10-17-2016

RowCore: A Processing-Near-Memory Architecture for Big Data Machine Learning

Nitin .
Purdue University, nnitin@ecn.purdue.edu

Mithuna Thottethodi
School of Electrical and Computer Engineering, Purdue University, mithuna@purdue.edu

T.N. Vijaykumar
Purdue University, vijay@ecn.purdue.edu

RowCore: A Processing-Near-Memory Architecture for Big Data Machine Learning

Nitin, Mithuna Thottethodi, and T. N. Vijaykumar
School of Electrical and Computer Engineering
Purdue University, West Lafayette, IN
{nnitin,mithuna,vijay}@ecn.purdue.edu

ABSTRACT

The technology-push of die stacking and application-pull of Big Data machine learning (BDML) 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 workload characteristics: (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 computation has a small intermediate live data and uses only a small amount of contiguous input data); and (c) memory-row-dense (i.e., process the input data without skipping over many bytes). We show that BDMLs have or can be transformed to have these characteristics which, except for irregularity, are necessary for bandwidth- and energy-efficient PNM, irrespective of the architecture. (2) Based on these characteristics, we propose RowCore, a row-oriented PNM architecture, which (pre)fetches and operates on entire memory rows to exploit BDMLs' row-density. Instead of this row-centric access and compute-schedule, traditional architectures opportunistically improve row locality while fetching and operating on cache blocks. (3) RowCore employs well-known MIMD execution to handle BDMLs' irregularity, and sequential prefetch of input data to hide memory latency. In RowCore, 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. RowCore employs novel cross-corelet flow-control to prevent such eviction. (4) RowCore further exploits its flow-controlled prefetch for frequency scaling based on novel coarse-grain compute-memory rate-matching which decreases (increases) the processor clock speed when the prefetch buffers are empty (full). Using simulations, we show that RowCore improves performance and energy, by 135% and 20% over a GPGPU with prefetch, and by 35% and 34% over a multicore with prefetch, when all three architectures use the same resources (i.e., number of cores, and on-processor-die memory) and identical die-stacking (i.e., GPGPUs/multicores/RowCore and DRAM).

1. 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]) is emerging as a way to achieve unprecedented high-bandwidth connection between memory and processor dies using thousands of fast, through-silicon vias (TSVs) instead of a few hundreds of slower, traditional pins. At the same time, prevalent Big Data machine learning (BDML) applications process vast amounts of data, are abundantly parallel, and require massive memory bandwidths (e.g., naive bayes, k-means, and principal component analysis). Thus, it is time for PNM architectures to match up die-stacking with BDMLs.

While processing-in-memory (PIM) has been around for decades [3, 4, 5, 6, 7, 8, 9, 10, 11, 12], there are three problems. The first is the mismatch between DRAM and logic processes. While some past proposals have addressed this mismatch by advocating PIM with SRAM [13], 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 BDMLs. The final problem is the lack of workloads with the right characteristics which BDMLs have.

We identify three key characteristics, which BDMLs either have naturally or can be transformed to have, that fit PNM: *irregular-and-light-compute*, *compact*, and *memory-row-dense*. First, BDMLs often perform data-dependent computation to differentiate among data which is fundamental to learning. Such computation involves data-dependent branches and/or irregular memory access to intermediate program state but not input data (e.g., counter[label] for 100 randomly-occurring labels). Further, BDMLs perform only a few operations per input data word (e.g., under 10) implying that the compute bandwidth needed is small, and simple, energy-efficient pipelines suffice. Conversely, compute-heavy applications would be compute-bound and not benefit much from PNM's bandwidth advantage, irrespective of the architecture. Second, BDMLs perform an acute data reduction via summarization so that the output is much smaller than the input (e.g., clustering data into a few clusters). Consequently, BDMLs 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 ensures avoids PIM's second problem above. the input data in and process almost all the data without skipping over sub-streams or leaving gaps in the input, 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 [11, 9]) does not necessarily imply the lack of gaps which is key for bandwidth efficiency. Except for irregularity, these characteristics are *necessary* for bandwidth- and energy-efficient PNM, *irrespective of the architecture* (see Section 3.4).

Following widespread practice for programmability reasons, we use MapReduce [14] (or Spark [15]) to implement BDMLs. While recent PNM work (e.g.,[9, 8]) considers general MapReduce workloads with a broad set of characteristics (see Section 2), our novelty is in identifying the specific characteristics that fit PNM, which is our first contribution.

Vector (SIMD), GPGPU (SIMT) and multicore, are well-known parallel architectures. However, BDMLs' irregularity makes SIMD and SIMT execution inefficient. While multicore's MIMD execution can handle irregularity, the cores stray far from each other in execution, interleave accesses to many memory rows, destroy row locality, and squander die-stacking bandwidth. Instead, we propose *RowCore*, a PNM architecture based on BDMLs' characteristics, with the key goal of utilizing the full bandwidth of die stacking while remaining energy-efficient. Before describing RowCore's novel features, we list its skeleton of well-known features: For one (or a few) memory array, there is a RowCore processor comprising a wide set of cores, called *corelets*, to exploit BDMLs' parallelism. RowCore employs MIMD execution for BDMLs' irregularity. BDMLs' light-compute nature affords simple corelets which employ small-scale hardware multithreading to tolerate pipeline hazards instead of complex branch prediction and register bypassing (e.g., 4 contexts). BDMLs' compact nature enables each corelet to employ a small register file and local memory. To hide memory latency of input data, RowCore employs sequential prefetch which exploits BDMLs' row-dense nature; RowCore does not employ hardware-managed, deep cache hierarchies.

While the above skeleton is not new, RowCore's novelty stems from being a *memory-row-oriented* architecture which (pre)fetches and operates 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 exploits BDMLs' row density to achieve RowCore's goal of utilizing the full die-stacking bandwidth. Such deliberate access-schedule coupling differs from best-effort row locality in traditional architectures which fetch and operate on cache blocks. Recent PNM work does not target row-orientedness, except for considering row locality in *Joins* [16] which are fundamentally not compact as we discuss in Section 3.4. This row-orientedness is our second contribution.

RowCore's MIMD execution also incurs the above-mentioned straying problem. Because one corelet prefetches for all the other corelets in a RowCore 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. RowCore employs *cross-corelet flow control* 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 [17, 18, 19]. Addressing accuracy and timeliness are easy in BDMLs 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. This flow-controlled cross-corelet row prefetching is our third contribution.

Finally. BDMLs, being compute-light, are likely to be memory-bandwidth bound whose energy can be reduced. To that end, we leverage the prefetch flow control to rate-match the RowCore processor and die-stacked memory via frequency scaling (or voltage-frequency scaling, if possible). 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 granularities, 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, BDML behavior does not change during execution. 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 and not smaller code sections. This compute-memory rate-matching is our fourth contribution. While rate-matching is well-known. our novelty is coarse-grained compute-memory rate-matching. Previous work explores rate-matching in hardware at the fine granularity of pipeline sub-components (space) and program phass (time) [20, 21], statically in the compiler [22], or by trading off accuracy [23].

To summarize, the key contributions of this paper are:

- identifying irregular-and-light-compute, compact, and row-dense as key workload characteristics that fit PNM;
- a row-oriented microarchitecture;
- flow-controlled cross-corelet row prefetching; and
- coarse-grain compute-memory rate-matching.

Using software simulations running BDMLs, we show that RowCore improves performance and energy by 135% and 20% over a comparable GPGPU with prefetch, and by 35% and 34% over a comparable multicore with prefetch, when all three architectures use the same resources (i.e., number of cores and on-processor-die memory) and identical die-stacking (i.e., GPGPUs/multicores/RowCore and DRAM).

The rest of the paper is organized as follows. We contrast RowCore to related work in Section 2. Section 3 discusses workload characteristics. Section 4 describes RowCore's microarchitecture. Section 5 describes our evaluation methodology. In Section 6, we present our experimental results. Finally, we conclude in Section 7.

2. RELATED WORK

We discuss previous work related to our key contributions.

Workload characteristics: As discussed in Section 1, previous work explores general workloads, including MapReduce, which include applications both with and without inter-thread communication or row locality. In contrast, we identify the key characteristics that fit PNM. Specifically, previous work labels workloads with inter-thread communication and without row locality as irregular [9]. In fact, our characterization excludes such workloads because the former may imply lack of parallelism, the latter would imply lack of memory bandwidth efficiency (see Section 3.4). Instead, our irregularity stems from data-dependent branches and indirect memory accesses to the intermediate state in local memory, not the input data which is row-dense.

Row-orientedness: Past PNM architecture work has explored vectors [4], VLIW [6], and GPGPUs [24] which are neither row-oriented (i.e., row-centric access and compute-schedule) nor MIMD, uniprocessors [5] and multicores [4, 7, 8, 9, 25], which are MIMD but not row-oriented, for near data processing. 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 due to unpredictable memory accesses. Unfortunately, the interleaving of numerous contexts destroys cache locality [26] and row locality. While GPGPU's multithreading degree can be turned down (e.g., assign only a few Warps to an SM) and supplemented with prefetching for the predictable BDMLs, even 100%-accurate cache-block prefetching does not address GPGPU's difficulty with irregularity. In multicores, the unavoidable variability in the cores' record-processing work causes the MIMD cores to stray from each other (similar to RowCore corelets without flow-controlled prefetch), interleaving accesses to different rows and destroying row locality. Here again, 100%-accurate cache-block prefetching does not address multicores' poor row locality. DIVA [7] targets irregular applications by supporting address translation and coherence but does not address row-orientedness. Centip3de [25] exploits die-stacking using a multicore architecture without any row-orientedness. None of the NDP workshop 2014-2015 papers [27] address row-orientedness.

There are a number of recently-proposed PNM architectures for various computational patterns. For example, NDA [11] maps dataflow programs to a coarse-grain reconfigurable architecture (CGRA) nodes that communicate over a network. RowCore targets BDMLs with abundant data parallelism for which general dataflow over communication networks is overkill. Further, though NDA's CGRAs can work in MIMD fashion, NDA does not have row-orientedness to maximize die-stacked bandwidth utilization. AC-DIMM [28], based on STT-MRAM, combines ternary associative search with PNM by co-locating key-value pairs in the TCAM. RowCore (a) explicitly stripes records to capture inter-record parallelism in the same DRAM row whereas AC-DIMM co-locates an entire record/tuple in the same TCAM row, and (b) exploits row-orientedness instead of just co-locating key-value pairs. While data reorganization acceleration for 3D-stacked memory [29] is orthogonal to our work, RowCore can leverage this work for our interleaved (struct of arrays of structs) layout (see Section 3.2). While Tesseract [10] targets graph-analysis workloads via MIMD processing and inter-core communication, such workloads are not row-dense or compact, and Tesseract is not row-oriented and would incur row-locality problems similar to multicores. Other work [30] develops PNM-enabled instructions which are offloaded to PNM cores if the data is not present in the caches. For our datasets which are much larger than caches, we expect that PNM accelera-

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.
```

Table 1: Walk-through example of Naive Bayes

tion is unavoidable.

Architectures in other compute-intensive domains are specialized for their specific purpose (e.g.,. [31, 32, 33, 34, 35]). Unlike these compute-intensive architectures, RowCore is a data-intensive architecture.

Flow-controlled cross-corelet row prefetching: While prefetching is well-studied in terms of accuracy and timeliness, we focus on cross-core prefetching where one core prefetches for others, as do our corelets, using helper threads [17], in GPUs [36], and in multicores [18, 19]. All but the last paper focus on accuracy or timeliness via helper threads or sharing history [18] whereas our concern is cross-core coordination to avoid premature eviction of prefetched data. The last paper regulates each core's prefetching into a shared LLC to ensure that cores do not overprefetch and hog the cache capacity. In contrast, RowCore's flow control ensures cross-core use of the prefetched data and not equitable sharing of the cache capacity.

Coarse-grain compute-memory rate-matching: RowCore achieves dynamic compute-memory rate-matching in hardware. While rate-matching is well-known, our novelty is 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 at the fine granularities of pipeline sub-component clock domains and program phases [20, 21]. These papers attempt to rate-match pipeline sub-components running typical sequential programs with high variability in instruction-level parallelism. Other work employs the compiler and profiling for static, compute-compute rate-matching in streaming applications [22]. Finally, work on compute-pin-I/O rate-matching for multimedia workloads (e.g., h.264) trades-off accuracy for energy by using application-level hints [23] or heterogeneous cores with varying power

/performance/reliability characteristics [37]. In contrast, RowCore achieves energy savings without diluting the accuracy.

3. SOFTWARE

BDMLs are written commonly as MapReductions where the Map tasks are completely independent of each other and the Reduce tasks are often commutative and associative, at least partially, so that they can be run concurrently [38].

3.1 MapReduce programming model

BDML 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 is local to each Map task and reduces only the records processed by a Map task (i.e., the reduction is not across multiple Map task outputs). In some cases due to local memory limitation, the intermediate state is partially-reduced across a subset of Map tasks which are still local to a corelet. Traditional MapReduce's Map tasks perform only Mapping optionally followed by some combining which is a form of within-Map-task partial Reduce. The final Reduce (with Shuffle) computes the final result by reducing the partially-reduced outputs across all Map tasks.

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 BDMLs' compute irregular-and-light. Only the input data, and not any other computed data, is large in BDMLs. The input data access is naturally, or as we show below can be made to be, row-dense and compact. Recall that general spatial locality implies spatially-nearby accesses but not the lack of gaps which is key for bandwidth efficiency. As discussed in Section 1, BDMLs 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 from memory (i.e., the intermediate Map output is written to the disk before being shuffled to the reduce tasks). BDMLs' intermediate state includes any constant data and each Map's partially-reduced output accumulated at any point in execution. BDMLs' being compute-light further helps in keeping the intermediate state small. The final Reduce combines the small per-Map task partially-reduced output to produce the final output.

Table 1 shows the memory organization, the local state and the map/reduce operations needed for *Naive Bayes*, a prominent BDML kernel (despite being named "naive"). The example in Table 1 assumes a large collection of n-dimensional records with an additional year field. Each record is logically in one of two classes depending whether the year exceeds the threshold. The key computation is the counting of conditional probabilities depending on the class (based on year) of each record. The computation makes row-dense and compact accesses to each record's coordinates in each each dimension and year of the record (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 results in irregularity because of (1) the branch to identify the subset of records of interest and (2) the indirect data-dependent access of the conditional probability matrix. (Alternately, the indirect access code could be rewritten with if-then-else constructs to increment the counters for the appropriate class; but that would lead to more control-flow irregularity.) Accumulating the counts into the 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 (similar to [8, 11]).

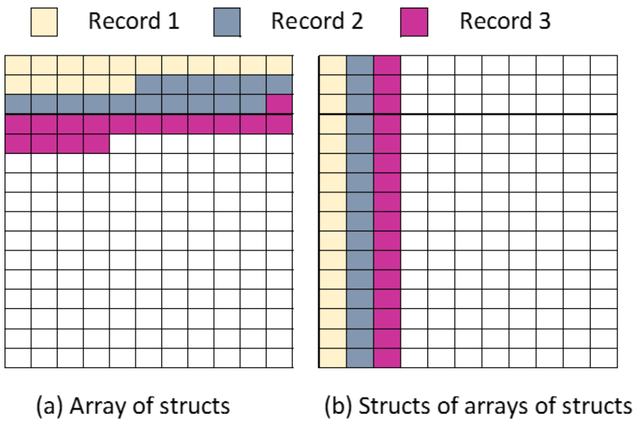

Figure 1: Interleaved layout

3.2 Layout issues

One key observation for BDML is that the parallelism is primarily inter-record. Unfortunately, a "row-major"-like or an "array of structs" layout in memory, as shown in Figure 1(a), cannot efficiently capture inter-record parallelism. Because consecutive records would fall in possibly different rows of memory as shown in Figure 1(a), accessing the records in parallel would destroy row locality. This layout issue is common to all the workloads and architectures.

One way to achieve better row locality would be a "column major"-like *interleaved* "array of structs of arrays" layout, as shown in Figure 1(b), where each record is striped over several rows and the same field of consecutive records fall in the same row. This interleaved layout does not have the alignment problem because each row holds as many words (from distinct records) as would fit, and as such is preferred. However, because there are likely more words in a row than there are cores, each core would have to process more than one record. Fortunately, the live state of the records processed by a core can be partially reduced to prevent increase in each core's live state. Nevertheless, because this layout implies that a core (a lane, or a corelet) processes an entire record, the full live state needs to fit in the core's resources. Fortunately, this state is small enough for most common BDMLs to fit in 4-8 KB of local memory. As such, our evaluation uses this interleaved layout for all the three architectures we compare – GPGPUs, multicores, and RowCore.

3.3 Workloads

Table 2 summarizes some of the BDMLs we consider. We show that these BDMLs are irregular-and-compute-light,

Workload	Input record	Per-node live state	Ops per byte
Random Sample	Movie rating tuple	Rating counts per bin + total elements per bin	O(1)
Count	Movie rating tuple	Bin Count	O(1)
Variance	Movie rating tuple	Bin Count + Bin sum-of-squares	O(1)
Naive Bayes (NB)	N-dimensional point + Bin-id	Conditional probabilities for each bin	O(1)
Kmeans (1-iteration)	N-dimensional point	N-dimensional centroids	O(1) for new centroid computation, O(k) for nearest centroid
Classify	N-dimensional point	N-dimensional centroids	O(k) for nearest centroid
Principal Components Analysis (PCA)	N-dimensional point	Mean and covariance	O(1) for mean, O(N) for covariance
Gaussian Determinant Analysis (GDA)	N-dimensional point + Bin-id	Bin-specific means/covariance	O(1) for mean, O(N) for covariance

Table 2: Summary of workload behavior

compact, and row-dense. while all the workloads are light (i.e., no super-linear compute complexity), two of them have relatively more compute than the others. Recall our stipulation that the workloads are naturally or, with some modifications, can be made to be compact and row-dense. Some of these BDMLs are naturally compact and row-dense. For example, the computation for *kmeans* involves computing the distance from each datapoint in a multi-dimensional space to a set of centroids. Because each datapoint is saved as a set of coordinates, the computation is inherently compact. Because every coordinate is used in the computation of the distance, the computation is dense. Note that the centroids are part of the live state that persists across datapoints. As such, they do not perturb the density or the compactness of the computation. However, the distance computation from each of the k centroids may require proportional effort (i.e., O(k)).

Other BDMLs, with appropriate data layout, can be made compact. For example, BDMLs like *NB* and *GDA* (Table 2) typically process a training set that includes: (1) co-ordinates of each datapoint in a multidimensional space, and (2) the bin/class to which the datapoint belongs. There are two ways to organize such training-set data. One way is to maintain two separate arrays for data-points and for the classification. However, such an organization leads to non-compact accesses because the computation accesses the datapoint with its corresponding classification. Instead, an array of structs organization, 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 *SB*) depending on the bin to which each data-point belongs. *PCA*, which computes the mean and the covariance matrix as key steps of the kernel, is inherently row-dense and compact.

The above workloads cover many important BDML subdomains. Further, because RowCore is targeted toward inter-record parallelism — a specific form of data parallelism at the granularity of medium-granularity data records (e.g., 100-dimensional record) which is common in BDMLs, we anticipate that either many of the as-yet unanalyzed BDML workloads already fit RowCore or can be transformed to fit. In future work, we will expand the set of our BDMLs.

3.4 Implications of workload 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 making PNM's bandwidth advantage less relevant. (3) Not compact (naturally nor through transformation) would mean that some data other than the input is large (the old PIM problem, as discussed in Section 1). Because the processor can be near only one large data (the input), the other data would have to use, and be bottlenecked by, traditional, non-die-stacked channels and networks, again, making PNM's bandwidth advantage less relevant. Consider an example where a computation uses another piece of large data, with the input present in the arrays stacked above the processor and the other data stacked above (an)other processor (or worse, the other data is off-package on another stack). Then the second data has to be moved through an on-PNM-die network to the first processor. Assuming all the processors are active in parallel, there is data movement to all the processors (and numerous corelets) in parallel. If the second data is needed at a rate similar to the high die-stacked rate of the first, then the network needs to support such high rates for each of the processors (i.e., a multiplicative effect). While such a network is likely to be expensive in energy and area, a lower-bandwidth network would become a bottleneck, forcing PNM's bandwidth to be underutilized for the first data. 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 [9] but not the implication of compactness. A non-compact example is a *join* operation on unstructured, unindexed data (as is common in BDMLs as opposed to databases), which requires pairwise comparisons of all the records in two large tables. Such joins 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 [16].) While poor row locality would mean

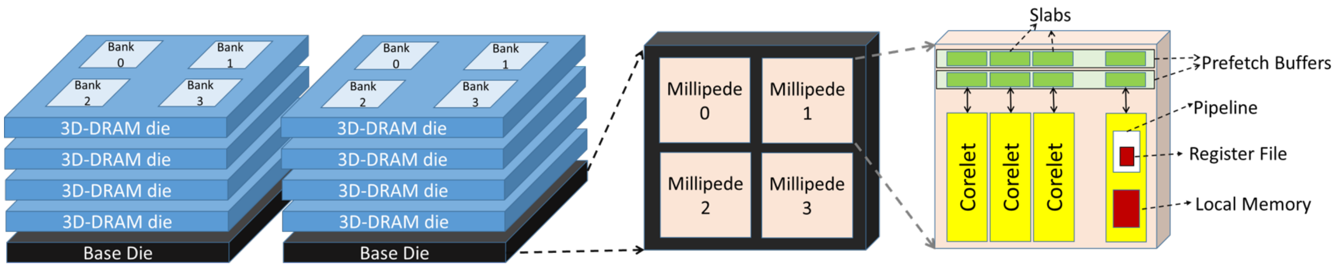

Figure 2: RowCore Architecture

poor performance [16], our point is that even with good row locality, *joins* are not compact and therefore would underutilize PNM's bandwidth and perform well below PNM's full potential. (4) Not row-dense (naturally nor through transformation) would mean memory bandwidth inefficiency and degrading 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 workloads that violate these characteristics, this limitation is not specific to RowCore but applies to any PNM architecture. On the positive side, we have identified BDMLs, which are used prevalently in the real world today. to fit within PNM's constraints. PNM not working for some other workloads does not diminish the fact that PNM works for the important BDMLs. This limitation is similar to how GPUs perform poorly for all but the most regular workloads but are still prevalent. Further, the idea of an architecture is to target a specific but important workload, and not a wide spectrum of workloads, for energy reasons. Our point is that RowCore can exploit the full die-stacking bandwidth in an energy-efficient manner for BDMLs.

3.5 Workload/Architecture match

We briefly consider mapping Naive Bayes to a GPGPU, multicore, and RowCore. 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 can be prefetched in cache blocks from the die-stacked DRAM to the L1 D-cache. In a multicore, both the live state and the input data are placed in the L1 D-cache to which both cache-block prefetches and demand accesses of the input data occur. In RowCore, 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 multicores incur problems which RowCore solves. As discussed in Section 2, GPGPUs SIMT incurs performance loss due to data-dependent branches (the indirect memory accesses to the live state can be handled by GPGPU's Shared Memory). While multicores' MIMD can avoid these SIMT penalties, the variability in per-record work across cores causes the cores to stray far from each other, interleave accesses to multiple rows, and destroy row locality (as explained in Section 2). Synchronizing the cores at each record would push multicore execution closer to SIMT and its overheads. As discussed in Section 2, even 100%-accurate cache-block prefetches do not address these problems of GPGPUs and multicores (row locality is a bandwidth problem whereas prefetching improves latency but not bandwidth). To avoid the SIMT problems, RowCore also employs MIMD. Unlike multicores, however, the row-oriented RowCore prefetches entire rows and employs flow-control to limit the corelets' straying from each other and avoid premature eviction of prefetched data. Thus, RowCore utilizes the full die-stacking bandwidth while enjoying MIMD's benefits.

Nevertheless, two key reasons for GPGPUs to use SIMT instead of 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. In general, MIMD execution loses these benefits. Nevertheless, the relative prevalence of branch and memory irregularity in BDMLs impedes SIMT execution and renders wide accesses ineffective. As such, we carefully model the energy differences between SIMT and MIMD in our results.

4. ROWCORE

Recall from Section 1 that there is a RowCore processor for each memory array (or a few arrays), as shown in Figure 2. Each processor comprises a wide set of simple cores, called *corelets*, which employ some well-known ideas (listed in Section 4.1). The key novel ideas are: row-oriented microarchitecture, flow-controlled, cross-corelet, row prefetching, and coarse-grained compute-memory rate-matching.

4.1 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), RowCore employs MIMD execution among the corelets each of which has its own instruction cache. Because the code size of BDMLs is small, we broadcast the code once at the beginning of execution (our BDMLs fit under 1 KB). Because the workloads are compact, each corelet has a small register file and local memory.

Because the workloads are compute-light, the corelets' pipeline is simple and energy-efficient. Accordingly, we avoid complex register bypassing and branch prediction. Instead, we employ small-scale hardware multithreading to tolerate pipeline hazards like GPGPUs (e.g., 4 contexts). Because memory latency is hidden by prefetching, the live state is held in fast local memory, and the pipeline is shallow, the pipeline hazards are short. Consequently, small-scale multithreading suffices. The hardware uses a simple round-robin policy to schedule the 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. GPGPUs and multicores can also employ such small-scale hardware multithreading to tolerate their pipeline hazards. In our evaluation, we assume so.

4.2 Row-orientedness

Millepede processors fetch and operate on entire rows before moving on to the next row (i.e., row-centric access and compute-schedule). RowCore employs simple row prefetching to exploit die-stacking's full bandwidth for the memory-row-dense BDMLs. 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 3.2 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 main CPU runs Final Reduce to combine this state from all the processors, as discussed later in Section 4.4.

4.3 Flow-controlled cross-corelet prefetch

The next row prefetch occurs before the current row processing starts. This simple prefetch could be in hardware but because the processing of a record is easily identifiable in MapReduce, this simple prefetch could also be in software. 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). However, such prefetching faces a problem. Because all the corelets execute the same Map code, there may be redundant prefetches to the same row from all the corelets. while redundant prefetches from one core can be avoided easily (by checking the MSHRs), preventing redundant prefetches from multiple corelets to the memory controller is not easy without extra coordination among the corelets. Avoiding this redundancy in the code is hard because any of the corelets can be ahead of the others due to MIMD execution. Instead, we employ a hardware scheme which sets a *prefetch-trigger bit* in the prefetch buffer entry upon a prefetch fill; each entry holds an entire row. The scheme clears the bit when the first demand fetch (Map access) from any of the corelets sees a set bit and issues the next prefetch. Later demand fetches see a clear bit and do not issue any prefetch. The bit indicates only whether the next prefetch has been issued, and not whether the buffer has been consumed fully and can be re-allocated; that detail comes later. While previous hardware prefetchers [39] use bits to indicate whether the prefetch is useful (i.e., prefetched data has been accessed) and further prefetches should be done, our scheme uses the prefetch-trigger bit to prevent redundant prefetches. Our hardware scheme can take hints from software about how far ahead to prefetch (usually only one

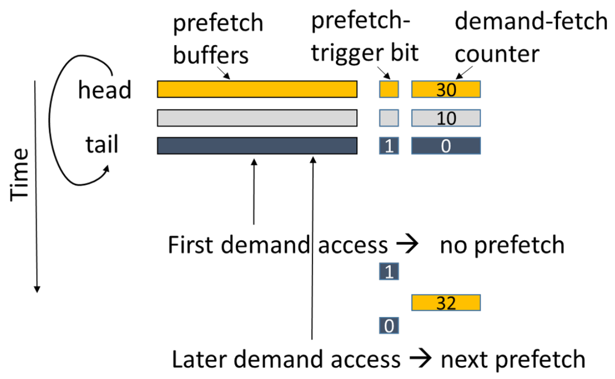

Figure 3: Flow control operation

row ahead).

A central issue is that the corelet's MIMD execution imposes a need for flow control in the prefetch buffer which is organized as a circular queue. Upon the first access to a buffer entry (matched by an address tag), the next entry is allocated for the accompanying prefetch. A leading corelet may surge past the other corelets and issue prefetches for all the buffer entries wrapping around to the head entry which has not been consumed fully yet. To prevent such premature re-allocation, RowCore provides a per-entry *demand-fetch counter* which is incremented upon demand fetch. An entry is re-allocated only after the counter saturates at the corelet count, indicating that the entry has been consumed fully. The counter is reset upon re-allocation and subsequent prefetch fill. Normally, the next entry has been consumed fully and can be re-allocated for the next prefetch (i.e., the circular queue is not full). When the queue is full, the counter of the next entry (the head entry) is unsaturated. Then, the leading corelet does not trigger a prefetch upon a demand fetch to the tail entry even if the tail entry's prefetch-trigger bit is set (see Figure 3). A later access to the head entry causes the entry's demand-acccess counter to saturate. The next demand fetch to the tail entry issues the next prefetch and clears the prefetch-trigger bit (see Figure 3). Because BDMLs access rows sequentially, the head entry's counter is guaranteed to saturate before the last demand fetch to the tail entry, ensuring that the next prefetch ia not missed. Because BDMLs are compute-light, the amount of variability in the corelets' execution times can be absorbed by a modest number of entries (e.g., 16 2-KB entries).

An important implementation consideration is that DRAM interface is subject to JEDEC standards. The row prefetches can comply with JEDEC in that the entire row need not be transferred in one super-wide access (e.g., 2 KB) which may be disallowed by the standards. Instead, the transfer can occur in JEDEC-allowed units (e.g., 128 B). As long as the raw DRAM internal bandwidth, the interface bandwidth and the RowCore processor's compute-rate match, there is no bandwidth loss. The interface bandwidth being lower than the DRAM bandwidth implies a JEDEC limitation (highly unlikely, as the whole point of die stacking is to deliver the internal bandwidth at the interface). The compute rate being lower than the DRAM bandwidth implies that the application is compute-bound and the slack in the memory bandwidth can be consumed based on area and power considerations (more or faster corelets). Conversely, the DRAM bandwidth being lower means a memory-bound application. A key point is that because the JEDEC-allowed transfer units for a memory row arrive at the prefetch buffer in a staggered manner, a leading corelet may find its slab to be present even though the rest of the units have not arrived yet. The corelet is free to proceed without waiting for the other corelets' units to arrive as would be the case with SIMT (which essentially imposes a barrier at every access and exposes the transfer latency). Thus, the prefetch needs to hide only the row access latency under the compute, and not the transfer latency. Without this concession, either a longer prefetch lookahead and more prefetch buffers would be needed, or the compute, being light, is likely to expose some or all of the transfer latency.

Because of the corelets' MIMD execution, the corelets access the prefetch buffers at different times. To provide 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 prefetch buffer access goes only to a slab-wide slice of the prefetch buffer entries, which is quite small (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 2). In our interleaved layout, a slab contains either words each from a contiguous set of records (word-interleaving), or contiguous words of a record (e.g., for a record's field larger than a word) where the next slab contains the next record and so on (slab-interleaving). Thus, each corelet can flexibly process multiple records or one record. while GPGPUs are forced to use word-size columns to achieve coalesceable accesses (wider columns would mean the lanes' accesses span multiple cache blocks), RowCore can use wider columns for layout flexibility.

The full-row prefetch and the flow control are fundamental for RowCore 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 above-mentioned per-buffer bits and counters would exist, but without the accompanying buffers, and function as before. However, such an implementation would need address tags in the local memory to detect presence of prefetch data whereas the above implementation needs tags only in the prefetch buffers.

4.4 Final Reduce

Because the final Reduce is much smaller than the Map and partial Reduce but also requires data from all the processors, providing support for communication among all the processors may not be worth it (e.g., Map and partial Reduce of billions of records versus final Reduce of 32 RowCore processors' partial reductions in their local memories). As such, this step can be done by the main CPU, as observed in [8, 11].

4.5 Memory Interface

RowCore assumes a discrete GPU-like memory interface where the main 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 main CPU's physical memory address space. The corelets do not provide support for virtual memory or coherence, which we leave for future work.

4.6 Coarse-grain compute-memory rate-matching

BDMLs, being compute-light, are likely to be memory-bandwidth-bound. Being memory-bound implies that cores idle when waiting for memory; the goal of our rate-matching is to eliminate such idling energy. To that end, we leverage the prefetch buffer flow control to rate-match the RowCore processor and die-stacked DRAM via dynamic frequency scaling (DFS). In our evaluation, we conservatively assume that further voltage scaling (i.e., DVFS) is not possible. If possible, our energy savings can be higher. While the corelets may diverge from each other at fine time granularities, they perform statistically similar amount of work over the full BDML execution (e.g., 10 billion records). Further, because the same computation is repeated for billions of records, BDML 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 hill-climbing control algorithm that decreases the processor clock speed in small steps (e.g., 5%) via frequency scaling whenever a leading corelet (defined in Section 4.3) finds the prefetch buffers to be empty, signifying a DRAM-bandwidth-bound application. The processor clock controls all the processor's corelets. Similarly, the algorithm increases the clock speed whenever a leading corelet finds the buffers full, signifying a 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 during the entire application execution. For instance, a small 5% step, a large 4x required change in the clock speed, and 200 cycles of computation per DRAM row (typical for our applications which are compute-light) implies convergence in 16,000 cycles compared to a few billions of cycles of application execution time. Though this simple algorithm would fluctuate after convergence within a band of the size of the step, the step is small enough that any resulting inefficiency is acceptable.

5. METHODOLOGY

We modify GPGPUsim to implement RowCore. For comparison purposes, we also simulate a GPGPU and a multicore using GPGPUsim. To capture their MIMD execution, we simulate a multicore and a RowCore processor each as an SM with only one lane corresponding to a simple core or corelet. Our simulation assumes that each architecture (i.e., multicore, GPGPU, and RowCore) 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 the three systems. All three systems use the interleaved layout (Section 3.2). While RowCore uses sequential row prefetch, the GPGPU and multicore use sequential cache-block prefetch. **Thus, our results isolate the benefits**

#RowCore processors # GPGPU SMs # multicores	1
Compute clock	700 MHz
# Corelets/lanes/cores per processor/SM/multicore	32
# Multithreading contexts per processor/SM/multicore	4
# Registers per corelet/lane/core	32
L1 I-cache per corelet/SM(not lane)/core	4 KB, 128B line,
Local memory per corelet	4 KB
Prefetch buffer per corelet	16 x 64B
L1 D-cache per SM	32 KB, 128B line
Shared memory per SM	128 KB, 4B interleaving
L1 D-cache per core	5 KB, 128B line
Die-stacked DRAM capacity	4 GB
# Die stack layers	4
# Memory channels	32
Channel clock	1.2 GHz
Channel width	128 bits
DRAM tCAS-tRP-tRCD-tRAS	9-9-9-27
Memory Controller	FR-FCFS (16 deep)
DRAM Access Energy	6pJ/bit [29]
Core Technology node	22nm

Table 3: Hardware Parameters

Workload	Record Size	Input Size	instrs per input word	Branch freq. (branches/inst.)	multicore row-misses per row
sample	8B	128MB	10	0.2	4.7
count	8B	128MB	7	0.14	8.0
variance	8B	128MB	12	0.08	11.2
nbayes	256B	128MB	14	0.11	11.0
kmeans	1024B	128MB	44	0.05	12.3
classify	1024B	128MB	40	0.05	12.6
pca	128B	128MB	150	0.02	15.9
gda	128B	128MB	180	0.015	15.9

Table 4: Benchmark parameters and characteristics

of RowCore's novel features while holding the effects of technology (CMOS 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 architectures.

The hardware parameters are shown in Table 3. We simulate a 32-corelet RowCore processor, a 32-core multicore, and a 32-lane GPGPU SM. All the three systems 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 multicore core has 5-KB L1-D (160 KB per multicore); and each GPGPU SM has 32-KB L1-D and 128-KB Shared Memory (total 160 KB per SM). Each RowCore corelet and each core in the multicore 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 RowCore and multicore in the energy estimates. The die-stacked DRAM's parameters, shown in Table 3, are typical [29]. The bandwidth is similar to HBM's specification of 128-bits per bank, with 1Gbps bandwidth per pin [2].

We implement the workloads in Table 2 in CUDA. Table 4 shows the record size, total input size, the branch frequency, and the number of row misses to each input data DRAM row in multicores (explained in Section 2). To achieve realistic simulation times, we limit the input data to 128MB and run the benchmarks to completion on one processor. The repetitive nature of record processing results in the workload behavior for a large-enough set of records being identical to that of any other. As such, the steady-state behavior (which was achieved well before 128 MB), will not change with larger datasets or more processors.

We use GPUWattch to estimate energy (parameters in Table 3). Recall from Section 3.5 that BDMLs 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 BDMLs' 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 3.5). However, such success is not often due to BDMLs' control-flow and memory irregularity (Section 3.3). We ensure that these benefits do not exist in RowCore and the multicore due to their MIMD execution. In multicores, the live data, input data cache-block prefetches and demand accesses all go to the L1 D-cache. In RowCore, the live state is in the per-corelet local memory, the input data is row-prefetched into the prefetch buffers (Section 4.3). Both the local memory and prefetch buffer are small and therefore dissipate low power. Each architecture's 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), DRAM energy dissipated in the stacked memory dies, and leakage energy of the logic die.

6. RESULTS

We start by comparing RowCore against GPGPUs and multicores 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 (workload characteristics is covered in Section 3.3). We then study RowCore's sensitivity to the system size and the number of prefetch buffers.

6.1 Performance

We compare a 32-corelet Millepede processor against a 32-lane GPGPU SM and a 32-core multicore, both with input data cache-block prefetch into their L1 D-caches. Recall from Section 5 that all the three systems use the same (a) number and type of cores, (b) amount of on-processor-die memory, and (c) interleaved layout. Further, all systems assume that each system places compute resources (i.e., cores, corelets, or GPGPU SMs) on a logic die with memory dies stacked above. Figure 4 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 num-

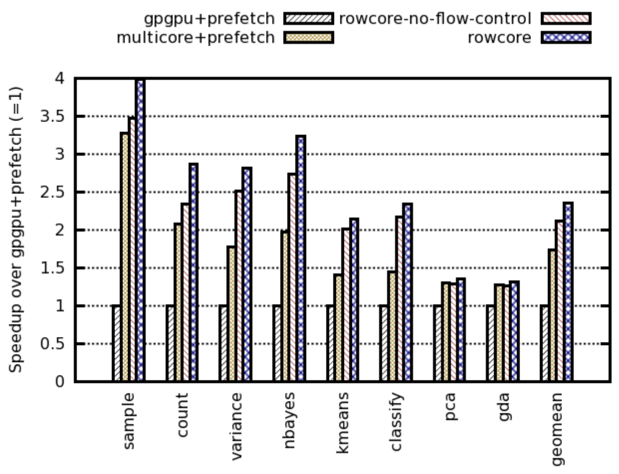

Figure 4: Performance

ber of instructions per input data word to show the trends clearly. The graph also shows RowCore without flow control (RowCore-no-flow-control) to isolate the impact of flow control. Table 4 shows the number of instructions per input data word, the branch frequency, and the number of row misses per row in multicores.

RowCore performs, on average, 135% and 35% better than GPGPU (with prefetch) and multicore (with prefetch), respectively. GPGPU (with prefetch) loses performance due to branches impeding SIMT execution (Table 4) but not due to irregular memory accesses which are handled by Shared Memory. Multicore (with prefetch) loses performance due to the cores straying from each other and destroying row locality (Table 4). In contrast, RowCore's row-oriented MIMD architecture avoids both problems. The difference between RowCore and GPGPU highlights the need for MIMD because both architectures enjoy row locality whereas the difference between RowCore and multicore highlights the need for row-orientedness because both architectures employ MIMD execution; the *only* differences between RowCore and multicores are row-orientedness and flow control. These numbers isolate the impact of RowCore's novel architectural features over GPGPU and multicore while holding technology (CMOS and die-stacking) and software (layout) effects constant.

The benchmarks are in the order of increasing number of instructions per input data word from left to right in Figure 4 (top to bottom in Table 4). Consequently, memory-bandwidth-boundedness and branch frequency decrease from left to right (i.e., computation per record increases while memory accesses and branches per record stays the same). Accordingly, RowCore's MIMD and row-orientedness advantages, respectively, over GPGPUs and multicore decrease from left to right causing RowCore's speedups to decrease.

Comparing RowCore-no-flow-control and multicore isolates the benefits of row-orientedness. The former includes row-centric access and compute-schedule via full-row prefetching (Section 4.2) but not flow control so that filling up of the prefetch buffers can cause premature eviction of prefetched data due to corelet straying (Section 4.3). However, such eviction is not common with 16 buffers allowing RowCore-no-flow-control to improve over multicores. Adding flow

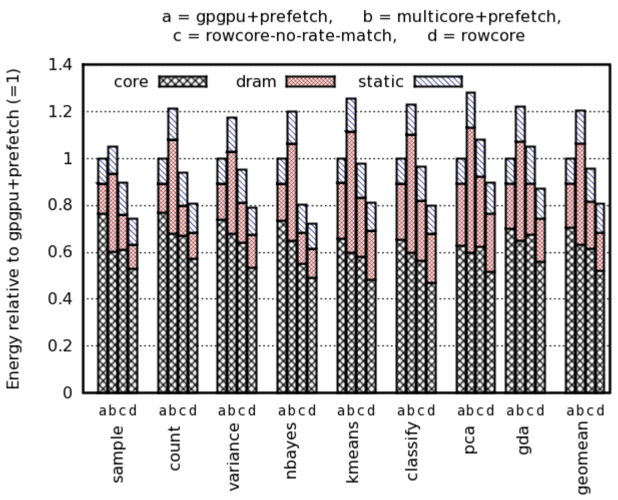

Figure 5: Energy

Benchmark	sample	count	variance	nbayes	kmeans	classify	pca	gda
Clock freq. (MHz)	528	544	581	565	613	625	644	644

Table 5: Rate-matching Frequency for RowCore

control improves performance further by avoiding such evictions (the RowCore bars). Thus, this graph isolates the benefits of RowCore's row-orientedness and flow-controlled prefetch (our second and third contributions). RowCore's rate-matching is an energy optimization analyzed next.

6.2 Energy

We now compare the three architectures in terms of energy. Figure 5 shows the architectures' energy on the Y axis, normalized to that of GPGPU. We show two variants of RowCore, 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.

GPGPUs incur higher core energy than multicores due to (1) higher local memory energy than multicores due to the power-hungry Shared Memory (Section 3.5), and (2) higher idle energy due to branches (Section 3.5). However, GPGPUs achieve lower DRAM energy than multicores because, unlike MIMD multicores, their memory accesses do not stray across rows. The net result of these factors is that multicores expend more energy than GPGPUs. Comparing multicores to RowCore without rate-matching, we see that RowCore achieves similar core energy as multicore because both architectures (1) use private, local memories avoiding the crossbar energy of GPGPU's Shared Memory, and (2) avoid GPGPU's branch inefficiency via MIMD execution. However, RowCore achieves much better (lower) memory energy due to its row-orientedness. Recall from Section 4.6 that RowCore's rate-matching slows down the corelets when applications are memory-bandwidth-bound. Figure 5 shows RowCore with rate-match further reduces the core energy

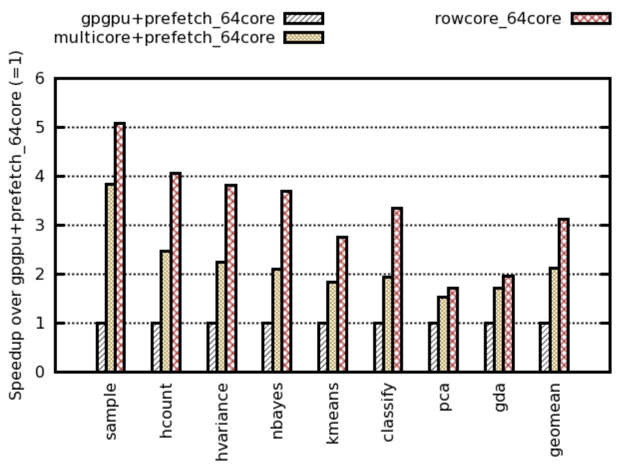

Figure 6: Speedup versus system size

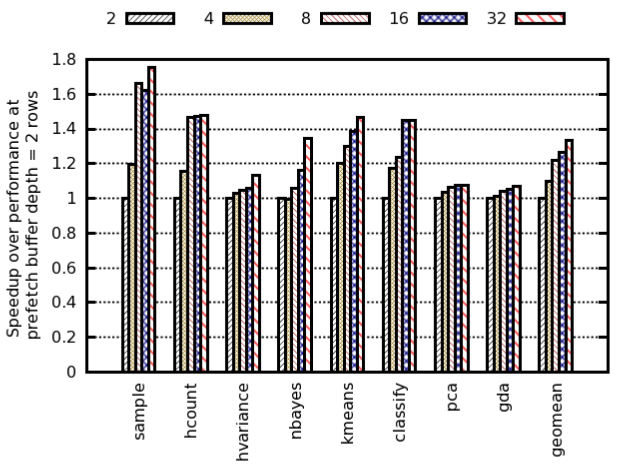

Figure 7: Speedup versus prefetch buffer count

over RowCore without rate-matching by 15% via DFS. While the nominal frequency is 700MHz (Table 3), Table 5 shows the clock speeds under rate-matching which inversely correlate with the number of instructions per input word in Table 4 (i.e., fewer instructions implies more DRAM-bandwidth-bound and therefore slower clock). These numbers isolate the impact of RowCore's rate-matching (our fourth contribution). Static energy of the cores and caches are comparable across the architectures with GPGPUs consuming slightly less energy than multicore and both RowCore variants due to their extra I-cache (Section 5). Overall, RowCore with rate-matching dissipates 20% and 34% less energy than GPGPU and multicore, respectively. Note that though multicore is closer to RowCore in performance for *PCA* and *GDA* than the other benchmarks (Figure 4), multicore incurs higher energy than RowCore for these benchmarks (Figure 5) due to numerous row misses (Table 4) which can be hidden in execution time but not in energy.

6.3 Sensitivity to system size

We change the number of corelets, lanes, and cores per RowCore processor, GPGPU SM, and multicore, from 32 (default) to 64, and correspondingly double the memory bandwidth. Figure 6 shows the performance of the three architectures normalized to that of a 64-lane GPGPU. As the lane count increases, GPGPU's branch inefficiency increases compared to RowCore which can gainfully utilize more corelets. Consequently, RowCore's speedup over GPGPU increases with more corelets (32 corelets in Figure 4 versus 64 corelets in Figure 6). Similarly, as the core count increases, the cores of a multicore stray from each other more disrupting row locality more. Therefore, RowCore's speedup over multicore also increases with more cores (Figure 4 versus Figure 6).

6.4 Sensitivity to prefetch buffer count

Recall from Section 4.3 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 RowCore processor for 2-KB rows.

7. CONCLUSION

This paper matched Big Data machine learning (BDML) applications with die-stacking via processing-near-memory (PNM). BDMLs 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 computation has a small intermediate live data and uses only a small amount of contiguous input data); and (c) memory-row-dense (i.e., process the input data without skipping over many bytes). While previous PNM work explores general MapReduce workloads, these characteristics (except for irregularity) are necessary for bandwidth- and energy-efficient PNM, irrespective of the architecture.

Based on these characteristics, we proposed RowCore, a row-oriented PNM architecture, which exploits BDMLs' row-density by (pre)fetching and operating on entire memory rows. Instead of this row-centric access and compute-schedule, traditional architectures opportunistically improve row locality while fetching and operating on cache blocks. RowCore handles BDMLs' irregularity and memory latency by employing MIMD execution and sequential prefetch of input data. However, one RowCore 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. RowCore employs novel cross-corelet flow-control to prevent such eviction. RowCore further exploits this flow control for frequency scaling based on novel coarse-grain compute-memory rate-matching. Using simulations, we show that RowCore improves performance and energy, by 135% and 20% over a GPGPU with prefetch, and by 35% and 34% over a multicore with prefetch, when all three architectures use the same resources (i.e., number of cores and on-processor-die memory) and identical die-stacking (i.e., GPGPU/multicore/RowCore and DRAM.

8. REFERENCES

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

[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] H. S. Stone, “A logic-in-memory computer,” *Computers, IEEE Transactions on*, vol. C-19, pp. 73–78, Jan 1970.

[4] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, “A case for intelligent ram,” *IEEE Micro*, vol. 17, pp. 34–44, Mar. 1997.

[5] J. B. Brockman, S. Thoziyoor, S. K. Kuntz, and P. M. Kogge, “A low cost, multithreaded processing-in-memory system,” in *Proceedings of the 3rd Workshop on Memory Performance Issues: In Conjunction with the 31st International Symposium on Computer Architecture*, WMPI '04, (New York, NY, USA), pp. 16–22, ACM, 2004.

[6] R. Nair, S. F. Antao, C. Bertolli, P. Bose, J. R. Brunheroto, T. Chen, C. Y. Cher, C. H. A. Costa, J. Doi, C. Evangelinos, B. M. Fleischer, T. W. Fox, D. S. Gallo, L. Grinberg, J. A. Gunnels, A. C. Jacob, P. Jacob, H. M. Jacobson, T. Karkhanis, C. Kim, J. H. Moreno, J. K. O'Brien, M. Ohmacht, Y. Park, D. A. Prener, B. S. Rosenburg, K. D. Ryu, O. Sallenave, M. J. Serrano, P. D. M. Siegl, K. Sugavanam, and Z. Sura, “Active memory cube: A processing-in-memory architecture for exascale systems,” *IBM Journal of Research and Development*, vol. 59, pp. 17:1–17:14, March 2015.

[7] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, and J. Park, “Mapping irregular applications to diva, a pim-based data-intensive architecture,” in *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, SC '99, (New York, NY, USA), ACM, 1999.

[8] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, “NDC: analyzing the impact of 3d-stacked memory+logic devices on mapreduce workloads,” in *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, Monterey, CA, USA, March 23-25, 2014*, pp. 190–200, 2014.

[9] M. Gao, G. Ayers, and C. Kozyrakis, “Practical near-data processing for in-memory analytics frameworks,” in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pp. 113–124, IEEE, 2015.

[10] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, “A scalable processing-in-memory accelerator for parallel graph processing,” in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), pp. 105–117, ACM, 2015.

[11] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, “Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules,” in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pp. 283–295, Feb 2015.

[12] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas, “Flexram: toward an advanced intelligent memory system,” in *Computer Design, 1999. (ICCD '99) International Conference on*, pp. 192–201, 1999.

[13] M. Gokhale, B. Holmes, and K. Iobst, “Processing in memory: The terasys massively parallel pim array,” *Computer*, vol. 28, pp. 23–31, Apr. 1995.

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

[15] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster computing with working sets,” in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2010.

[16] N. Mirzadeh, O. Kocberber, B. Falsafi, and B. Grot, “Sort vs. Hash Join Revisited for Near-Memory Execution,” in *5th Workshop on Architectures and Systems for Big Data (ASBD 2015)*, 2015.

[17] M. Kamruzzaman, S. Swanson, and D. M. Tullsen, “Inter-core prefetching for multicore processors using migrating helper threads,” in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, (New York, NY, USA), pp. 393–404, ACM, 2011.

[18] C. Kaynak, B. Grot, and B. Falsafi, “Shift: Shared history instruction fetch for lean-core server processors,” in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, (New York, NY, USA), pp. 272–283, ACM, 2013.

[19] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, “Coordinated control of multiple prefetchers in multi-core systems,” in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, (New York, NY, USA), pp. 316–326, ACM, 2009.

[20] Q. Wu, P. Juang, M. Martonosi, and D. W. Clark, “Voltage and frequency control with adaptive reaction time in multiple-clock-domain processors,” in *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pp. 178–189, Feb 2005.

[21] A. Iyer and D. Marculescu, “Power efficiency of voltage scaling in multiple clock multiple voltage cores,” in *Computer Aided Design, 2002. ICCAD 2002. IEEE/ACM International Conference on*, pp. 379–386, Nov 2002.

[22] T. W. Bartenstein and Y. D. Liu, “Green streams for data-intensive software,” in *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, (Piscataway, NJ, USA), pp. 532–541, IEEE Press, 2013.

[23] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, “Dynamic knobs for responsive power-aware computing,” in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, (New York, NY, USA), pp. 199–212, ACM, 2011.

[24] B. Y. Cho, W. S. Jeong, D. Oh, and W. W. Ro, “XSD: Accelerating MapReduce by Harnessing GPU inside SSD,” in *1st Workshop on Near Data Processing (WoNDP 2013) In Conjunction with the 46th International Symposium on Microarchitecture*, 2013.

[25] R. G. Dreslinski, D. Fick, B. Giridhar, G. Kim, S. Seo, M. Fojtik, S. Satpathy, Y. Lee, D. Kim, N. Liu, M. Wieckowski, G. Chen, D. Sylvester, D. Blaauw, and T. Mudge, “Centip3de: A 64-core, 3d stacked near-threshold system,” *IEEE Micro*, vol. 33, pp. 8–16, March 2013.

[26] T. G. Rogers, M. O'Connor, and T. M. Aamodt, “Cache-conscious wavefront scheduling,” in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, (Washington, DC, USA), pp. 72–83, IEEE Computer Society, 2012.

[27] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson, “Near-data processing: Insights from a micro-46 workshop,” *IEEE Micro*, vol. 34, pp. 36–42, July 2014.

[28] Q. Guo, X. Guo, R. Patel, E. Ipek, and E. G. Friedman, “Ac-dimm: Associative computing with stt-mram,” in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, (New York, NY, USA), pp. 189–200, ACM, 2013.

[29] B. Akin, F. Franchetti, and J. C. Hoe, “Data reorganization in memory using 3d-stacked dram,” in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), pp. 131–143, ACM, 2015.

[30] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, “Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture,” in *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, pp. 336–348, June 2015.

[31] R. St. Amant, A. Yazdanbakhsh, J. Park, B. Thwaites, H. Esmaeilzadeh, A. Hassibi, L. Ceze, and D. Burger, “General-purpose code acceleration with limited-precision analog computation,” in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, (Piscataway, NJ, USA), pp. 505–516, IEEE Press, 2014.

[32] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, “Shidiannao: Shifting vision processing closer to the sensor,” in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), pp. 92–104, ACM, 2015.

[33] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha, “A million spiking-neuron integrated circuit with a scalable communication network and interface,” *Science*, vol. 345, no. 6197,

pp. 668–673, 2014.

[34] D. E. Shaw, M. M. Deneroff, R. O. Dror, J. S. Kuskin, R. H. Larson, J. K. Salmon, C. Young, B. Batson, K. J. Bowers, J. C. Chao, M. P. Eastwood, J. Gagliardo, J. P. Grossman, C. R. Ho, D. J. Ierardi, I. Kolossváry, J. L. Klepeis, T. Layman, C. McLeavey, M. A. Moraes, R. Mueller, E. C. Priest, Y. Shan, J. Spengler, M. Theobald, B. Towles, and S. C. Wang, "Anton, a special-purpose machine for molecular dynamics simulation," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, (New York, NY, USA), pp. 1–12, ACM, 2007.

[35] J. P. Grossman, J. S. Kuskin, J. A. Bank, M. Theobald, R. O. Dror, D. J. Ierardi, R. H. Larson, U. B. Schafer, B. Towles, C. Young, and D. E. Shaw, "Hardware support for fine-grained event-driven computation in anton 2," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, (New York, NY, USA), pp. 549–560, ACM, 2013.

[36] J. Lee, N. B. Lakshminarayana, H. Kim, and R. Vuduc, "Many-thread aware prefetching mechanisms for gpgpu applications," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, (Washington, DC, USA), pp. 213–224, IEEE Computer Society, 2010.

[37] Y. Yetim, S. Malik, and M. Martonosi, "Eprof: An energy/performance/reliability optimization framework for streaming applications," in *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, pp. 769–774, Jan 2012.

[38] C. Chu, S. K. Kim, Y. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun, "Map-reduce for machine learning on multicore," in *Advances in Neural Information Processing Systems 19, Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 4-7, 2006*, pp. 281–288, 2006.

[39] A. J. Smith, "Cache memories," *ACM Comput. Surv.*, vol. 14, pp. 473–530, Sept. 1982.