of each datapoint and its classification are contiguous, enables acceleration. Subsequently, the applications' compact computation includes partial mean/covariance (for *GDA*) and partial conditional probabilities (for *NB*) depending on the bin to which each data-point belongs. *PCA*, which computes the mean and the covariance matrix, is inherently row-dense and compact.

While Deep Neural Networks (DNNs) are important for image processing and have received much attention recently from computer architects, the BMLAs are commercially-important analytics [3], [4], as discussed in Section I, and do not include DNNs. Further, traditional database analytics (e.g., [15]) do not share BMLAs' characteristics. For example, *scan* is regular and *join* is not compact for unstructured, unindexed data (as is common in BMLAs as opposed to databases). Such a join requires pairwise comparisons of all the records in two large tables, which cannot be made compact because while one of the tables can be tiled and streamed in, multiple passes are needed over the other table. As such, both tables are accessed at high rates. (Databases may employ hash-joins on previously-indexed data, but the hashing incurs its own problem of lack of row locality especially in PNM [18].) While poor row locality would mean poor performance [18], our point is that even with good row locality, *joins* are not compact and therefore would underutilize PNM's bandwidth (Section III-D). Finally, many of the BMLAs are *full applications* which produce final results (e.g., *unsupervised clustering via kmeans* clusters data for a market segmentation analysis).

D. Implications of BMLA characteristics

Absence of the characteristics identified by us have strong implications. (1) Regular computation would mean that vector or GPGPUs may suffice. (2) Compute-heavy would imply compute-boundedness weakening PNM's bandwidth advantage. (3) Not compact (despite transformation) would mean that some data other than one input is large (the old PIM problem, as discussed in Section I). Because the processor can be near only one large data, the other data may be bottlenecked by traditional, non-die-stacked channels and networks, again, weakening PNM's bandwidth advantage (e.g., *join*). While the problem does not occur if the second data is needed at low rates, such a case degenerates to a computation that predominantly uses only one large data (i.e., is compact). Such data movement is discussed in [11] but not the implication of compactness. (4) Not row-dense (despite transformation) would mean memory bandwidth inefficiency and degradation of PNM's advantage.

Except for irregularity, these characteristics are necessary for efficient PNM, irrespective of the architecture. As such, all but the first implication expose the fundamental limits of PNM, irrespective of the architecture. While PNM's advantage is diminished for applications that violate these characteristics, this limitation is not specific to Millipede but applies to any PNM architecture. On the positive side, we have identified BMLAs, which are prevalent in the real world today, to fit within PNM's constraints.

E. Application-architecture match

We briefly consider mapping Naive Bayes (Table I) to a GPGPU, plain SSMC, and Millipede. The mapping for the other applications is similar. The indirect access would cause uncoalesced accesses to the L1 D-cache in a GPGPU. Instead, the per-thread live state can be allocated in the GPGPU Shared Memory and striped across its banks (i.e., i^{th} thread's state in the i^{th} bank). Because Shared Memory has as many banks as lanes with word-level interleaving, the indirect access in each thread can access any word within its bank in parallel with the other threads. The input data is prefetched in cache blocks from the die-stacked DRAM to the L1 D-cache. In plain SSMC, both the live state and the input data are placed in the L1 D-cache to which both cache-block prefetches and demand accesses occur. In Millipede, the live state is in the per-corelet local memory, and the input-data row-prefetches and demand accesses go to the prefetch buffer.

Despite these good mappings, GPGPUs and plain SSMC incur problems which Millipede solves. As discussed in Section II, GPGPUs SIMT incurs performance loss due to data-dependent branches; our results show that GPGPU schemes for branches [41] do not help. While plain SSMC's MIMD can avoid these SIMT penalties, the cores stray from each other due to work variability, interleave accesses to multiple rows, and destroy row locality (as explained in Section II). Synchronizing the cores at each record would push SSMC closer to SIMT and its overheads. As discussed in Section II, even 100%-accurate cache-block prefetches do not help GPGPUs and plain SSMC (row locality is a bandwidth problem whereas prefetching improves latency but not bandwidth). To avoid the SIMT problems, Millipede also employs MIMD. Unlike plain SSMC, however, the row-oriented Millipede prefetches entire rows and employs flow-control to limit the corelets' straying from each other and avoid premature eviction of prefetched data. Thus, Millipede utilizes the full bandwidth while enjoying MIMD's benefits.

Nevertheless, two key advantages of SIMT over MIMD are (1) wide access to registers, caches, and memory greatly amortizes the bandwidth and energy cost of each access, and (2) the amortization of instruction processing costs over multiple threads. However, the branch and memory irregularity in BMLAs impede SIMT execution and renders these advantages less effective. As such, we carefully model these differences between SIMT and MIMD in our experiments.

IV. Millipede

Recall from Section I that there is a Millipede processor for each memory array (or a few arrays), as shown in Figure 1. Like SSMC, each Millipede processor comprises a wide set of simple cores, called *corelets*, which employ some well-known ideas. The key novel ideas are: row-orientedness, flow-controlled, cross-corelet, row prefetching, and coarse-grained compute-memory rate-matching.

A. Corelets (well-known ideas)

To handle our irregular data-dependent branches and memory accesses to the intermediate program state (input data is row-dense and sequential), Millipede employs MIMD execution where each corelet has its own instruction cache. Because BMLA code size is small (e.g., under 4 KB), we broadcast the code once at the beginning of execution. Because the applications are compact, each corelet has a small register file and local memory (similar to the Cell [42]). Recall from Section III-B that the live state fits in the local memory without spilling to the DRAM. Due to MapReduce's automatic data partitioning, BMLA MapReduce code can be compiled to allocate the intermediate state in the local memory, obviating the need for hardware-managed, deep cache hierarchies and coherence.

Because the applications are compute-light, the corelets' pipeline is simple and energy-efficient. Because memory latency is hidden by prefetching, the local memory is small and fast, and the pipeline is shallow, the pipeline hazards are short. Therefore, instead of complex register bypassing and branch prediction, we employ small-scale hardware multithreading like GPGPUs to tolerate the short hazards (e.g., 4 contexts). Each context needs its own registers which are only a few; hence, the register file remains small. The local memory holds the partially-reduced live state which is shared among the contexts and therefore need not be replicated. In our evaluation, we assume that GPGPUs and plain SSMC can also employ such small-scale hardware multithreading to tolerate their pipeline hazards.

B. Row-orientedness

A Millipede processor's corelets collectively but asynchronously fetch and operate on entire rows before moving on to the next row (i.e., row-centric access and compute-schedule). Millipede employs simple row prefetching to exploit die-stacking's full bandwidth for the memory-row-dense BMLAs. Each corelet works on a *slab* of the input data brought into the prefetch buffer (e.g., 64 B). Thus, this deliberate access-schedule coupling preserves full row bandwidth. Recall from Section III-B that in the interleaved layout each slab holds the same field(s) of one or more records whose Map tasks are completely independent of each other. Each corelet runs the Map for each of its records which successively update the partially-reduced intermediate state held in the corelet's local memory. The host CPU runs the per-node Reduce to combine this state from all the processors, as discussed later in Section IV-D.

C. Flow-controlled cross-corelet prefetch

The next row prefetch occurs before the current row processing starts. This simple prefetch could be in hardware or software because the processing of a record is easily identifiable in MapReduce. Each slab is large enough that its processing is enough to hide the next row access latency, else we can prefetch one more row ahead. Our hardware scheme can take hints from software about how far ahead to prefetch (not needed for the BMLAs we evaluate). The prefetch buffer is organized as a circular queue where each entry has an address tag. Because all the corelets execute the same Map code, there may be multiple prefetches to the same row. The first demand access to an entry, identified by a full-empty bit called *prefetch-trigger (PFT) bit* in the entry, allocates a new entry, and triggers the next prefetch. The PFT bit prevents later demand accesses from triggering redundant prefetches, similar to traditional MSHRs.

A central issue is that the corelet's MIMD execution imposes a need for flow control in the prefetch buffer. A leading corelet may surge past the other corelets and issue prefetches for all the free buffer entries, wrapping around to the head entry and later entries which have not been consumed fully yet. A crucial detail, best explained by an example, increases the chances of such premature re-allocation. Assuming 2-KB memory row, 32 cores, lanes, or corelets per processor in SSMC, GPGPU, or Millipede, and 4-way multithreading, there are 512 records per row in our interleaved layout (4 bytes per word) and 128 concurrent threads each of which processes only 4 records per row, irrespective of the PNM architecture. This low number implies high work variability across the threads increasing the chances of the premature re-allocation; a higher number would have the variation-reducing effect of statistical smoothing.

Accordingly, Millipede smooths out the variability across the threads by processing more records per thread resulting in the concurrent processing of more rows (e.g., 16 rows). To prevent this concurrency from degrading row locality, Millipede enforces loose synchronization via a per-prefetch-

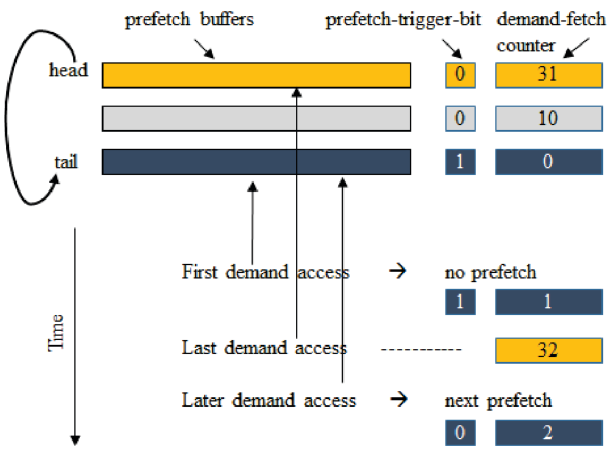

Fig. 2. Flow control operation

buffer entry *demand-fetch (DF) counter* which is incremented upon every demand fetch. A prefetch buffer entry is re-allocated only after the DF counter saturates at the corelet count, indicating that the entry has been consumed fully. The DF counter is reset upon re-allocation and subsequent prefetch fill. Unlike the PFT bit, the DF counter cannot just be a full-empty bit because we do not know which corelet would be the last consumer and hence we must count. Normally, the next entry would be consumed fully (i.e., its DF counter would be saturated) and can be re-allocated for the next prefetch (i.e., the circular queue is not full). When the queue is full, however, the DF counter of the next entry (the head entry) would be unsaturated ('First demand access' in the timeline in Figure 2). Then, the leading corelet does not trigger a prefetch upon a demand fetch to the tail entry even if the tail entry's PFT bit is set (implying the first demand access to the entry). A later access to the head entry causes the entry's DF counter to saturate ('Last demand access' in Figure 2). The next demand fetch to the tail entry issues the next prefetch and clears the PFT bit ('Later demand access' in Figure 2). As BMLAs access rows sequentially, the head entry's DF counter is guaranteed to saturate before the last demand fetch to the tail entry, ensuring that the next prefetch is not missed.

Because of their MIMD execution, the corelets access the prefetch buffers at different times. For full bandwidth to all the corelets, we break up each prefetch buffer entry into as many slabs as corelets so that a slab is accessed by only one corelet. Thus, each corelet's access goes only to a small slab-wide slice of the prefetch buffer entries (e.g., 64-byte slabs and 16 entries means 1-KB prefetch buffer slice). By using fixed-size slabs, the interconnection between the prefetch buffer and the corelets remains simple (see Figure 1). In our interleaved layout, a slab contains either words each from a contiguous set of records (word-interleaving), or n contiguous words of a record (slab-interleaving). The latter has many choices for n such that 'n $*$ coreletCount $*$ multithreadingDegree $=$ row size' for typical values. Thus, each corelet can flexibly process one or more records. While GPGPUs must use word-size columns to achieve coalesceable accesses (wider columns would mean the lanes' accesses span multiple cache blocks), Millipede can use wider columns for layout flexibility.

The full-row prefetch and the flow control are fundamental for Millipede but not the prefetch buffers. The prefetches can bring the data into the local memory instead of the prefetch buffers. The slabs from a prefetched row would go to their respective corelets. Then, the PFT bits and DF counters would exist without the accompanying buffers. However, such an implementation would need address tags in the local memory to detect prefetch data whereas the above implementation needs tags only in the prefetch buffers.

We make a few observations: First, while more prefetch buffer entries achieve more statistical smoothing across the threads, some variability remains although the probability is lower. Thus, our flow control incurs less, but non-zero, waiting. Second, without flow control, this variability still causes premature prefetch buffer evictions, as shown by our results (Section VI-A). Further, once a premature eviction occurs (non-zero probability), the lagging corelets miss in the prefetch buffer and are exposed to die-stacked memory latency which widens the lag for long periods of time (i.e., left to chance, there is little self-correction). Our results show that this lag is worse for performance than our flow control's short waiting. Finally, software barriers across Map tasks could prevent premature evictions instead of our hardware flow control. A barrier after all the entries are consumed would incur the barrier cost only once for all the entries. However, the Map tasks operate on record granularity while the prefetch buffer holds arbitrary number of bytes. Hence, the barrier may fall in the middle of records which would break MapReduce's task granularity and not be expressible. Generic thread-based *application* programming, instead of MapReduce, would work but would severely degrade data-center/cluster programmability. Our results show that placing software barriers at record granularity within MapReduce does not help. Because the full records far exceed the prefetch buffer entries (many tens to a few hundreds of rows in our interleaved layout versus 16-32 entries), the barriers are not invoked when needed. Instead, Millipede employs simple hardware to maintain programmability.

D. Final Reduce

The per-node Reduce and global final Reduce (Section III-A) are much smaller than the Map and partial Reduce but require data from all the processors within a node and all the nodes, respectively. Therefore, providing communication support for these Reduce phases may not be worth it. Instead, the host CPU performs the per-node Reduce, as observed in [10], [13]. The global final Reduce uses the cluster network like other MapReductions. For example, Map and partial Reduce of tens of millions of records in each node take a few seconds versus per-node Reduce across 32 Millipede processors of a node takes hundreds of microseconds and the global final Reduce across 5000 nodes of a cluster takes tens of milliseconds.

E. Memory Interface

For simplicity, Millipede assumes a discrete GPU-like memory interface where the host CPU copies the input data into the die-stacked memory before processing and copies out the output data in the corelets' local memories after processing. The die-stacked memory and local memories are not part of the host CPU's physical memory address space. The corelets do not support virtual memory or coherence with the host, which we leave for future work.

Loading the input data into the die-stacked memory for every run would *fundamentally* make BMLAs, or any application, host-memory-bound (or disk-bound, if the data is spilled to the disk), rendering die-stacking bandwidth irrelevant for *any* PNM architecture – GPGPU, SSMC, or Millipede. Instead, the input data resides in the interleaved layout in the datacenter/cluster's die-stacked memory like Spark, Web-Search's memory-resident Web Index, or memory-resident databases, and unlike generic MapReduce (Section III-A). Further, to amortize the disk and layout costs, numerous MapReductions (chained or not) reuse the data similar to Web-Search/memory-resident database queries.

F. Compute-memory rate-matching

Being compute-light, most BMLAs are memory-bandwidth-bound. Our rate-matching eliminates the idling energy incurred when the corelets wait for memory, To that end, we leverage our flow-controlled prefetch to rate-match the Millipede processor and die-stacked DRAM via dynamic frequency scaling (DFS). Our evaluation conservatively assumes that voltage scaling is impossible; otherwise, our energy savings would be higher. While the corelets may diverge from each other at fine time granularities, they perform statistically similar amount of work over the full BMLA execution (e.g., billions of records). Further, because the same computation is repeated for billions of records, BMLA behavior does not change across code sections. Accordingly, our rate-matching is at the coarse granularity (in space) of the processor and not the individual corelets, and (in time) of the full application execution and not smaller code sections.

Because of the coarse application-level granularity, we employ a simple, one-dimensional hill-climbing control algorithm that decreases (increases) the processor clock speed in small steps (e.g., 5%) via frequency scaling whenever a leading corelet (defined in Section IV-C) finds the prefetch buffers to be empty (full), signifying a memory-bandwidth-bound (compute-bound) application. The small steps suffice due to the application-level granularity where the algorithm needs to converge just once at the start of the application whose compute-work behavior does not change later. For instance, small 5% steps, a large 4x required change in the clock speed, and 200 cycles of computation per DRAM row (typical for BMLAs) imply convergence in 16,000 cycles compared to a few billions of cycles of execution time. Any oscillations after convergence would be within a band of the size of the small step, resulting in acceptable inefficiency.

TABLE III
HARDWARE PARAMETERS

#Millipede processors # GPGPU SMs # SSMC processors	1 of 32
Compute clock	700 MHz
# Corelets/lanes/cores per processor/SM/SSMC	32
# Multithreading contexts per processor/SM/SSMC	4
# Registers per corelet/lane/core	32
L1 I-cache per corelet/SM(not lane)/core	4 KB, 128B line,
Local memory per corelet Prefetch buffer per corelet L1 D-cache per SM Shared memory per SM L1 D-cache per core	4 KB 16 x 64B 32 KB, 128B line 128 KB, 4B interleaving 5 KB, 128B line
Die-stacked DRAM capacity	4 GB
# Die stack layers	4
# Memory channels	1 of 32
Channel clock	1.2 GHz
Channel width	128 bits
DRAM tCAS-tRP-tRCD-tRAS	9-9-9-27
DRAM row size, banks/channel	2 KB, 4
Memory Controller	FR-FCFS (16 deep)
DRAM Access Energy	6pJ/bit [31]
Core Technology node	22nm

Overall, Millipede adds simple hardware – PFT bits, DF counters, and adders (for the above 5% steps) – to SSMC.

V. Methodology

We modify GPGPUsim to implement Millipede. For comparison purposes, we use GPGPUsim to simulate PNM architectures based on a GPGPU, Variable Warp Sizing (VWS) [41] which is currently the best branch-optimized GPGPU (for BMLAs' branches), and SSMC (representing previous multicores without row-orientedness [11], [10], [12]). This SSMC matches Millipede in compute bandwidth, unlike conventional, superscalar-core multicore which has large caches and far fewer high-performance but power-hungry cores. Because narrower GPGPU warps lose less performance in the presence of branch divergence and wider warps achieve lower energy otherwise, VWS dynamically chooses between 4-wide and 32-wide warps based on branch divergence. To capture their MIMD execution, we simulate an SSMC processor and Millipede processor each as an SM with only one lane corresponding to a simple core or corelet. Our simulation assumes that each PNM architecture (i.e., SSMC, GPGPU, VWS, and Millipede) is on the logic die with stacked DRAM and is separate from the host CPU. In addition, we ensure that the number and pipeline of the cores and the on-processor-die memory size are identical in all these PNM architectures. All the architectures use the interleaved layout (Section III-B). While Millipede uses sequential row prefetch, the GPGPU, VWS, and SSMC use sequential cache-block prefetch. *Thus, our results isolate the benefits of Millipede's novel features while holding the effects of technology (on-die transistor count and die-stacking), well-known architecture schemes (simple cores, hardware multithreading, and sequential prefetch) and software (interleaved layout) to be the same for all the PNM architectures we compare.*

TABLE IV
BENCHMARK PARAMETERS AND CHARACTERISTICS

Benchmark	instrs per input word	Branches per instruction	SSMC's row miss rate	Rate-match clock (MHz)
count	7	0.14	0.253	544
sample	10	0.2	0.162	528
variance	12	0.08	0.351	581
nbayes	14	0.11	0.344	565
classify	40	0.05	0.393	625
kmeans	44	0.05	0.384	613
pca	150	0.02	0.489	644
gda	180	0.015	0.497	644

The hardware parameters are shown in Table III. We simulate a 32-corelet Millipede processor, a 32-core SSMC, and a 32-lane GPGPU SM. VWS varies the warp widths on the same SM. All the PNM architectures use simple, in-order-issue pipelines with 4-way hardware multithreading to tolerate pipeline hazards. Each corelet has 4-KB local memory and 1-KB prefetch buffer (total 160 KB per processor); each SSMC core has 5-KB L1-D (160 KB per SSMC processor); and each GPGPU SM has 32-KB L1-D and 128-KB Shared Memory (total 160 KB per SM). Recall from Section III-B that BMLAs' compact nature implies that *all* of the intermediate state and input prefetch data *completely fit* in the small local memory (or L1 D-cache), obviating the need to experiment with larger L1 D-caches. Each Millipede corelet and each core in the SSMC has an L1 I-cache. The GPGPU SM has an L1 I-cache shared among the lanes. We account for the extra I-cache in Millipede and SSMC in the energy estimates. The die-stacked DRAM's parameters, shown in Table III, are typical [31]. The bandwidth is similar to HBM's specification of 128-bits per bank, with 1 Gbps bandwidth per pin [2].

We implement the applications in Table II in CUDA. Table IV shows the instruction count per input word, the branch frequency, the row miss rate (row misses / row accesses) in SSMC and Millipede's rate-matched clock speed. To achieve realistic simulation times, we limit the input data to 128MB and run the benchmarks to completion on one processor. Being repetitive, BMLAs behave identically for large-enough and larger inputs. As such, the steady-state behavior (achieved well before 128 MB), will not change with larger datasets or more processors.

We use GPUWattch [43] to estimate energy (parameters in Table III). Recall from Section III-E that BMLAs access the input data (prefetches) and intermediate live state. In GPGPUs, the live state is in the Shared Memory and the input data is cache-block prefetched into the L1 D-cache. The live state is not in the L1 D-cache because BMLAs' indirect memory accesses would cause uncoalesced accesses to the L1 D-cache whereas the Shared Memory supports 32 uncoalesced word accesses, one from each lane, by using 32-way banking and a 32x32 switch. While the Shared Memory is power-hungry, the GPGPU enjoys the energy benefits of wide accesses to the register file and L1 D-cache, and shared access to the L1 I-cache whenever SIMT execution succeeds (Section III-E).

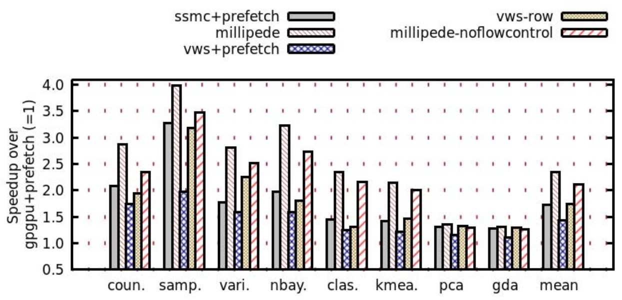

Fig. 3. Performance

However, such success is not often due to BMLAs' control-flow and memory irregularity (Section III-C). We ensure that these benefits do not exist in Millipede and SSMC due to their MIMD execution. In SSMC, the live data, input data cache-block prefetches and demand accesses all go to the L1 D-cache. In Millipede, the live state is in the per-corelet local memory, the input data is row-prefetched into the prefetch buffers (Section IV-C). Both the local memory and prefetch buffer are small and therefore dissipate low power. Each architecture's dynamic energy includes the components for the core (which includes the pipeline energy, L1 I-cache, local memory (or L1 D-cache, as appropriate) and idle dynamic energy due to imperfect clock gating), the DRAM energy dissipated in the stacked memory dies, and leakage energy of the logic die.

VI. RESULTS

We start by comparing Millipede against PNM architectures based on GPGPU and SSMC in terms of performance and energy. We isolate the impact of each of our contributions: memory row-orientedness, cross-core flow-controlled prefetching, and coarse-grain compute-memory rate-matching (application characteristics is covered in Section III-C). We then study Millipede's sensitivity to the system size and the number of prefetch buffers.

A. Performance

We compare a 32-corelet Millipede processor against 32-lane GPGPU and VWS SMs and a 32-core SSMC, all with input data cache-block prefetch into their L1 D-caches. Figure 3 shows these architectures' performance, on the Y axis, normalized to that of GPGPU and the benchmarks on the X axis in increasing order of the number of instructions per input data word to show the trends clearly. The graph also shows (1) Millipede without flow control (Millipede-no-flow-control) to isolate the impact of flow control, and (2) VWS with row-orientedness and flow control (*VWS-row*) to show Millipede's generality. Table IV shows the number of instructions per input data word, the branch frequency, and the row miss rate in SSMC. We discuss the rate-match clock in Section VI-B.

GPGPU (with prefetch) loses performance due to branches impeding SIMT execution (Table IV) but not due to irregular memory accesses which are handled by Shared Memory. SSMC (with prefetch) performs better than GPGPU but loses performance due to the cores straying from each other and degrading row locality (Table IV). In contrast, Millipede's row-oriented MIMD architecture avoids both problems. Millipede performs, on average, 135% and 35% better than GPGPU (with prefetch) and SSMC (with prefetch), respectively. VWS (with prefetch) always chooses 4-wide warps for better branch handling (8 4-wide concurrent warps), and hence performs better than GPGPU. However, VWS still loses performance due to (1) the remaining branch inefficiency, and (2) the warps interleaving accesses to different rows and degrading row locality. Branch inefficiency remains in VWS because BMLAs' data-dependent branches have 70-/30+ taken-or-not split (unlike loop branches' 90+/10-) resulting in under 25% chance that a warp's 4 threads are either all taken or all not-taken. VWS-row performs better than VWS by curbing VWS's straying, confirming Millipede's generality, but still lags Millipede due to the remaining branch inefficiency.

The difference between Millipede and GPGPU highlights the need for MIMD because both architectures enjoy row locality whereas the difference between Millipede and SSMC highlights the need for row-orientedness because both architectures employ MIMD execution; the *only* differences between Millipede and SSMC are row-orientedness and flow control. GPGPU would not benefit from these Millipede features because GPGPU already achieves row locality due to SIMT which, however, incurs branch problems. These numbers isolate the impact of Millipede's novel architectural features over GPGPU and SSMC while holding technology (CMOS and die-stacking) and software (layout) effects constant. Millipede's speedups are due solely to the architecture and extend beyond the die-stacking benefits of 400-800% higher bandwidth over pins. Moreover, Millipede achieves these speedups through simple hardware additions.

Comparing Millipede-no-flow-control and SSMC isolates the benefits of row-orientedness. The former includes row-centric access and compute-schedule via full-row prefetching (Section IV-B) but not flow control so that filling up of the prefetch buffers can cause premature eviction of prefetched data due to corelet straying (Section IV-C). However, such eviction is not frequent with 16 buffers allowing Millipede-no-flow-control to improve over SSMC. Adding flow control improves performance further by fully avoiding such evictions (the Millipede bars). Thus, this graph isolates the benefits of Millipede's row-orientedness and flow-controlled prefetch (our second and third contributions). Flow control alone cannot be applied to Millipede's SSMC skeleton without row-orientedness because flow control is for preserving row-orientedness in the face of the straying of the corelets. Using software barriers at record granularity (Section IV-B performs similarly to Millipede-no-flow-control because the barriers are too infrequent to be effective (not shown). Millipede's rate-matching is an energy optimization analyzed next.

The benchmarks are roughly in the order of decreasing branch frequency and increasing row miss rate in SSMC from left to right in Figure 3 (top to bottom in Table IV). Accordingly, Millipede's MIMD advantage over GPGPUs decreases from left to right causing Millipede's speedups roughly to decrease. However, Millipede's row-orientedness advantage over SSMC increases from left to right causing the gap between Millipede's and SSMC's speedups roughly to increase barring for *PCA* and *GDA* whose heavy compute breaks this trend.

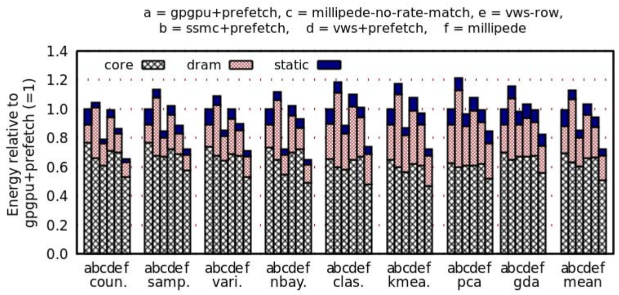

Fig. 4. Energy

B. Energy

We now compare the PNM architectures in terms of energy. Figure 4 shows the architectures' energy on the Y axis, normalized to that of GPGPU. We show two variants of Millipede, one with rate-matching and the other without rate-matching. Each bar shows the breakdown between core energy (which includes cores, caches and idle dynamic energy), DRAM energy, and static leakage energy of the cores as stacked bars.

GPGPU incurs higher core energy than SSMC due to (1) higher Shared Memory energy than SSMC's L1 D-cache (Section III-E), and (2) higher idle energy due to branches (Section III-E). Further, while GPGPUs are more energy-efficient than typical multicores using power-hungry, superscalar cores, the SSMC cores here are identical to the simple GPGPU lanes. However, GPGPU achieves lower DRAM energy than SSMC because, unlike MIMD SSMC, GPGPU does not degrade row locality. The net result of these factors is that SSMC expends more total energy than GPGPU. Nevertheless, BMLAs' irregularity shrinks GPGPU's energy (and performance) advantages over SSMC as compared to typical, regular GPGPU workloads. Comparing SSMC to Millipede-no-rate-match, we see that the latter achieves similar core energy as the former because both architectures avoid (1) the crossbar energy of GPGPU's Shared Memory via private, local memories, and (2) GPGPU's branch inefficiency via MIMD execution. However, Millipede-no-rate-match achieves lower memory energy due to its row-orientedness. By incurring lower branch inefficiency but poorer row locality than GPGPU, VWS lies between GPGPU and SSMC. VWS-row lowers memory energy over VWS via better row locality due to row-orientedness but still lags Millipede-no-rate-match in core energy due to the remaining branch inefficiency.

Recall from Section IV-F that Millipede's rate-matching slows down the corelets when applications are memory-bandwidth-bound. Figure 4 shows Millipede with rate-match further reduces the core energy over Millipede without rate-matching by 16%. While the nominal frequency is 700MHz (Table III), Table IV (column 5) shows the clock speeds under rate-matching which inversely correlate with the number of

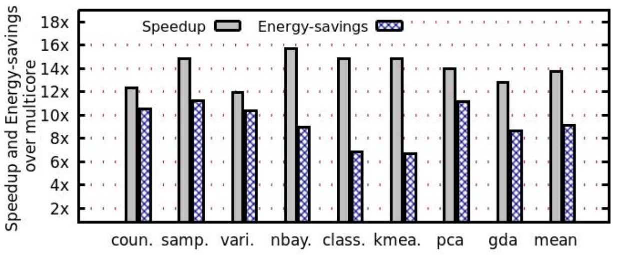

Fig. 5. Millipede versus conventional multicore

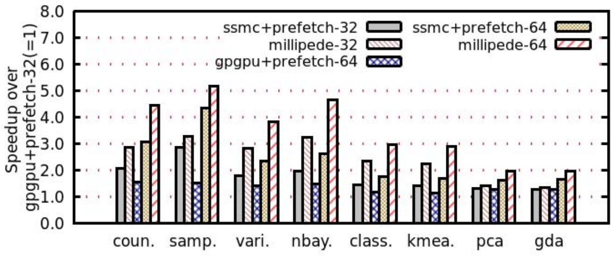

Fig. 6. Speedup versus system size

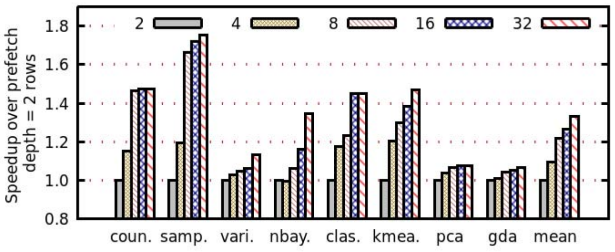

Fig. 7. Speedup versus prefetch buffer count

instructions per input word in Table IV (i.e., fewer instructions implies more DRAM-bandwidth-bound and therefore slower clock). These numbers isolate the impact of Millipede's rate-matching (our fourth contribution). While static *power* of the cores and caches are comparable across the architectures except for GPGPU's lower I-cache power (Section V), Millipede incurs the least static *energy* due to its shortest run time. Overall, Millipede with rate-matching dissipates 27% and 36% less energy than GPGPU and SSMC, respectively. Though SSMC is closer to Millipede in performance for *PCA* and *GDA* than the other benchmarks (Figure 3), SSMC incurs higher energy than Millipede for these benchmarks (Figure 4) due to numerous row misses (Table IV) which can be hidden in execution time but not in energy.

C. Comparison to conventional multicore

To compare Millipede against a conventional multicore (as opposed to SSMC), we simulate an 8-core Xeon-like system with 4-wide, out-of-order issue, 4-way SMT pipelines running at 3.6 GHz, multi-level cache hierarchies (64-KB L1, 1-MB/core L2), and an off-chip memory (one-fourth bandwidth of die-stacked memory). We assume 70 pJ per bit for off-chip memory access [44]. Figure 5 shows Millipede's performance and energy improvements. The far fewer compute threads in the multicore (32) compared to those in Millipede (4096 in 32-processor Millipede) account for most of the speedups. The multicore's high clock speed and off-chip memory access energy are the reasons for most of the energy benefits. However, these large improvements (on average, 125x better energy-delay) come with a caveat. Using many, simple cores for abundantly data-parallel workloads, where single-thread performance does not matter, is well-known. In contrast, the multicore has fewer, complex cores for single-thread performance (e.g., database transactions) where simple cores would perform worse. Similarly, the performance and energy benefits of die-stacked memory stem from die-stacking technology and not the architecture. Instead, our above comparison to GPGPU and SSMC isolate the true benefits of Millipede's novel features while holding all other factors constant. Indeed, the above arguments apply to other recent accelerator architectures and not only to Millipede, and should be kept in mind when comparing Millipede's improvements against those architectures' improvements.

D. Sensitivity to system size

We change the number of corelets, lanes, and cores per Millipede processor, GPGPU SM, and SSMC processor, from 32 (default) to 64, and correspondingly double the memory bandwidth. Figure 6 shows the performance of the three PNM architectures normalized to that of a 32-lane GPGPU. As the lane count increases, GPGPU's branch inefficiency increases compared to Millipede which can gainfully utilize more corelets. Consequently, Millipede's speedup over GPGPU increases with more corelets. Similarly, as the core count increases, the SSMC cores stray from each other more disrupting row locality more. Therefore, Millipede's speedup over SSMC also increases with more cores.

E. Sensitivity to prefetch buffer count

Recall from Section IV-C that the prefetch buffers decouple the corelets from each other by absorbing any temporary work imbalance among the corelets. We vary the prefetch buffer count as 2, 4, 8, 16 (default), and 32 in Figure 7. As expected, more buffers improve performance by absorbing more imbalance though the incremental improvement decreases as the exposed imbalance decreases. Performance levels off around 32 buffers which amount to a reasonable 64 KB per Millipede processor for 2-KB rows.

The record size is independent of the memory row size in our interleaved layout where each record is laid out vertically across the rows (Section III-B). Memory rows are long enough in practice to keep 32-64 corelets busy (e.g., 2-4 KB). Therefore, we do not study the sensitivity to record or row sizes.

VII. Conclusion

This paper matched Big Data machine learning analytics (BMLA) applications with die-stacking via processing-near-memory (PNM). BMLAs are: (a) *irregular-and-compute-light* (i.e., perform only a few operations per input word which include data-dependent branches and indirect memory accesses); (b) *compact* (i.e., the relevant portion of the input data and the intermediate live data for each thread are small); and (c) *memory-row-dense* (i.e., process the input data without skipping over many bytes). These characteristics are not all shared by traditional database analytics, and except for irregularity, are necessary for bandwidth- and energy-efficient PNM, irrespective of the architecture.

Based on these characteristics, we proposed memory optimizations for a "sea of simple MIMD cores (SSMC)" PNM architecture, called *Millipede*, which exploits BMLAs' row-density by (pre)fetching and operating on entire memory rows. Instead of this deliberate *row-oriented* access and compute-schedule, conventional multicores opportunistically improve row locality while fetching and operating on cache blocks. Millipede handles BMLAs' irregularity and memory latency by employing MIMD execution and sequential prefetch of input data. However, because Millipede's MIMD corelets may stray far from each other, a leading corelet may prematurely evict the prefetched data before consumption by lagging corelets. Millipede employs *cross-corelet flow-control* to prevent such eviction. Millipede further exploits this flow control for frequency scaling based on *coarse-grain compute-memory rate-matching*. Using simulations, we compared PNM architectures to show that Millipede improves performance and energy, by 135% and 27% (198% energy-delay) over a GPGPU with prefetch, and by 35% and 36% (84% energy-delay) over SSMC with prefetch, when all three architectures use the same resources (i.e., number of cores and on-processor-die memory) and identical die-stacking. Millipede achieves these improvements by adding simple full-empty bits, counters, and incrementers to an SSMC-based PNM skeleton. As such, its performance, energy, and simplicity make Millipede an attractive PNM architecture for BMLAs.

VIII. Acknowledgments

This material is based upon work supported in part by SK Hynix Inc. and the National Science Foundation (under Grant No. 1633412 and Grant No. 1405939).

References

[1] J. T. Pawlowski, "Hybrid memory cube (HMC)," in *2011 IEEE Hot Chips 23 Symposium (HCS)*, Aug 2011, pp. 1–24.

[2] J. Kim and K. Tran, "HBM: Memory solution for bandwidth-hungry processors." Presented at 'Hot Chips: A Symposium on High Performance Chips', 2014.

[3] R. Bordawekar *et al.*, "Analyzing analytics," *SIGMOD Rec.*, vol. 42, no. 4, pp. 17–28, Feb. 2014.

[4] h2o.ai, "H2o.ai," http://www.h2o.ai/h2o/machine-learning/, accessed: 10-14-2017.

[5] H. S. Stone, "A logic-in-memory computer," *Computers, IEEE Transactions on*, vol. C-19, no. 1, pp. 73–78, Jan 1970.

[6] D. Patterson *et al.*, "A case for intelligent ram," *IEEE Micro*, vol. 17, no. 2, pp. 34–44, Mar. 1997.

[7] J. B. Brockman *et al.*, "A low cost, multithreaded processing-in-memory system," in *WMPI '04*, 2004.

[8] R. Nair *et al.*, "Active memory cube: A processing-in-memory architecture for exascale systems," *IBM Journal of Research and Development*, vol. 59, no. 2/3, pp. 17:1–17:14, March 2015.

[9] M. Hall *et al.*, "Mapping irregular applications to DIVA, a PIM-based data-intensive architecture," in *SC '99*, 1999.

[10] S. H. Pugsley *et al.*, "NDC: analyzing the impact of 3d-stacked memory+logic devices on mapreduce workloads," in *ISPASS '14*, 2014, pp. 190–200.

[11] M. Gao *et al.*, "Practical near-data processing for in-memory analytics frameworks," in *PACT '15*. IEEE, 2015, pp. 113–124.

[12] J. Ahn *et al.*, "A scalable processing-in-memory accelerator for parallel graph processing," in *ISCA '15*, 2015, pp. 105–117.

[13] A. Farmahini-Farahani *et al.*, "NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules," in *HPCA '15*, Feb 2015, pp. 283–295.

[14] M. Gokhale *et al.*, "Processing in memory: The terasys massively parallel pim array," *Computer*, vol. 28, no. 4, pp. 23–31, Apr. 1995.

[15] M. Drumond *et al.*, "The mondrian data engine," in *ISCA '17*, 2017, pp. 639–651.

[16] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI 2004*, 2004, pp. 137–150.

[17] M. Zaharia *et al.*, "Spark: Cluster computing with working sets," in *HotCloud '10*, 2010, pp. 10–10.

[18] N. Mirzadeh *et al.*, "Sort vs. Hash Join Revisited for Near-Memory Execution," in *ASBD '15*, 2015.

[19] M. Kamruzzaman *et al.*, "Inter-core prefetching for multicore processors using migrating helper threads," in *ASPLOS '11*, 2011, pp. 393–404.

[20] C. Kaynak *et al.*, "Shift: Shared history instruction fetch for lean-core server processors," in *MICRO '13*, 2013, pp. 272–283.

[21] E. Ebrahimi *et al.*, "Coordinated control of multiple prefetchers in multi-core systems," in *MICRO '09*, 2009, pp. 316–326.

[22] Q. Wu *et al.*, "Voltage and frequency control with adaptive reaction time in multiple-clock-domain processors," in *HPCA '05*, Feb 2005, pp. 178–189.

[23] A. Iyer and D. Marculescu, "Power efficiency of voltage scaling in multiple clock multiple voltage cores," in *ICCAD '02*, Nov 2002, pp. 379–386.

[24] T. W. Bartenstein and Y. D. Liu, "Green streams for data-intensive software," in *ICSE '13*, 2013, pp. 532–541.

[25] H. Hoffmann *et al.*, "Dynamic knobs for responsive power-aware computing," in *ASPLOS '11*. New York, NY, USA: ACM, 2011, pp. 199–212.

[26] B. Y. Cho *et al.*, "XSD: Accelerating MapReduce by Harnessing GPU inside SSD," in *WoNDP 2013 with MICRO '13*, 2013.

[27] R. G. Dreslinski *et al.*, "Centip3de: A 64-core, 3d stacked near-threshold system," *IEEE Micro*, vol. 33, no. 2, pp. 8–16, March 2013.

[28] T. G. Rogers *et al.*, "Cache-conscious wavefront scheduling," in *MICRO '12*, 2012, pp. 72–83.

[29] R. Balasubramonian *et al.*, "Near-data processing: Insights from a MICRO-46 workshop," *IEEE Micro*, vol. 34, no. 4, pp. 36–42, July 2014.

[30] Q. Guo *et al.*, "AC-DIMM: Associative computing with STT-MRAM," in *ISCA '13*. New York, NY, USA: ACM, pp. 189–200.

[31] B. Akin *et al.*, "Data reorganization in memory using 3D-stacked DRAM," in *ISCA '15*, 2015, pp. 131–143.

[32] J. Ahn *et al.*, "PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," in *ISCA '15*, June 2015, pp. 336–348.

[33] R. St. Amant *et al.*, "General-purpose code acceleration with limited-precision analog computation," in *ISCA '14*, 2014, pp. 505–516.

[34] Z. Du *et al.*, "Shidiannao: Shifting vision processing closer to the sensor," in *ISCA '15*, 2015, pp. 92–104.

[35] P. A. Merolla *et al.*, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, no. 6197, pp. 668–673, 2014.

[36] D. E. Shaw *et al.*, "Anton, a special-purpose machine for molecular dynamics simulation," in *ISCA '07*, 2007, pp. 1–12.

[37] J. P. Grossman *et al.*, "Hardware support for fine-grained event-driven computation in anton 2," in *ASPLOS '13*, 2013, pp. 549–560.

[38] J. Lee *et al.*, "Many-thread aware prefetching mechanisms for gpgpu applications," in *MICRO '10*, 2010, pp. 213–224.

[39] Y. Yetim *et al.*, "EPROF: An energy/performance/reliability optimization framework for streaming applications," in *ASP-DAC '12*, Jan 2012, pp. 769–774.

[40] C. Chu *et al.*, "Map-reduce for machine learning on multicore," in *NIPS '06*, 2006, pp. 281–288.

[41] T. G. Rogers *et al.*, "A variable warp size architecture," in *ISCA '15*, 2015, pp. 489–501.

[42] T. Chen *et al.*, "Cell broadband engine architecture and its first implementation: A performance view," *IBM J. Res. Dev.*, vol. 51, no. 5, pp. 559–572, Sep. 2007.

[43] J. Leng *et al.*, "Gpuwattch: Enabling energy optimizations in gpgpus," *SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 487–498, Jun. 2013.

[44] K. T. Malladi *et al.*, "Towards energy-proportional datacenter memory with mobile DRAM," in *ISCA '12*, 2012, pp. 37–48.