Proc of Int Conf for High Performance Computing, Networking, Storage and Analysis (SC21)

CAKE: Matrix Multiplication Using Constant-Bandwidth Blocks

H. T. Kung
Harvard University
Cambridge, MA, USA
kung@harvard.edu

Vikas Natesh
Harvard University
Cambridge, MA, USA
vnatesh@g.harvard.edu

Andrew Sabot
Harvard University
Cambridge, MA, USA
asabot@g.harvard.edu

ABSTRACT

We offer a novel approach to matrix-matrix multiplication computation on computing platforms with memory hierarchies. Constant-bandwidth (CB) blocks improve computation throughput for architectures limited by external memory bandwidth. Configuring the shape and size of CB blocks operating from within any memory hierarchy level (e.g., internal SRAM), we achieve high throughput while holding external bandwidth (e.g., with DRAM) constant. We explain how, surprisingly, CB blocks can maintain constant external bandwidth as computation throughput increases. Analogous to partitioning a cake into pieces, we dub our CB-partitioned system CAKE.

We show CAKE outperforms state-of-the-art libraries in computation time on real-world systems where external bandwidth represents a bottleneck, demonstrating CAKE's ability to address the memory wall. CAKE achieves superior performance by directly using theoretically optimal CB-partitioned blocks in tiling and scheduling, obviating the need for extensive design search.

KEYWORDS

memory management, computation for deep neural network (DNN), parallel processing, parallel architectures, optimal block scheduling, arithmetic intensity, matrix multiplication, memory wall

ACM Reference Format:
H. T. Kung, Vikas Natesh, and Andrew Sabot. 2021. CAKE: Matrix Multiplication Using Constant-Bandwidth Blocks. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3458817.3476166

1 INTRODUCTION

Matrix-matrix multiplication (MM) underlies many computational workloads in scientific computing and machine learning. For example, most computations in the forward pass of a convolutional neural network consist of one matrix multiplication per convolutional layer between the inputs to and the weights of a layer (see, e.g., [26]). Computation throughput for MM depends on processing power, local memory bandwidth (e.g., between internal SRAM and processing cores), local memory size, and DRAM bandwidth. However, performance gains from increasing processing power are limited by the other three factors. For example, DRAM bandwidth may become the limiting factor as more processing power is added.

Current approaches to MM (e.g., Goto's algorithm [12]) are designed for systems where memory and compute bandwidth are presumably balanced for target workloads. However, there is a need for a new approach that can adapt to architectures with differing characteristics. These architectures may arise as a result of emerging technologies such as special-purpose accelerators [10, 23], low-power systems [20, 24], 3D DRAM die stacking [6, 14, 18, 25] and high-capacity non-volatile memory (NVM) [27]. The memory wall [30] remains a fundamental problem faced by all computing systems: shrinking technology processes do not directly address this issue. Finding a computation schedule which, for example, can maximize data reuse in the local memory to alleviate this disparity between memory and compute can be challenging. Often, the computation schedule is found through a grid search of the parameter space, which becomes computationally intractable for large systems.

This paper proposes the CAKE system that utilizes **constant-bandwidth** (CB) blocks in computation partitioning and block scheduling. CAKE offers a theory for optimal partitioning and substantially reduces the search space for an optimal schedule. A CB block is a block of computation with the property that, when computing the block from within a local memory, the required external bandwidth is constant. We can design a CB block capable of achieving a target computation throughput by controlling its shape and size (Section 3). With sufficient local memory resources, the CAKE algorithm can improve MM computation throughput without having to increase external DRAM bandwidth.

In CAKE, we partition the MM computation space, a 3D volume of multiply-accumulate (MAC) operations, into a grid of 3D CB blocks. The blocks are scheduled and then sequentially executed on a computing platform comprising multiple computing cores (see Figure 1). This scheme is analogous to how a host can cut and serve a "cake" to their guests. To consume the entire cake as quickly as possible, each guest must continuously eat: the rate of serving pieces must match the guests' rate of consumption.

The intuition behind CAKE is that, by adjusting the shape (i.e., aspect ratios) and size of CB blocks, we can control the ratio of computations to external memory accesses, i.e., arithmetic intensity (see Figure 4). As a result, we can use CB blocks to increase the use of available computing power without requiring a comparable increase in IO bandwidth to external DRAM. We analytically determine the shape and size of a CB block from available DRAM bandwidth and computing resources (Section 3). Under the CB framework, we can precisely characterize the required size and bandwidth of local memory for achieving a target computation throughput with a given external memory bandwidth.

We evaluate CAKE's performance in computation throughput and external memory bandwidth. Experiments are conducted on two desktop CPUs and a low-power embedded CPU. On the desktop CPUs, we found CAKE can outperform state-of-the-art GEMM libraries in computation throughput (Sections 5.2.5 and 5.2.6). For the low-power system, CAKE achieves substantially higher throughput

SC '21, November 14–19, 2021, St. Louis, MO, USA
2021. ACM ISBN 978-1-4503-8442-1/21/11.
https://doi.org/10.1145/3458817.3476166

Table 1: Terminology used throughout the paper

Term	Description
MM	matrix-matrix multiplication
CB	constant-bandwidth block
Tile	small matrix processed by a core (Section 3) or SIMD registers (Section 4.1)
Local Memory	cache on CPUs
Internal BW	bandwidth between last level cache and CPU cores
External BW	bandwidth between local memory and external memory (DRAM)

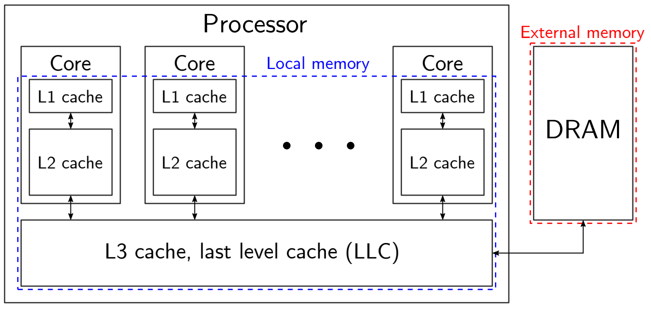

Figure 1: Computing architecture with many cores, multiple levels of local memory, and external memory.

than vendor libraries (Section 5.2.4). Table 1 lists terminologies used in this paper. The main contributions of this paper are:

- MM block scheduling with surface sharing (Section 2).
- CAKE constant-bandwidth blocks: blocks analytically shaped and sized to reduce external memory access requirements (Section 3).
- Comparison of CAKE and Goto's algorithms (Section 4).
- CAKE library: a drop-in replacement for MM calls used by existing frameworks that does not require manual tuning.
- Demonstration of CAKE library performance gains on 3 different CPU architectures (Section 5).

2 BLOCK MM COMPUTATION

In this section, we introduce a block framework for matrix multiplication (MM) computation and how we may partition the computation space into blocks. Algorithm 1 defines MM where reduction occurs on dimension K.

Consider an MM between matrices A and B, where A is size $M \times K$ and B is size $K \times N$. The MM can be computed via a set of vector operations using one of the two strategies. That is, we can obtain the MM result C through $M \cdot N$ inner products between M row vectors (size $1 \times K$) of A and N column vectors (size $K \times 1$) of B, or summation of K outer products between column vectors (size $M \times 1$) of A and the corresponding row vectors (size $1 \times N$) of B.

Outer products, unlike inner products, yield partial result matrices which will be summed together to produce C, allowing reuse and in-place accumulation. In the $A \times B$ example (Figure 2), there are K outer products between vectors of size $M \times 1$ and $1 \times N$, which each produce partial result matrices of size $M \times N$. Partial results are accumulated across the K-dimension (reduction dimension). Note that we may store the intermediate partial result matrix locally to be accumulated in place with forthcoming partial result matrices. Thus the locally stored intermediate results are reused K times. We use outer-product MM throughout this paper to leverage this reuse.

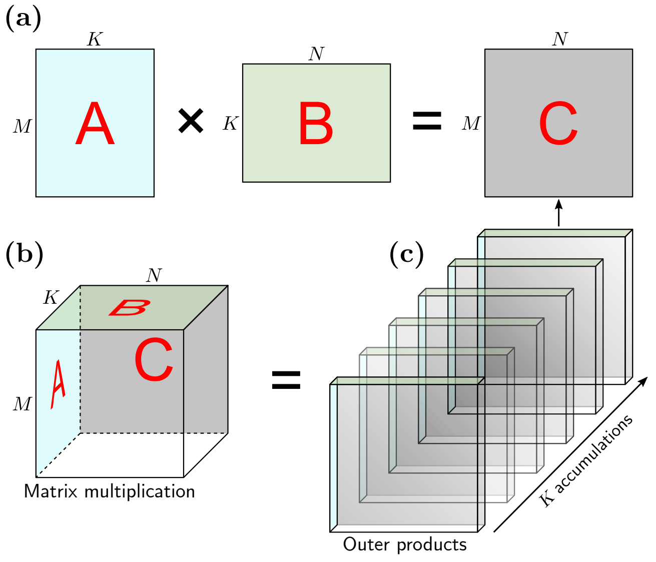

Figure 2: (a) $C = A \times B$ **matrix multiplication. (b) Computation space represented as an** $M \times K \times N$ **3D volume of multiply-accumulate (MAC) operations as defined by Algorithm 1. (c) Computation space as an accumulation of outer products.**

Algorithm 1: Matrix multiplication algorithm

```
for $i = 1 \rightarrow M$ do
    for $j = 1 \rightarrow N$ do
        for $k = 1 \rightarrow K$ do
            $C[i][j] \longleftarrow C[i][j] + A[i][k] \cdot B[k][j]$;
```

The **computation space** for $C = A \times B$ is a 3D volume of $M \cdot N \cdot K$ multiply-accumulate (MAC) operations depicted in Figure 2b. The volume shown in Figure 2b is determined by 3 IO surfaces: In this paper, we consider matrix A as being on the "left" side, matrix B being on the "top", and matrix C being the "back wall".

2.1 Block Framework for MM Computation

If a computing platform has a sufficient local memory (internal memory) to hold the 3 IO surfaces, all MACs in the 3D computation space may be completed without any additional IO operations. For large MMs exceeding local memory capacity, the computation space must be partitioned and computed in smaller blocks, whose results are later combined.

In CAKE, we use uniform blocks of the same shape and size. As shown in Figure 3, we partition the $M \times N \times K$ computation space into $m \times n \times k$ blocks. Section 3 describes how block dimensions are calculated based on available resources, such as total available computing power and external DRAM bandwidth. Computation of a block may be viewed as a sum of k outer products, for multiplying the two corresponding sub-matrices within the larger MM computation space. The computation yields the back wall of the block. Block computations result in $m \times n$ sub-matrices which, when accumulated together, yield a portion of the final matrix C. Matrix addition is commutative, so computation order for blocks in the computation space does not matter for correctness.

All cores in the processing grid work (Figure 3b) in parallel, on input tiles at the rate of one tile result per unit time for each core, to compute a block by performing k outer products. Each core works through the N-dimension of the block computation space, producing a row of partial results by computing tile-wise multiplications between a single A tile and n B tiles. It is also possible to compute computation blocks in the M or K-dimension but we focus our presentation on the N-dimension. Each column of cores in the grid (e.g., cores 1, 5, 9, 13 in Figure 3b) computes an outer product between a sub-column of A and a sub-row of B to produce a partial result sub-matrix. The resulting sub-matrices are accumulated, yielding a partial sub-matrix which will be further accumulated with results from other blocks.

Intra-block data movement is reduced by each core sequentially reusing one A tile with many B tiles. The B tiles are broadcast to cores in the same column to maximize intra-block reuse. Partial results are summed along the K dimension (towards the back of the computation space), maximizing reuse via in-place accumulation.

As noted earlier, a block is defined by three IO surfaces: an input surface A of size $m \times k$, an input surface B of size $k \times n$, and a result surface C of size $m \times n$. Surfaces A and B correspond to the aforementioned sub-matrices within the larger MM computation space. Depending on the location of the block within the computation space, result surface C will consist of either partial or completed reduction results.

For each block, the total external memory IO and required local memory size are both equal to the sum of the three IO surfaces. External memory bandwidth requirements may be determined from the computation time of the block. When the block is shaped properly (see Figure 4), the IO time for the three surfaces will match the computation time of the block, allowing IO to overlap computation that fully utilizes available processing power.

IO requirements are further reduced when sequentially computed blocks share an IO surface (i.e., the blocks are adjacent within the computation space). The surface can be kept local: the following adjacent block can *reuse* the surface without needing to fetch it from external memory. Furthermore, if the surface is a partial result surface, the previous block does not need to writeback the results to external memory before the next computation. IO surfaces can be shared in the M, N, or K-dimension, and IO cost is minimized when the largest IO surface is reused most frequently.

2.2 Scheduling Blocks for MM Computation

To minimize IO, blocks in the MM computation space are scheduled so adjacent blocks are computed in sequence. Per the cake analogy, scheduling is analogous to cutting a cake into large chunks (blocks) that are then divided into smaller pieces for each guest (core). The partitioning (Figure 3b) is then used to generate a sequence of blocks, which are sequentially executed on the grid of cores.

Recall the 3 types of IO: A, B and results C (where results are either partial or complete). The IO for a partial result is twice that

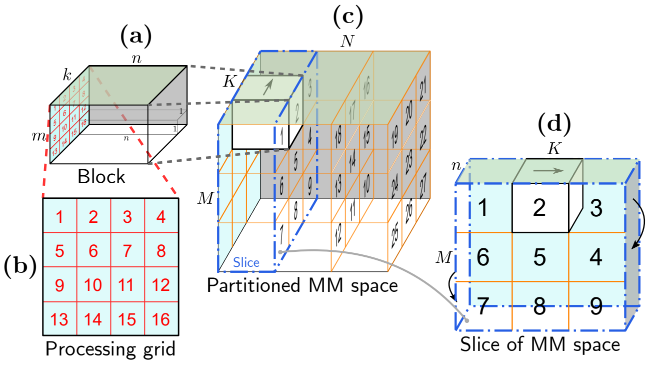

Figure 3: (a) Block defined by an $k \times m$, $n \times k$ input surfaces and an $m \times n$ output surface. (b) Grid of 16 processing cores. (c) Block-partitioned computation space for an MM between $M \times K$ and $K \times N$ matrices. (d) Rotated view of a slice of the computation space. The numbers represent the order of execution for blocks in a K-first schedule.

of a completed result, A, or B because it must be written back to external memory and, later, fetched again for further use. A schedule that minimizes external IO must therefore completely reuse all partial surfaces, and then reuse A and B. When a partial dimension is complete there is a choice of which surface to reuse next: A (in the M dimension) and B (in the N dimension). When B is reused first, the M-dimension is completed after the K-dimension and vice versa if A is reused first. We refer to this schedule as a K-first (reduction-first) schedule.

The choice of whether to first reuse A (in the N-dimension) or B (in the M-dimension) after completing a reduction run (e.g., blocks 1, 2 and 3 in Figure 3d) depends on the shape of the computation space. When the B surface is larger ($N > M$) reusing the larger B surface before the A surface minimizes IO. This is done by computing the M-dimension before the N-dimension.

To allow inter-block reuse between different dimensions (e.g., between K and M), the computation space traversal must "turn" every time it completes a dimension. If instead the loops always started at the 0 index of a dimension, no A or B surfaces would be reused, leading to $O(MN + N)$ missed IO surface reuses. The traversal direction flips after each dimension to allow for IO surface reuse, shown in Algorithm 2. Note the pseudocode assumes $N \geq M$, when $M > N$ the outer two loops are switched because the A surfaces should be reused before the B surfaces. The algorithm defines the K-first computation order of blocks, which sweeps the space of computation space by first traversing the K-dimension to maximize partial result reuse, then the M-dimension to reuse A, and lastly the N-dimension to reuse B.

3 CONSTANT BANDWIDTH BLOCK SHAPE AND SIZE

A constant bandwidth (CB) block is a block (described in Section 2.1) with dimensions (n, m, k) shaped and sized according to external bandwidth (as seen in Figure 4). CB block shaping provides control over **arithmetic intensity** (AI), allowing us to match external IO time with computation time. AI is defined as the ratio of computation volume to data transferred, which is equivalent to the ratio

Algorithm 2: *K*-first block partitioning algorithm

```
// Get the number of blocks in each dimension
$M_b = M/m; N_b = N/n; K_b = K/k$
for $n_{idx} = 0, 1, \ldots$ to $N_b - 1$ do
    // Flip direction of $M$ traversal for $A$ reuse
    if $n_{idx} \bmod 2 == 0$ then $m_{start} = 0; m_{end} = M_b$;
    else $m_{start} = M_b; m_{end} = 0$;
    for $m_{idx} = m_{start}$ to $m_{end}$ do
        // Flip direction of $k$ traversal for $B$ reuse
        if $n_{idx} \bmod 2 == 0$ then ;
            if $m_{idx} \bmod 2 == 0$ then $k_{start} = 0; k_{end} = K_b$;
            else $k_{start} = K_b; k_{end} = 0$;
        else
            if $m_{idx} \bmod 2 == 0$ then $k_{start} = K_b; k_{end} = 0$;
            else $k_{start} = 0; k_{end} = K_b$;
        for $k_{idx} = k_{start}$ to $k_{end}$ do
            // Compute inner block multiplication
            $C[m_{idx} * m][n_{idx} * n] \ += $
              $A[m_{idx} * m][k_{idx} * k] \times B[k_{idx} * k][n_{idx} * n]$;
```

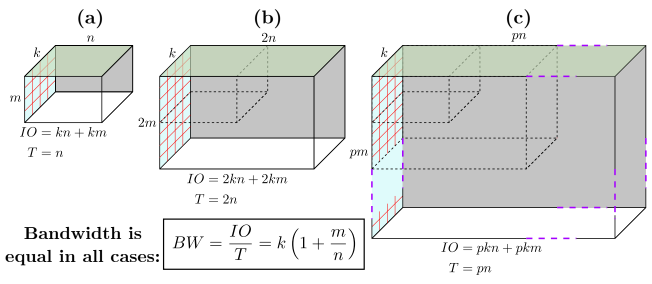

Figure 4: Changing the shape and size of a block can keep a block's external bandwidth (*BW*) constant when increasing the computation throughput (CT) by increasing core count. The block gets taller and wider, matching *IO* and computation time *T* as volume (*V*) changes. Note that *arithmetic intensity* is V/IO. Since $V/IO = \frac{V/T}{IO/T} = \frac{CT}{BW}$, and CB blocks in (a), (b) and (c) have equal *BW* and increasing volume, they have increasing arithmetic intensity. Importantly, computation throughput is V/T. Thus, CB blocks in (a), (b) and (c) have increasing computation throughput at $\frac{mkn}{n} = mk$, $\frac{4mkn}{2n} = 2mk$, and $\frac{p^2 mkn}{pn} = pmk$, respectively.

of computation throughput (CT) to external memory bandwidth (BW): $AI = \frac{CT}{BW}$. Therefore, we can, for example, increase CT or decrease required BW by using CB blocks to control AI. We can also increase the size of the CB block to leverage any additional BW and decrease internal memory size requirements.

Consider a computing architecture with a number of cores, each performing one tile multiplication per unit time. As seen in Section 2.1, each core handles one tile from *A*, so the number of tiles in the *A* surface (size $m \times k$) of a CB block is equal to the number of cores. The size of k determines how many cores contribute their partial results for accumulation. In this analysis, we assume k is a fixed optimal value calculated from available external bandwidth (Section 3.2). Given k, we can compute m so that $m \cdot k$ is the number of available cores (Figure 3a and 3b). We reflect an increase in the amount of cores used in p (e.g., when trying to increase CT by doubling the number of cores used, p would increase by a factor of 2). To reduce the number of variables in our analysis, we set m to a multiple of k, based on the number of available cores (i.e., $m = pk$), but m does not need to be a multiple of k.

Thus, the CB block shape is defined by $m = pk$ and $n = \alpha pk$ where $\alpha \geq 1$ and k are unitless constants calculated from available external memory bandwidth (see Section 3.2). When external memory bandwidth is low, raising α increases block computation time, thereby decreasing the CB block's external bandwidth requirement (BW). When there is sufficient external bandwidth, α is set to 1.

To compute a CB block in the N-dimension, each core is first loaded with one A tile. B tiles are then streamed to each core from local memory (e.g., L3 cache). The CB block is shaped to have exactly one A tile per core, keeping A tiles stationary, to reduce local congestion. Results are accumulated between cores and **cycled back** to local memory (e.g., L3 cache) for reuse and moved to external DRAM when complete. The computation time T for a CB block is $n = \alpha pk$ unit times because each core is assigned to compute n tile multiplications.

Alternatively, we can compute a CB block in the M or K-dimension, resulting in a CB block computation time of k or m unit times, respectively. Computing CB blocks in alternative directions may be advantageous on certain architectures. For example, computing CB blocks in the K-dimensions is preferable when doing in-place accumulation. In a future paper we will show how the same shaping methodology applies when computing CB blocks in the M or K-dimension. In this analysis, we do not factor in accumulation time as we assume accumulation can be overlapped with multiplication.

3.1 Internal Memory Size Requirement

A CB block consists of 3 IO surfaces that must be stored locally. The IO for each surface is equal to its size. For A, $IO_A = pk \cdot k = pk^2$. For B, $IO_B = k \cdot \alpha pk = \alpha pk^2$. For result surface C, $IO_C = pk \cdot \alpha pk = \alpha p^2 k^2$. The internal memory size requirement is simply the combined size of the surfaces:

$$MEM_{internal} = IO_A + IO_B + IO_{partial} = \alpha pk^2 + pk^2 + \alpha p^2 k^2 \quad (1)$$

To increase the target processing power p-fold, internal memory size must increase by a factor of p^2 (due to the third term, $\alpha p^2 k^2$).

3.2 External Bandwidth Analysis

We may compute the minimum external bandwidth required for a CB block based on the external IO for its A and B surfaces, using the following equation, where T is the block computation time:

$$BW_{min} = \frac{IO_A + IO_B}{T} = \frac{\alpha pk^2 + pk^2}{\alpha pk} = \left(\frac{\alpha + 1}{\alpha}\right) \cdot k \text{ tiles/cycle} \quad (2)$$

Increasing α allows us to compensate for low external bandwidth but increases both computation time and required local memory size. Partial result IO is not considered since results are held locally.

We define external bandwidth as $BW_{ext} = R \cdot k$ tiles/cycle, where $R > 1$ is a constant capturing the difference between available external bandwidth and the minimum bandwidth defined previously. We satisfy the minimum external bandwidth requirement when $BW_{ext} \geq BW_{min}$, or $\alpha \geq \frac{1}{R-1}$.

Now consider the case when more processing power is available (e.g., when the number of cores increases from 16 to 32, as in Figure 4). We choose to increase the N and M-dimensions by a factor of $p = 2$ because IO and computation time will increase by the same factor. BW_{min} does not depend on p, which increases both computation and IO time equally. As a result, we can increase the number of utilized cores (pk^2) while keeping the same external bandwidth requirement (see Figure 4).

3.3 Internal Bandwidth Requirements

Recall that the local memory holds 3 IO surfaces: two input surfaces and one result surface. During a CB block computation, each input surface is read once and loaded onto the cores. The partial result surface is accessed twice: once for reading and once for storing new partial results. We see the internal bandwidth must be at least:

$$\frac{IO_A + IO_B + 2IO_{partial}}{T} = Rk + 2pk \text{ tiles/cycle} \tag{3}$$

Thus, as the number of cores (pk^2) increases, internal bandwidth must increase proportionally (due to the second term, $2pk$) to match external IO time with computation time.

4 ANALYSIS OF CAKE AND GOTO ON CPUS

Modern CPUs contain a multilevel memory hierarchy consisting of external memory, a shared cache for all cores, and local caches on each core (Figure 1). We compare CAKE and Goto's algorithm [13] (hereafter referred to as GOTO) by adapting our computation throughput and memory bandwidth analysis from Section 3 to CPUs with a multilevel memory hierarchy. In our analysis, we assume the memory hierarchy comprises a local L1 and L2 cache per core, a shared L3 cache for all cores, and external DRAM.

During an MM computation, there are various ways of utilizing the different levels of memory. Algorithms that reuse data to different degrees will differ in system resource requirements including external memory bandwidth, internal memory bandwidth, and size of local memories (caches). To increase computation throughput via utilizing additional cores, algorithms must mitigate the constraints imposed by cache size bottlenecks. In the previous analysis (Section 3), we expressed the number of cores as mk. If we allow $k > 1$ on the CPU, we are limited to using a multiple of k cores. For instance, if $k = 2$, we can only use $mk = 2, 4, 6, \ldots$ cores. When $k = 1$, we can use any number of cores between 1 and p to clearly demonstrate scaling up to p cores. For this reason, throughout Section 4, we use $k = 1$.

In CAKE, when using additional cores, we increase both m and n by the same factor to maintain the same external bandwidth requirement. In contrast, GOTO only increases m when using additional cores.

We calculate the maximum-possible computation throughput allowed by available cores given a fixed external DRAM bandwidth for both CAKE and GOTO. We show that, unlike GOTO, by using internal memory resources, CAKE need not increase external bandwidth to increase computation throughput when using an increased number of cores.

4.1 Bandwidth Analysis for GOTO

GOTO is the current state-of-the-art algorithm for MM on CPUs, and is widely adopted by many libraries including Intel Math Kernel Library (MKL) [7], ARM Performance Libraries [4] , and OpenBLAS [31]. While GOTO is able to achieve high performance when fully utilizing available DRAM bandwidth, its bandwidth requirement quickly becomes a limiting factor when trying to use additional cores.

We describe GOTO on a CPU with p cores, as depicted in Figure 5. In the previous analysis (Section 3), we expressed the number of cores as $m = pk$. With $k = 1$ for the CPU, we see that p is the number of cores. Each of the p cores works on an independent portion of the MM computation (i.e., there is no accumulation between cores).

Consider an MM computation $C = A \times B$ where A is $M \times K$, B is $K \times N$, and C is $M \times N$. The GOTO algorithm uses an outer product formulation on sub-matrices or tiles with dimensions defined in terms of parameters m_c, k_c, and n_c which are chosen based on cache and register characteristics (GOTO [13]). More precisely, C is partitioned into column sub-matrices of size $M' \times n_c$ with some M' <= M (Figure 5a). Via outer-product MM, these sub-matrices of C are computed by multiplying A's column sub-matrices (size $M' \times kc$) with B's row sub-matrices (size $k_c \times n_c$) and accumulating across the K-dimension. In GOTO, a square $m_c \times k_c$ sub-matrix (i.e., $m_c = k_c$) of A resides in the L2 cache of each core while the $k_c \times n_c$ sub-matrix of B resides in the L3 cache (Figure 5b). The values of m_c, k_c, and n_c are chosen such that each sub-matrix fits into its respective local memory level, i.e., $m_c \cdot k_c \leq Size_{L2}$ and $k_c \cdot n_c \leq Size_{L3}$. Note also that k_c only refers to the K dimension values of A's sub-matrix in the L2 cache and bears no relation to k from the previous analysis (Section 3).

At the level of SIMD registers, $m_r \times n_r$ partial result tiles are processed, where the width of the available registers determines m_r and n_r(Figure 5e). In tiled MM, $m_r \times k_c$ tiles of A are multiplied with $k_c \times n_r$ tiles of B while $m_r \times n_r$ partial result tiles of C are streamed to DRAM. For accumulation with subsequent partial results of C, previously computed partial results of C are streamed from DRAM back to the CPU cores. DRAM streaming can dominate IO.

From GOTO's description, we derive the required external DRAM bandwidth when using additional cores. Assuming a single core can multiply an $mr \times kc$ tile with an $k_c \times n_r$ tile and produce an $m_r \times n_r$ result tile in a single unit time, we obtain $m_r \cdot k_c \cdot n_r$ MAC operations per unit time. The total number of MAC operations needed to compute an $m_c \times n_c$ result sub-matrix is $k_c \cdot m_c \cdot n_c$. We can compute p $m_c \times n_c$ result sub-matrices in parallel since each of the p cores has its own L2 cache to support the local SIMD computation. Thus, the time to compute p $m_c \times n_c$ result sub-matrices is:

$$T = \frac{m_c \cdot k_c \cdot n_c}{m_r \cdot k_c \cdot n_r} = \frac{m_c \cdot n_c}{m_r \cdot n_r}$$

To compute p $m_c \times n_c$ result sub-matrices of C, we must bring in p $m_c \times k_c$ sub-matrices of A and one $k_c \times n_c$ sub-matrix of B from DRAM into local memory while also streaming p $m_c \times n_c$ partial result sub-matrices back to DRAM:

$$IO = IO_A + IO_B + IO_C = (p \cdot m_c \cdot k_c) + (k_c \cdot n_c) + (p \cdot m_c \cdot n_c)$$

The external (DRAM) bandwidth required is then:

$$BW_{ext}^{GOTO} = \frac{IO}{T} = \left(1 + p + \frac{k_c}{n_c} \cdot p\right) \cdot m_r \cdot n_r$$

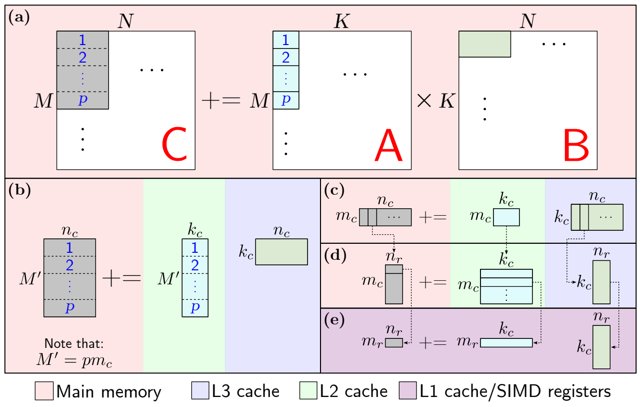

Figure 5: GOTO data reuse in the CPU memory hierarchy. (a) $m_c \times n_c$ sub-matrices of C are computed by accumulating outer products between column sub-matrices of A and row sub-matrices of B. (b) Each p sub-matrix from A is reused in the L2 cache of a core while data from B is reused in the L3 cache, and results for C are written back to external memory. (c, d) Computation of an $m_c \times n_c$ sub-matrix of C on a core. (e) Tile-level MM on CPU SIMD registers.

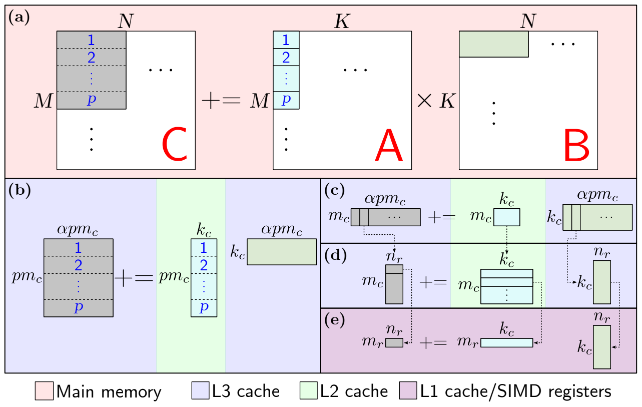

Figure 6: CAKE data reuse in the CPU memory hierarchy. (a) Schedule of CB blocks. (b) Each p sub-matrix from A is reused in the L2 cache of a core while data from B and partial results for C are reused in the L3 cache. (c, d) Computation of a single $m_c \times \alpha pm_c$ sub-matrix of C on a core, similar to Figure 5c and d. (e) Tile-level MM, performed identically to Figure 5e.

by assuming $m_c = k_c$, as noted earlier. When using an increased number of cores by a factor of p, the required external bandwidth for GOTO increases by a factor of at least p. As we will present, CAKE can mitigate this issue.

4.2 Bandwidth Analysis for CAKE

In contrast to GOTO, CAKE's bandwidth requirements do not grow proportional to the number of cores used. We analyze how shaping and sizing the CB block affects external bandwidth requirements. Figure 6 shows CAKE's block shaping on our CPU with the multilevel memory hierarchy. The computation space of $C = A \times B$ is partitioned into a 3D grid of CB blocks, as described in Section 2.2. Box (a) shows the schedule of entire CB blocks within the input matrices.

Box (b) shows the different components of the CB block within the memory hierarchy. Consider a CPU with p cores. To compute a single CB block on the CPU, the IO components are first loaded into the L3 cache. The A component is partitioned into p square $m_c \times k_c$ sub-matrices (i.e., $m_c = k_c$), and each sub-matrix is loaded into the L2 cache of one of the cores. The data component of size $k_c \times \alpha pm_c$ is then broadcast from the L3 cache to each core while the partial result component of size $p \cdot m_c \times \alpha \cdot p \cdot m_c$ is reused in the L3 cache. Note that shaping the CB block on the CPU as $pm_c \times k \cdot k_c \times \alpha pm_c$ with $k = 1$ is analogous to the $pk \times k \times \alpha pk$ shaping of the CB block in Section 3. Here, $\alpha > 1$ is a unitless constant that is set based on the available DRAM bandwidth . Partial results are returned to DRAM only after being reused for all accumulation in the K-dimension. Since we reuse partial results in the L3 cache until they are complete, the total IO for a CB block is the sum of the A and B component's IO:

$$IO = IO_A + IO_B = p \cdot m_c \cdot k_c + \alpha p \cdot m_c \cdot k_c$$

Computation time for CAKE is derived analogously to GOTO (Section 4.1). Box (e) in Figure 6 shows MM at the tile level on a CPU. We assume a single core can multiply a $m_r \times k_c$ tile with a $k_c \times n_r$ tile to produce a $m_r \times n_r$ result tile per cycle, This amounts to $m_r \cdot k_c \cdot n_r$ multiply-accumulate (MAC) operations per cycle. The total number of MAC operations needed to compute a $m_c \times \alpha \cdot p \cdot m_c$ result sub-matrix is $m_c \cdot k_c \cdot \alpha \cdot p \cdot m_c$. We can compute p of these $m_c \times \alpha \cdot p \cdot m_c$ result sub-matrices in parallel, so the CB block compute time is:

$$T = \frac{m_c \cdot k_c \cdot \alpha pm_c}{m_r \cdot k_c \cdot n_r} = \frac{\alpha \cdot p \cdot m_c^2}{m_r \cdot n_r}$$

The external memory bandwidth required to maximize computation throughput when increasing the number of cores is:

$$BW_{ext}^{CAKE} = \frac{IO}{T} = \frac{pm_c kc(\alpha+1) \cdot m_r n_r}{\alpha pm_c^2} = \frac{\alpha+1}{\alpha} \cdot m_r n_r \quad (4)$$

We see CAKE can, unlike GOTO, balance computation with IO, while requiring only constant external bandwidth.

As described in Section 3, when growing the number of cores by a factor of p, CAKE requires that local memory size increase by a factor of p^2 and internal bandwidth by a factor of p:

$$MEM_{local}^{CAKE} = IO_A + IO_B + IO_C = pm_c k_c \cdot (\alpha+1) + \alpha p^2 m_c^2 \quad (5)$$

$$BW_{int}^{CAKE} = \frac{IO_A + IO_B + 2IO_C}{T} = \left(2p + \frac{1}{\alpha} + 1\right) \cdot m_r n_r \quad (6)$$

4.3 Sizing CAKE CB Blocks to Minimize Cache Evictions

The CAKE CB block shape allows us to keep bandwidth requirements constant, but CB block size is still dependent on available local memory. Due to the least-recently-used (LRU) eviction scheme of our L2 and L3 caches, we are unable to use the entirety of each cache for matrix operands. CAKE CB blocks must be sized to minimize superfluous memory accesses resulting from cache evictions. GOTO uses a similar strategy for sizing its blocks to minimize translation lookaside buffer (TLB) misses [13].

We want to reuse a CB block surface (either input surfaces A and B, or partial result surface C) in local memory. Assume a current working CB block i reuses partial results $C[i]$ in local memory. If the CPU cache has size S and uses an LRU eviction protocol, we must size our CB block such that:

$$C + 2(A + B) \leq S$$

The factor of 2 allows entries for $A[i]$ and $B[i]$ to coexist with entries for $C[i]$, $A[i+1]$, and $B[i+1]$. Suppose we are currently working on CB block i and size the block such that $A + B + C = S$. When the next CB components $A[i+1]$ and $B[i+1]$ are first addressed, their data should replace the cache entries that were used for $A[i]$ and $B[i]$. However, upon completion of CB block i, some entries related to $C[i]$ will be LRU and will be replaced by $A[i+1]$ and $B[i+1]$ from block $i+1$. Furthermore, the factor of 2 ensures, when $A[i+2]$ and $B[i+2]$ of block $i+2$ are first addressed, the entries for $A[i]$ and $B[i]$ are guaranteed to be LRU and will be replaced.

4.4 Comparing CAKE and GOTO

CAKE and GOTO employ similar techniques for reusing data within a memory hierarchy, as seen in Figures 5 and Figures 6. Both use outer products to compute blocks of C by multiplying column sub-matrices of A with row sub-matrices of B and summing each partial product in the K dimension. Furthermore, both partition the column sub-matrix of A into square $m_c \times k_c$ sub-matrices and reuse these square sub-matrices in the L2 cache of each core. However, by using CB block shaping and sizing, CAKE is able to account for available external bandwidth constraints.

To accommodate additional cores, both CAKE and GOTO increase the size of the sub-matrices reused in local memory. Figures 5b and 6b) show that both CAKE and GOTO will increase the size of the column sub-matrix of A in the M dimension (via pm_c) by utilizing the L2 cache from each additional core to operate on several new $m_c \times k_c$ sub-matrices. In GOTO, the B sub-matrix occupies most of the L3 cache as the value of n_c is chosen based on L3 cache size. GOTO assumes that computation power and memory bandwidth are balanced, allowing wide column sub-matrices of C (with large n_c value) to be computed while buffering partial results in main memory, i.e., GOTO attempts to overlap partial result movement with the computation of the next outer product. Since the M-dimension increases proportionally with the number of cores, the partial result sub-matrix IO to main memory also increases proportionally. GOTO relies on increasing DRAM bandwidth to balance the computation time with such increasing IO requirements.

In contrast, CAKE controls the size of the B and C sub-matrices in the L3 cache by shaping the CB block in the N dimension (αpm_c) according to the available DRAM bandwidth (via α) and number of cores (via p, see Figure 6a). Unlike GOTO, CAKE eliminates the movement of partial results to external memory entirely through in-place accumulation in local memory.

Partial results of C in CAKE occupy the greatest proportion of L3 cache relative to the sub-matrices of A and B. For example, in our experiments with the Intel i9-10900K CPU, suppose $p = 10$ cores, $\alpha = 1$, and sub-matrix dimensions $m_c = k_c = 192$ are chosen such that the B and C CB block surfaces fill up the L3 cache. Then, the C and B surfaces will take up up 91% and 9% of the L3 cache, respectively. In comparison, GOTO uses all of the L3 cache for B.

Both CAKE and GOTO will see reduced performance as the number of cores grows faster relative to available internal memory size. Past a certain point, GOTO cannot leverage additional cores as DRAM bandwidth becomes the limiting factor. Analogously for CAKE, limited internal bandwidth and local memory will eventually impede performance as the number of cores increases. However, CAKE can further increase throughput by utilizing additional local memory or DRAM bandwidth whereas GOTO can only do so by increasing DRAM bandwidth usage.

Table 2: CPUs Used in CAKE Evaluation

CPU	L1	L2	L3	DRAM	Cores	DRAM Bandwidth
Intel i9-10900K	32 KiB	256 KiB	20 MiB	32 GB	10	40 GB/s
AMD Ryzen 9 5950X	32 KiB	512 KiB	64 MiB	128 GB	16	47 GB/s
ARM v8 Cortex A53	16 KiB	512 KiB	N/A	1 GB	4	2 GB/s

5 PERFORMANCE EVALUATION OF CAKE ON CPUS

Current state-of-the-art MM algorithms are based on GOTO and bounded by external memory bandwidth. We show CAKE improves MM performance on multiple CPU architectures, achieving maximum computation throughput without increasing external bandwidth by leveraging increased internal memory bandwidth and size.

We evaluate CAKE on CPUs with different system characteristic constraints. We measure CAKE's performance, using computation throughput and DRAM bandwidth usage as metrics.

5.1 CPUs Evaluated

We evaluate CAKE on an ARM v8 Cortex A53, Intel i9-10900K, and AMD Ryzen 9 5950X. Each CPU has unique architecture and system characteristics, shown in Table 2, allowing us to evaluate CAKE under different constraints in external bandwidth, local memory size, and internal bandwidth. The ARM v8 Cortex A53 is an embedded low-power CPU with severely limited DRAM bandwidth, local memory size, and internal bandwidth. The Intel i9-10900K is a 10-core desktop CPU with high DRAM bandwidth and large local memory but constrained internal memory bandwidth. The AMD Ryzen 9 5950X is a 16-core desktop CPU with high DRAM bandwidth, large local memory, and high internal memory bandwidth. In each section, we compare CAKE to the state-of-the-art GEMM library for its respective CPU. We then explain CAKE's performance in the context of each CPU's characteristics and constraints.

5.2 CAKE Implementation and CPU Experiments

We implemented CAKE in C++ for use in our evaluations. Our implementation uses the BLIS kernel library [28] to execute MM at the tile level on CPU SIMD registers. BLIS was chosen for its portability across CPU architectures, enabling CAKE to act as a drop-in GEMM library on multiple platforms.

In Section 5.2.2, we profile CAKE on the Intel i9 and ARM CPUs in terms of how often the CPUs are stalled on memory requests from different memory levels. Then, in Section 5.2.3 we show CAKE's performance relative to Intel's Math Kernel Library (MKL) and ARM Performance Libraries (ARMPL) for various matrix sizes on the Intel and ARM CPUs. Matrix size and shape were systematically varied in three dimensions to identify regions where CAKE outperforms the competing GEMM library. In Sections 5.2.4, 5.2.5, and 5.2.6, we vary the number of cores and measure computation throughput (in GFLOP/s) and DRAM bandwidth usage (in GB/s) when performing a large square MM computation. Matrix sizes were chosen to fit into the system's available DRAM. For example, we use relatively small 3000×3000 matrices for the ARM system due to its limited main memory. CAKE's theoretically optimal DRAM bandwidth usage (calculated in Section 4.2) is shown as a dashed curve. Internal bandwidths between the last level cache (LLC) and CPU cores were measured using the parallel memory bandwidth benchmark tool (*pmbw*) [5]. **Local memory refers to the LLC** shared among all cores, and may be the L2 or L3 cache depending on architecture (i.e., L2 for ARM and L3 for Intel). We also show performance extrapolations for CAKE and the state-of-the-art GEMM libraries when using additional CPU cores, assuming fixed DRAM bandwidth. The extrapolation assumes local memory size increases quadratically and internal bandwidth increases linearly with the number of cores. We use the last two data points in each plot to initialize the extrapolation line.

5.2.1 *Packing Overhead and Considerations.*

GEMM libraries, such as Intel MKL, contain distinct packing implementations optimized for many matrix shapes [16] to copy input matrices into a contiguous buffer. We implemented only one packing method since our development effort focused on the MM kernel over packing. By packing matrices into contiguous buffers, GEMM libraries, including CAKE, are able to minimize cache evictions. As an added benefit, packing also prevents cache self-interference, as articulated in [22]. In all experiments, **we include packing overhead** when measuring throughput and DRAM bandwidth during MM.

In a typical system, cache eviction schemes are not modifiable by users. If the eviction scheme is not properly accounted for, additional runtime due to unnecessary evictions and cache misses will dominate GEMM running time. When M, N, and K are large, time spent packing is small relative to time spent in the MM kernel. However, we note that, with CAKE, packing overhead for skewed matrix shapes (i.e., where one of M, N, and K is much smaller than the other two) may constitute a significant fraction of total computation time.

5.2.2 *Reducing Main Memory Stalls With CAKE.*

We profile the time the CPU spends serving requests from different memory levels when running a fixed-size MM using CAKE, MKL, and ARMPL libraries. On the Intel i9-10900K, we profile memory request stalls for an MM between two 10000×10000 matrices. A stall occurs when CPU cores are waiting for data from some memory level, and are unable to be productive. We use Intel's Vtune Profiler [8] to measure the number of cycles where the CPU is stalled on the L1, L2, and L3 caches as well as main memory. On ARM v8 Cortex A53, we measure the number of cache hits and DRAM memory requests using the Linux Perf Tool [2] for an MM between two 3000×3000 matrices.

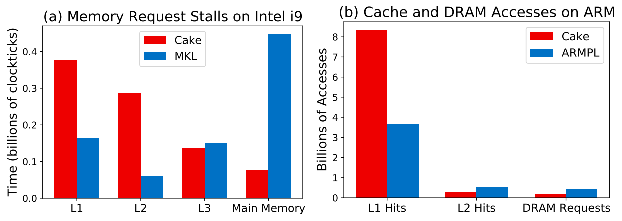

Figure 7: (a) Number of clock ticks spent stalled on requests at different memory levels for CAKE and MKL for 10000×10000 matrices using all 10 cores of the Intel i9-10900K CPU. Stalled time is defined as the time that the operands are not in the CPU registers but instead are present in a particular memory level (L1, L2, L3, DRAM). For this large problem size, CAKE's performance is $\approx 10\%$ lower than MKL's (see Figure 8 for problem sizes where CAKE outperforms MKL). However, CAKE spends less absolute time stalled on main memory and more time stalled on local memory than MKL. (b) Cache Hits and DRAM Memory accesses when performing MM between two 3000×3000 matrices on ARM v8 Cortex. CAKE shifts memory demand to internal memory (L1, L2, L3) whereas both MKL and ARMPL rely on main memory transfers.

As depicted in Figure 7a, with CAKE, the Intel CPU is most often stalled on memory requests to a local memory level. In contrast, with MKL, the CPU stalls most often on requests to main memory. We see similar results for the ARM CPU in Figure 7b. CAKE serves more memory requests from the L1 cache than ARMPL, which performs $\approx 2.5x$ more DRAM requests. This reflects CAKE's use of local memory to buffer partial results. If DRAM bandwidth remains fixed, as the number of cores grows, main memory stalls will significantly impact performance due to high memory access latencies.

5.2.3 *CAKE's Performance for Different Matrix Shapes.*

As depicted in Figure 8, we use all 10 cores for the Intel CPU and vary M and K from 1 to 8000. Each plot shows a different $M : N$ aspect ratio, and shaded regions represent input matrix dimensions where CAKE outperforms MKL by at least some factor.

As the matrix size decreases in any dimension, CAKE's throughput, relative to MKL, increases. For a general MM between square $N \times N$ matrices, arithmetic intensity is $O(N)$. Hence, as the problem size N decreases, arithmetic intensity also decreases and the MM becomes more memory-bound. CAKE increases arithmetic intensity, and thus MM throughput, by analytically blocking the computation to minimize external IO.

Figure 9 shows CAKE's speedup in computation throughput for square matrices of varying size compared to MKL and ARMPL on the Intel i9-10900K and ARM v8 Cortex CPUs, respectively. Speedup in throughput for a fixed-size MM using p cores is defined as t_p/t_1, where t_1 and t_p are the throughputs for the MM using a single core and p cores, respectively. This speedup metric allows us to evenly compare CAKE's performance across different platforms (i.e., Intel and ARM) for the same problem sizes. For absolute performance results, please refer to Sections 5.2.4, 5.2.5, and 5.2.4.

In Figure 9a, CAKE's improvement in speedup over MKL is more pronounced for smaller matrices. When matrices grow to a certain size, achievable throughput reaches a steady state, allowing MKL to approach CAKE's performance. In contrast, Figure 9b shows that, on the ARM CPU, CAKE has higher speedup than ARMPL for all problem sizes. The low available DRAM bandwidth impedes ARMPL's performance when it attempts to use more cores, meaning CAKE outperforms ARMPL for all problem sizes. Kinks in the speedup curves for MKL and CAKE in Figure 9a around 6 cores may be attributable to non-deterministic latency introduced by bus contention or L3 cache coherency delays.

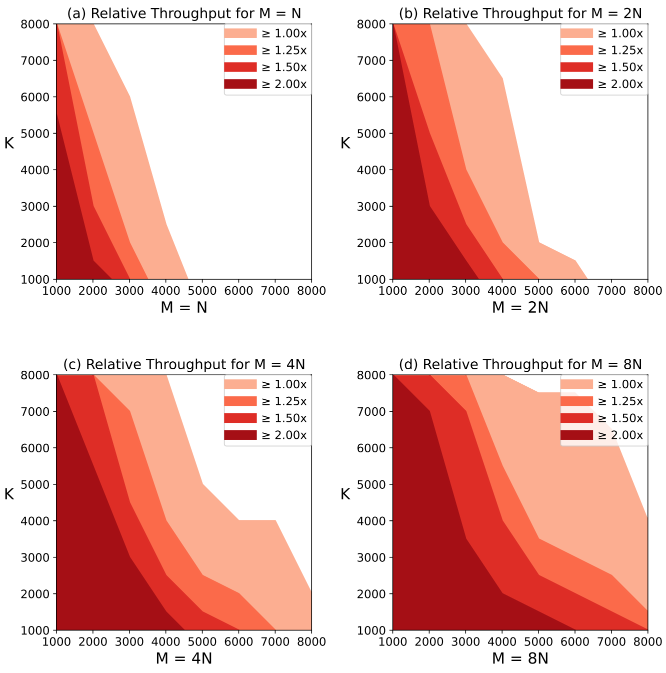

Figure 8: Relative improvement in throughput for CAKE, compared to MKL, over various matrix dimensions on an Intel i9-10900K. Each each contour boundary indicates dimensions for which CAKE outperforms MKL by a certain factor. Darker regions indicate greater throughput for CAKE relative to MKL.

*5.2.4 **Achieving Maximum Computation Throughput Allowed Under External Bandwidth Constraints**.* We evaluate the CAKE algorithm on an ARM v8 Cortex A53, which has a low DRAM bandwidth, local memory size, and internal bandwidth. We multiply two 3000×3000 matrices using CAKE and the ARMPL and collect performance data using the Linux perf tool [2]. We use Perf to record DRAM accesses by monitoring the ARM PMU event counter for L2 cache refills from DRAM. Figures 11a and 11b show, as the number of cores increases, ARMPL must increase DRAM usage to increase computation. As a result, ARMPL does not achieve

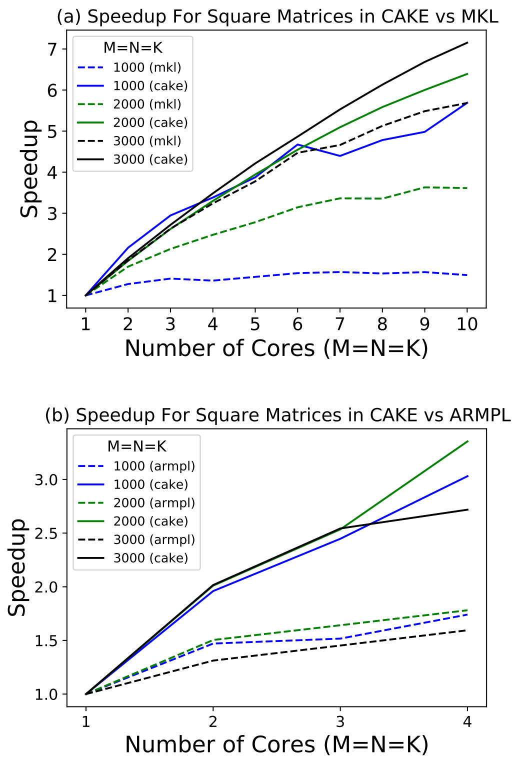

Figure 9: Speedup in computation throughput (speedup in time compared to a single core) for square matrices on Intel i9 10900K and ARM v8 Cortex CPUs. In (a), CAKE's speedup improvement over MKL is more pronounced for small matrices. MKL's performance approaches CAKE as matrix size increases and achievable throughput reaches a steady state. In (b), CAKE outperforms ARMPL on square MM consistently for all sizes. Unlike CAKE, ARMPL cannot scale performance with the number of cores due to limited DRAM BW on the ARM CPU.

the maximum computation throughput as there is little additional DRAM bandwidth available (Figure 11b). In contrast, CAKE can adjust the CB block shape to achieve the maximum computation throughput for a given number of cores without increasing DRAM bandwidth.

We use *pmbw* to measure the ARM CPU's internal bandwidth between the L2 cache and cores, as shown in Figure 11c. Internal bandwidth on the ARM CPU does not increase proportionally with the number of cores, beyond 2 cores. Consequently, we see the DRAM bandwidth required for CAKE increases slightly above the theoretical optimum for 3 and 4 cores in Figure 11a.

In Figure 11b, insufficient internal bandwidth causes CAKE's throughput to grow slightly less than proportional to the number of cores. Our extrapolation in Figure 11c suggests that, if internal bandwidth continued to increase proportionally to the number of cores, CAKE throughput would also continue to increase.

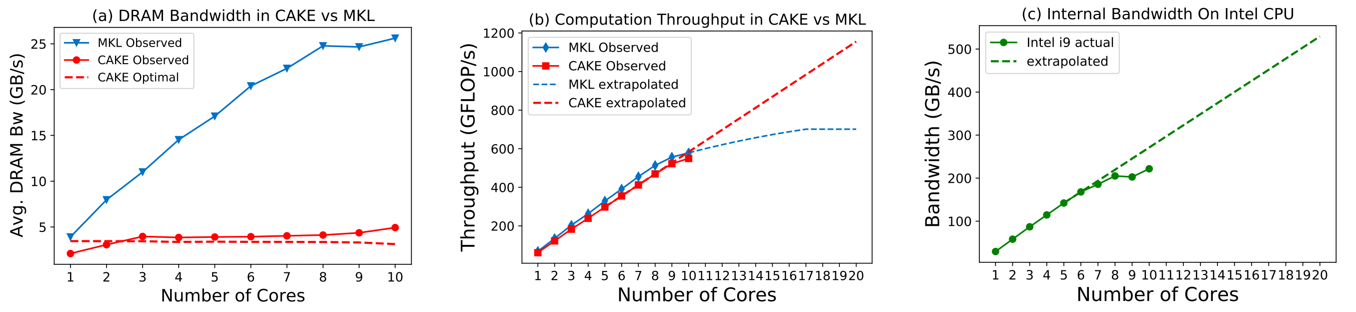

Figure 10: CAKE on an Intel i9-10900k, performing MM between two 23040×23040 matrices. (a) Compared to MKL (blue), CAKE (red) does not need to increase DRAM bandwidth to utilize more cores. The slight increase in CAKE's DRAM bandwidth, above the optimal, for 9 and 10 cores is caused by insufficient internal bandwidth between the L3 cache and CPU cores. (b) CAKE (red), achieves within 3% of MKL's (blue) computation throughput without needing to increase DRAM bandwidth. (c) Using *pmbw*, we measure internal bandwidth between the L3 cache and cores. On this CPU, internal bandwidth does not increase proportionally with the number of cores past 6 cores. Consequently, we see in (a) that CAKE's required DRAM bandwidth increases slightly above optimal past 6 cores, and in (b) that insufficient internal bandwidth causes CAKE throughput to grow slightly less than proportional to core count. The dotted extrapolation lines assume internal memory bandwidth increases proportionally for each additional core, local memory size increases quadratically, and DRAM bandwidth is fixed.

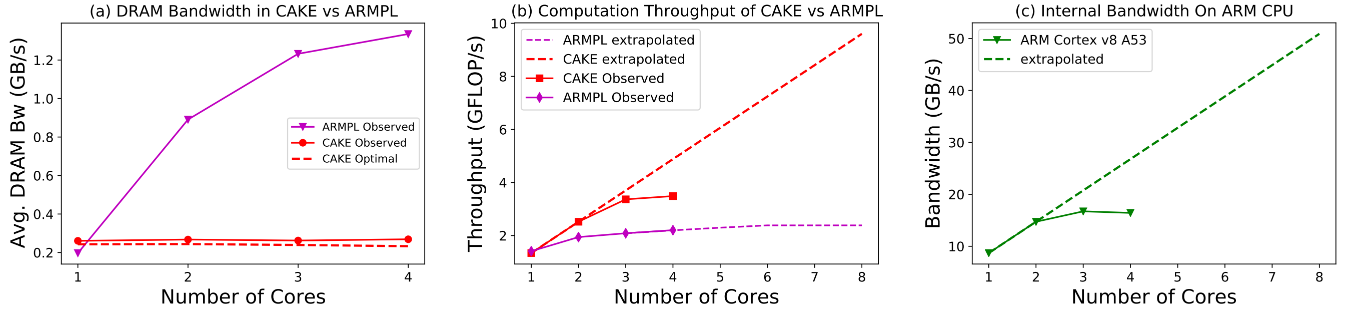

Figure 11: CAKE on an ARM v8 Cortex A53, performing MM between 3000×3000 matrices. CAKE adjusts the CB block to achieve the maximum computation throughput without increasing DRAM bandwidth. (a) Unlike ARMPL (magenta), CAKE (red) does not need to increase DRAM bandwidth to utilize more cores. (b) CAKE achieves higher computation throughput than ARMPL, which is limited by DRAM bandwidth. (c) Using *pmbw*, we find internal bandwidth between the L2 cache and cores does not increase with the number of cores for the ARM CPU. Consequently, we see in (a) that CAKE's required DRAM bandwidth increases slightly above optimal for 3 and 4 cores, and in (b) that insufficient internal bandwidth causes CAKE's throughput to grow slightly less than proportional to the number of cores. The dotted extrapolation lines assume internal memory bandwidth increases proportionally for each additional core, local memory size increases quadratically, and DRAM bandwidth is fixed.

5.2.5 ***Achieving Maximum Computation Throughput Allowed Under Internal Memory Bandwidth Constraints.*** We compare CAKE to MKL on an Intel i9-10900K for an MM between two 23040×23040 matrices. We use Intel VTune Profiler to record computation throughput and DRAM bandwidth usage. CAKE achieve within 3% of MKL's computation throughput, but requires substantially less DRAM bandwidth (Figures 10a, 10b).

We measure internal bandwidth between the L3 cache and processor cores, again using *pmbw*. As shown in Figure 10c, the Intel CPU is constrained by internal bandwidth, which fails to grow proportionally with the number of cores beyond 6 cores. Consequently, CAKE requires slightly more than the theoretically optimal DRAM bandwidth for 9 and 10 cores, as seen in Figure 10a. In Figure 10b, we also see that insufficient internal bandwidth causes CAKE's throughput to grow less than proportional to the number of cores past 8 cores. Figure 10b includes extrapolations for expected performance for both for CAKE and MKL, under the assumption that internal memory bandwidth continues to increase proportionally with the number of cores while DRAM bandwidth remains fixed. With sufficient local memory, CAKE will achieve the maximum possible computation throughput for a given number of cores. While MKL's throughput may continue to increase beyond 10 cores, it relies on increased DRAM bandwidth to increase throughput, which

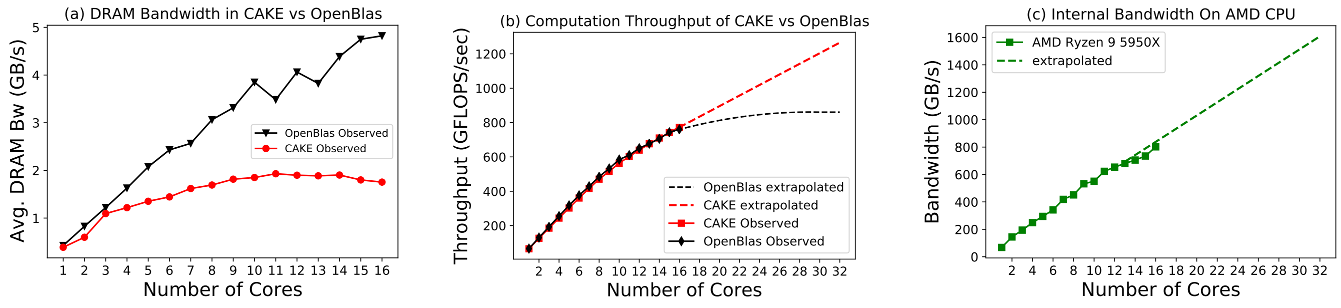

Figure 12: CAKE on an AMD Ryzen 9 5950X performing MM between two 23040×23040 matrices. Relative to the other CPUs evaluated, the 5950X is least constrained by system resources. (a) Unlike OpenBLAS (black), CAKE (red) does not need to increase DRAM bandwidth to utilize more cores. On this CPU, DRAM bandwidth is estimated from the number of L1 cache-line refills from DRAM. (b) CAKE matches OpenBLAS's peak performance, but with a smaller DRAM bandwidth requirement, as seen previously. (c) Using *pmbw*, we measured internal bandwidth between the L3 cache and cores. The 5950X has high internal bandwidth between L3 cache and cores that grows roughly linearly with the number of cores. Consequently, we see in (a) that CAKE's required DRAM bandwidth stays constant past 9 cores, and in (b) that sufficient internal bandwidth enables CAKE to achieve high computation throughput. The dotted extrapolation lines assume internal memory bandwidth increases proportionally for each additional core, local memory size increases quadratically, and DRAM bandwidth is fixed.

will plateau when all available DRAM bandwidth is consumed. Although 40 GB/s of DRAM bandwidth is available on this system, CAKE only utilizes an average of 4.5 GB/s, instead relying on local memory bandwidth to leverage more cores. Relying on local memory is preferable because, while DRAM bandwidth is high, latency and power consumption may be problematic [29].

5.2.6 *Achieving Maximum Computation Throughput Allowed Under Available Processing Power (without other system resource constraints)*.

We compare CAKE to OpenBLAS when computing an MM between two large 23040×23040 matrices on an AMD Ryzen 5950X CPU. OpenBLAS was chosen for the AMD CPU because it is an optimized library using GotoBLAS and outperforms MKL on non-Intel hardware [9]. We use AMD μProf [15] to measure DRAM bandwidth and computation throughput. However, due to the lack of an available DRAM access counter on this CPU at the time of writing, DRAM accesses during MM are estimated using the PMU counter for L1 data cache refills. Only a portion of total DRAM bandwidth usage is measured, so we omit comparison to the theoretically optimal DRAM bandwidth curve in Figure 12a.

Figure 12c shows internal bandwidth between the L3 cache and processor cores increases roughly proportionally by 50 GB/s per core. CAKE takes advantage of the increasing internal bandwidth to achieve peak computation throughput without increasing DRAM bandwidth usage beyond 9 cores (Figures 12a and 12b). Since this CPU is not constrained by internal bandwidth or DRAM bandwidth, both CAKE and OpenBLAS can increase computation throughput when adding more cores, but OpenBLAS uses more DRAM bandwidth than CAKE.

We include extrapolations for expected performance for CAKE and OpenBLAS, again assuming internal memory bandwidth continues to increase proportionally with the number of cores while DRAM bandwidth remains fixed in Figures 12b and 12c

6 EXTENDING CAKE BEYOND CPUS AND CAKE ARCHITECTURE SIMULATOR

6.1 Beyond CPUs

CAKE trades off increased internal memory usage for reduced external bandwidth requirements, which can benefit any computing platform with a memory hierarchy with multiple levels (e.g., L1, L2, DRAM).

While this paper focuses on CPUs, CAKE is not limited to these platforms. For example, the CAKE methodology can apply to GPUs or other heterogeneous systems comprised of general-purpose computing engines and special-purpose accelerators.

Indeed, CUTLASS, a GPU MM library, similarly utilizes block shaping and sizing, but only exposes them as "compile-time constants that the programmer specifies." [1] CAKE's CB blocks can eliminate the need to manually search for optimal block designs. For similar reasons, CAKE can also help reduce searches for optimal multi-tenant schedules [11].

6.2 CAKE Architecture Simulator

We developed a SystemC architecture simulator [3] using MatchLib [19] connections to validate the correctness of the CB block design and execution schedule. The simulator models timings between external memory, local memory, and cores under various system characteristics (e.g., low external memory bandwidth). This flexibility helps verify the correctness of the CAKE algorithm for corner cases that are difficult to analyze. Results from the simulator were used to develop the libraries evaluated in Section 5 and will ease the development of future extensions of CAKE (as noted in Section 6.1) to various computing platforms (including GPUs, Maestro [21], TPU [17], etc.).

To reduce module complexity and simplify programming, standardized packets are used for all communication between simulated hardware modules. Packets originate from external memory and contain headers to control routing (i.e., source routing) as well as fields containing the packet's tile index into the computation space and CB block. Packet-based scheduling allows us to easily modify the architecture and computation schedule. For example, to double the number of cores, we simply instantiate new modules to represent the added cores, and adjust the packet headers accordingly.

7 CONCLUSION

When performing a block of computations in a matrix multiplication (MM) space from within an internal memory, CAKE uses a constant-bandwidth (CB) block to simultaneously satisfy two objectives. First, CAKE adapts the computation's arithmetic intensity to balance computation and IO times by controlling block aspect ratios. Second, CAKE adjusts the computation volume by controlling CB block sizes to fully use all available computing power. With CAKE, we can trade off increased internal memory usage for reduced external bandwidth requirements. From a technology viewpoint, relying on local memory is generally preferable since DRAM has relatively high latency and power consumption. Furthermore, CAKE explicitly specifies the required bandwidth and size of the internal memory.

Supported by analysis and measurements on real-world systems, CAKE can substantially improve computational throughput for MM on CPUs as a drop-in replacement for MM calls. In both theory and practice, CAKE has advanced state-of-the-art MM computation.

ACKNOWLEDGMENTS

This work was supported in part by the Air Force Research Laboratory under award number FA8750-18-1-0112, a gift from MediaTek USA and a Joint Development Project with TSMC. We thank Wei-Te Mark Ting for his comments and drawings. We also thank anonymous reviewers whose comments helped improve and clarify this manuscript.

REFERENCES

[1] 2020. CUTLASS: Fast Linear Algebra in CUDA C++. https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda/.

[2] 2021. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page.

[3] Accellera Systems Initiative Inc. 2016. *SystemC Synthesis Subset Standard v1.4.7.* Accellera Systems Initiative Inc. https://www.accellera.org/downloads/standards/systemc

[4] Arm Limited 2021. *Arm Performance Libraries Reference Guide.* Arm Limited. https://developer.arm.com/documentation/101004/latest/

[5] Timo Bingmann. 2013. Parallel Memory Bandwidth Benchmark/Measurement. https://panthema.net/2013/pmbw/.

[6] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh, D. McCaule, P. Morrow, D. W. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. Shen, and C. Webb. 2006. Die Stacking (3D) Microarchitecture. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 469–479. https://doi.org/10.1109/MICRO.2006.18

[7] Intel Corporation. 2021. Intel oneAPI Math Kernel Library. https://software.intel.com/content/www/us/en/develop/articles/mkl-reference-manual.html.

[8] Intel Corporation. 2021. Intel VTune Profiler. https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html.

[9] Agner Fog. 2020. Intel's "cripple AMD" function. https://www.agner.org/optimize/blog/read.php?i=49.

[10] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. 2017. TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. 751–764.

[11] S. Ghodrati, B. H. Ahn, J. Kyung Kim, S. Kinzer, B. R. Yatham, N. Alla, H. Sharma, M. Alian, E. Ebrahimi, N. S. Kim, C. Young, and H. Esmaeilzadeh. 2020. Planaria: Dynamic Architecture Fission for Spatial Multi-Tenant Acceleration of Deep Neural Networks. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 681–697. https://doi.org/10.1109/MICRO50266.2020.00062

[12] Kazushige Goto and Robert A. van de Geijn. 2002. On Reducing TLB Misses in Matrix Multiplication. *Technical Report TR-02-55* (2002).

[13] Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of High-Performance Matrix Multiplication. *ACM Trans. Math. Softw.*, Article 12 (2008), 25 pages.

[14] J.H. Huang. 2013. Keynote. (2013). NVidia GPU Technology Conference.

[15] Advanced Micro Devices Inc. 2021. AMD μProf. https://developer.amd.com/amd-uprof/.

[16] Intel Corporation 2016. *Intel oneAPI Deep Neural Network Library.* Intel Corporation. https://oneapi-src.github.io/oneDNN/

[17] Norman P. Jouppi, Cliff Young, et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. *CoRR* abs/1704.04760 (2017). arXiv:1704.04760 http://arxiv.org/abs/1704.04760

[18] H. Jun, J. Cho, K. Lee, H. Son, K. Kim, H. Jin, and K. Kim. 2017. HBM (High Bandwidth Memory) DRAM Technology and Architecture. In *2017 IEEE International Memory Workshop (IMW)*. 1–4. https://doi.org/10.1109/IMW.2017.7939084

[19] Brucek Khailany, Evgeni Khmer, et al. 2018. A Modular Digital VLSI Flow for High-Productivity SoC Design. In *Proceedings of the 55th Annual Design Automation Conference* (San Francisco, California) *(DAC '18)*. Association for Computing Machinery, New York, NY, USA, Article 72, 6 pages. https://doi.org/10.1145/3195970.3199846

[20] Ashish Kumar, Saurabh Goyal, and Manik Varma. 2017. Resource-Efficient Machine Learning in 2 KB RAM for the Internet of Things. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70* (Sydney, NSW, Australia) *(ICML'17)*. JMLR.org, 1935–1944.

[21] H. T. Kung, B. McDanel, S. Q. Zhang, X. Dong, and C. C. Chen. 2019. Maestro: A Memory-on-Logic Architecture for Coordinated Parallel Use of Many Systolic Arrays. In *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Vol. 2160-052X. 42–50.

[22] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. 1991. The Cache Performance and Optimizations of Blocked Algorithms. (1991), 63–74. https://doi.org/10.1145/106972.106981

[23] Z. Li, Z. Wang, L. Xu, Q. Dong, B. Liu, C. I. Su, W. T. Chu, G. Tsou, Y. D. Chih, T. Y. J. Chang, D. Sylvester, H. S. Kim, and D. Blaauw. 2021. RRAM-DNN: An RRAM and Model-Compression Empowered All-Weights-On-Chip DNN Accelerator. *IEEE Journal of Solid-State Circuits* 56, 4 (2021), 1105–1115.

[24] Ji Lin, Wei-Ming Chen, Yujun Lin, John Cohn, Chuang Gan, and Song Han. 2020. MCUNet: Tiny Deep Learning on IoT Devices. arXiv:2007.10319 [cs.CV]

[25] Jason Lowe-Power, Mark D. Hill, and David A. Wood. 2016. When to use 3D Die-Stacked Memory for Bandwidth-Constrained Big Data Workloads. arXiv:1608.07485 [cs.AR]

[26] Bradley McDanel, Sai Qian Zhang, HT Kung, and Xin Dong. 2019. Full-stack optimization for accelerating CNNs using powers-of-two weights with FPGA validation. In *Proceedings of the ACM International Conference on Supercomputing*. 449–460.

[27] Dennis Rich, Andrew Bartolo, Carlo Gilardo, Binh Le, Haitong Li, Rebecca Park, Robert Radway, Mohamed Sabry, H.-S. Philip Wong, and Subhasish Mitra. 2020. *Heterogeneous 3D Nano-systems: The N3XT Approach?* 127–151. https://doi.org/10.1007/978-3-030-18338-7_9

[28] Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Trans. Math. Software* 41, 3 (June 2015), 14:1–14:33. http://doi.acm.org/10.1145/2764454

[29] T. Vogelsang. 2010. Understanding the Energy Consumption of Dynamic Random Access Memories. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. 363–374. https://doi.org/10.1109/MICRO.2010.42

[30] Wm. A. Wulf and Sally A. McKee. 1995. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News* 23, 1 (March 1995), 20–24. https://doi.org/10.1145/216585.216588

[31] Zhang Xianyi, Wang Qian, and Zaheer Chothia. 2012. Openblas. *URL: http://xianyi.github.io/OpenBLAS* (2012), 88.