Hardware Accelerator Integration Tradeoffs for High-Performance Computing: A Case Study of GEMM Acceleration in N-Body Methods

Mochamad Asri*, Dhairya Malhotra†, Jiajun Wang*, George Biros‡, Lizy K. John* and Andreas Gerstlauer*
*Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX, USA
†Flatiron Institute, New York, NY, USA
‡Institute for Computational Engineering and Sciences, The University of Texas at Austin, Austin, TX USA
asri@utexas.edu, dmalhotra@flatironinstitute.org, jiajunwang@utexas.edu, gbiros@acm.org, {ljohn,gerstl}@ece.utexas.edu

Abstract—In this paper, we study performance and energy saving benefits of hardware acceleration under different hardware configurations and usage scenarios for a state-of-the-art Fast Multipole Method (FMM), which is a popular N-body method. We use a dedicated Application Specific Integrated Circuit (ASIC) to accelerate General Matrix-Matrix Multiply (GEMM) operations. FMM is widely used in applications and is representative example of the workload for many HPC applications. We compare architectures that integrate the GEMM ASIC next to, in or near main memory with an on-chip coupling aimed at minimizing or avoiding repeated round-trip transfers through DRAM for communication between accelerator and CPU. We study tradeoffs using detailed and accurately calibrated x86 CPU, accelerator and DRAM simulations.

Our results show that simply moving accelerators closer to the chip does not necessarily lead to performance/energy gains. We demonstrate that, while careful software blocking and on-chip placement optimizations can reduce DRAM accesses by 2X over a naive on-chip integration, these dramatic savings in DRAM traffic do not automatically translate into significant total energy or runtime savings. This is chiefly due to the application characteristics, the high idle power and effective hiding of memory latencies in modern systems. Only when more aggressive co-optimizations such as software pipelining and overlapping are applied, additional performance and energy savings can be unlocked by 37% and 35% respectively over baseline acceleration. When similar optimizations (pipelining and overlapping) are applied with an off-chip integration, on-chip integration delivers up to 20% better performance and 17% less total energy consumption than off-chip integration

1 Introduction

The end of Moore's law era means that single-socket, general-purpose hardware is quickly approaching its limits. A possible path to increasing single-socket performance is to consider ASIC-based special-purpose hardware (hereby an "accelerator"). By sacrificing programmability, accelerators provide the best performance with respect to time-to-solution and energy consumption.

However, most of the existing research on ASIC-based accelerators has focused on individual computational kernels [12], [11], [42], [41], [27], [35]. In practice, many applications comprise several compute and data transformation kernels (e.g., scatter/gather, transpose, and permutation operations), out of which selected kernels such as General Matrix-Matrix Multiplies (GEMMs) and Fast Fourier Transforms (FFTs) are offloaded to an accelerator.

A key concern in system-level design is the architectural integration of the accelerator with the host system. Where in the memory hierarchy do we place the accelerator in order to reduce data movement and communication overhead? Traditionally, accelerators have been integrated as off-chip devices placed near DRAM. This requires all data to be exchanged through DRAM. Such off-chip data movement can negate performance benefits of acceleration and can account for up to 50% of dynamic access energy in HPC applications [14]. More recently, accelerators have been moved in or near main memory [3], [50], but this only affects how efficiently the accelerator itself can access memory. To minimize or completely avoid off-chip DRAM transfers, accelerators can instead be integrated on the same die and coupled by sharing the last-level cache (LLC) with CPUs [32], [48], [1]. However, such a design also has its own shortcomings. If data exceeds LLC capacity, expensive spills to DRAM will still take place—depending on application locality patterns. Overall, deciding on whether on-chip or off-chip placement is better depends strongly on application and architecture interactions.

In this paper, we present a comprehensive algorithm/architecture co-design case study for a complex HPC application consisting of multiple interdependent kernels executing on a host CPU assisted by a custom hardware accelerator. We quantify, analyze, and explain the impact of joint accelerator integration and placement, algorithm/architecture and hardware/software co-optimizations on a representative HPC code both from a performance and energy perspective. Specifically, we study these tradeoffs for an application that consists of multiple GEMM calls that are combined with complex data-structure accesses and memory reshuffling. This workload is common in many HPC applications but as an example we study the Fast Multipole Method (FMM). FMM approximates a dense, $O(N^2)$ matrix-vector multiplication by a sparse, $O(N)$ matrix-vector multiplication. FMM belongs to the broad class of N-body methods used in computational physics and machine learning [17], [6], [38], [31], [20]. The fraction spent in N-body solvers varies from 100% in electromagnetic and acoustic scattering, to 80% in astrophysics and complex fluid simulations, to less than 50% in molecular dynamics.

We consider the near-interaction computation in volume FMM (a particular FMM variant), referred to as the ***"U-list"***. The U-list computation is a sparse matrix-vector multiplication computed by traversing a non-uniform octree to setup a series of GEMMs of various sizes, interleaved

with data reshuffling, and matrix-matrix accumulation. In well-optimized FMM codes the U-list accounts for 50% of execution time. The remaining time goes to far-interactions, which have the same workload pattern [37]. Indeed, this pattern—GEMMs interleaved with data-transformations—is prominent in HPC applications, e.g., N-body methods, high-order finite element methods [7] and hierarchical matrices [49]. By varying a parameter that controls the size of the GEMMs in the U-list, we can model workloads that are both compute- and memory-bound.

We accelerate the GEMM operations in the U-list using a dedicated ASIC, the Linear Algebra Processor (LAP) from [41] as an example of a typical GEMM accelerator. We chose the LAP for our studies since it represents an extreme case of acceleration that amplifies the impact of placement and data movement tradeoffs. Compared to modern GPUs, the LAP delivers a similar performance while having 7-10x better energy efficiency [41]. Note that our primary goal is to study relative accelerator placement effects/tradeoffs in terms of total system energy and runtime, not absolute benefits or comparison of specific acceleration options. As such, while quantitative results will vary, we expect the main qualitative placement insights to be independent of and to transfer to other accelerators or system configurations.

To study placement tradeoffs, we first develop a generic performance model. Using this model, we perform design space explorations across different application and architecture parameters. We complement analytical studies with simulations using the extended MARSSx86 system simulator from [5], which integrates a LAP and is calibrated to closely match the performance of modern x86 machines for faithful heterogeneous system studies. We extend this simulator with detailed energy [33] and DRAM [43] models. Using this setup, we study different architecture variants by integrating the U-list with GEMM on the LAP, and we test performance and energy for U-list without LAP, U-list with a LAP next to or in DRAM, and U-list with on-chip coupling of the LAP at the LLC. We combine architecture with algorithm co-optimizations by proposing software modifications for data blocking, pipelining and overlapping of accelerator computations.

Our results show that simply integrating accelerators onto the chip does not necessarily lead to performance/energy wins. For the FMM application we evaluate in our study, naive on-chip integration of a hardware accelerator alone can reduces DRAM accesses by up to 26% compared to an off-chip integration. With careful algorithm co-optimizations, DRAM accesses can be reduced further by up to 2x for a total of 3x savings. However, even after carefully blocking GEMM operations to fit into LLCs, there are only minor performance and energy variations over a naive off-chip integration. This is due to the compute-dominated nature of GEMM operations and performance otherwise being determined by base memory overhead. We similarly observed that when taking static memory idle, leakage and refresh power into account, the large reductions in dynamic DRAM accesses do not translate into significant total energy savings. However, when more aggressive co-optimizations such as software pipelining and overlapping with on-chip accelerator coupling are applied, further runtime and energy savings of 37% and 35% are achieved, respectively. When similar optimizations are employed in off-chip integration, on-chip integration offers up to 20% better performance with 17% less total energy consumption. As such, we observe that on-chip integration offers more opportunities for runtime and energy gains. However, depending on application characteristics and parameters, overall performance and energy are in other cases comparable across off- or on-chip couplings.

In summary, we make the following contributions:

- We study accelerator integration tradeoffs for a representative N-body code comprehensively under consideration of different application and architecture scenarios, hardware/software co-optimizations, problems sizes and FMM parameters, which allow us to simulate both compute-bound and memory-bound workloads.
- We co-optimize architectures and algorithms by developing novel accelerator-aware hardware/software coupling and algorithm optimizations.
- We quantify, analyze and identify detailed acceleration benefits, tradeoffs and limitations in the context of the FMM U-list and the LAP ASIC using both analytical performance models as well as simulations.
- To the best of our knowledge, this is the first comprehensive co-design study of actual system-wide hardware acceleration benefits and tradeoffs under different accelerator integration/placement, architecture and software optimizations, specifically for custom hardware FMM acceleration.

The rest of the paper is organized as follows: after related work discussion in Section 2, we provide a background on FMM and the LAP accelerator in Section 3. We then introduce generalized application and architecture models as well as HW/SW co-optimizations in Sections 4 through 6. Using analytical performance models, we present projections and an exploration of different application/architecture scenarios in Section 7. Finally, we demonstrate a comprehensive simulation-based performance and energy evaluation in Section 8, followed by a summary and outlook in Section 9.

2 Related Work

The demand for increased system specialization and heterogeneity originated as a response to the end of traditional semiconductor scaling [18], [47]. To date, heterogeneous computing has received steady attention both in academia [11], [42], [23], [54] and industry [29], [1]. As one type of specialization, custom hardware accelerators and accelerator-rich system designs promise order of magnitude performance and power benefits beyond traditional CPUs and GPUs. Accelerators are often designed as standalone components that communicate with the rest of the system through some interface [27], [35]. While a large amount of accelerator research has focused on the performance and energy improvements [47], [11], [23], [41], system-level integration and implications have received less consideration.

Initially, such ASICs were envisioned as external devices located in a separate chip different from the host system. As

such, any data transfer has to be exchanged through DRAM. To address the overhead of off-chip memory accesses, researchers have proposed methods for hiding communication latency using overlapped computations, e.g. via special direct memory access (DMA) [51], [44], [26] and DMA pipelining mechanisms [41]. More recently, processing-in-memory (PIM) architectures have gained renewed attention [34]. With advancements in 3D technology, integration of general-purpose or specialized compute units into the logic layer of 3D stacks has been investigated to exploit massive memory bandwidth at low energy consumption. While such approaches in general help move accelerator computation closer to memory, not every application pattern can benefit. In particular, most PIM work assume limited to no data sharing between the PIM accelerators and the host [3], [50]. By contrast, in domains where data is frequently shared and exchanged, the potential benefit of PIM acceleration is limited and unclear.

Alternatively, we can place the accelerator on the chip. To optimize on-chip cache-level integration, caching and cooperative prefetching policies have been proposed [32], [48]. Other approaches allow accelerator local memories to become part of the shared cache substrate, thereby extending LLC capacity when accelerators are not in use [15]. Within this context, PIM-like approaches closely couple accelerators with on-chip caches [2], but such architectures are again targeted at offloading operations that manipulate data already residing in caches.

In all cases, existing work is limited to hardware- and architecture-managed integration issues. There is no existing work that evaluates and compares actual system-wide benefits and tradeoffs of different on-/off-chip accelerator couplings. Our work is the first to study and demonstrate cross-layer acceleration tradeoffs from a system-level and hardware/software perspective, specifically in the context of HPC applications. Furthermore, not only do we study hardware-centric optimizations, but we also propose software modifications to fully exploit performance and energy opportunities of acceleration. Our results show that for real applications, careful architecture and software co-optimization effort is required in order to unlock on-chip integration gains. We do not discuss the vast literature on HPC applications on GPU-accelerated systems. Again, our primary goal is to study relative accelerator placement effects/tradeoffs in terms of total system energy and runtime, not advocating for absolute benefits or comparison of specific acceleration options. For a review focused on energy consumption see [39].

Previous work have investigated FMM acceleration opportunities using GPUs and FPGAs [53], [16], [24], [9]. Examples of GPUs and FMM coupling (and an energy consumption model) include [10] and [14]. Special-purpose systems for N-body codes include [40], [45]. More relevant to the questions we study in this paper are efforts to integrate N-body methods with FPGAs. Examples include [46] in which a direct $\mathcal{O}(N^2)$ N-body code (without FMM acceleration) is ported to an FPGA. In [52], the authors combine FPGAs with tree codes (another variant of fast N-body code). This is an FFT based N-body particle code, where near interactions for each tree node are shipped to an FPGA. The authors evaluate performance improvements, but they do not study the coupling placement and co-optimizations tradeoffs questions we consider here. In [36] the authors study an ASIC for the direct evaluation of pairwise interactions. The main focus of [36] is the description of the ASIC and characterization of its performance. It does not examine more complex algorithms like FMM and no other architectural tradeoffs are studied.

3 Background

In this section, we briefly describe the FMM application and LAP accelerator used as examples for our study.

3.1 FMM

The FMM was originally presented in [22] as an algorithm to accelerate computation of gravitational (or Coulombic) potentials due to a distribution of N-particles (also called an N-body problem),

$$\phi_i = \sum_{j=1}^{N} G(x_i, x_j)\, m_j, \tag{1}$$

where x_i and m_i are the positions and masses of the particles, $G(x_i, x_j) = \frac{1}{4\pi|x_i - x_j|}$ is the gravitational potential kernel and ϕ_i is the required potential at particle i. This is essentially a matrix-vector product with a dense $N \times N$ matrix whose entries are $G_{ij} = G(x_i, x_j)$ and it requires $\mathcal{O}(N^2)$ work to compute this product. The FMM works by geometrically partitioning the problem into smaller boxes (Figure 1-Left) using a tree data-structure (called octree in 3D). The interactions between adjacent boxes (called the U-list interactions) are computed directly and all other interactions are computed using clever hierarchical low-rank approximations. This reduces the overall cost to $\mathcal{O}(N)$.

Based on this idea several variants of FMM have been developed, including the volume FMM [21], [19], [13], [30], [37], [38], which computes potentials due to continuous mass (or electric charge) distributions through an integral transform. The potential computed in this way solves the Poisson's equation: $\Delta\phi = f$ where f is the mass density and ϕ is the required potential. This idea is also applicable to solving many other partial differential equations (PDEs). Instead of computing interactions between irregular distributions of particles, volume FMM can be viewed as having regularly spaced particles in each leaf-box of the tree, sampling the density function at those locations[1] (Figure 1-Center). In 3D, there are $(q+1)(q+2)(q+3)/6 \approx q^3/6$ particles per box, where the parameter q is the order of the method. Interactions between adjacent boxes, called the near-interaction or the U-list interactions, are computed directly using matrix-vector products while the interactions with the remaining boxes are computed using hierarchical low-rank approximations. In this paper, we only consider the U-list part of the computation which we describe in more detail below.

The N-body problems that we consider are translationally invariant and scale invariant i.e. if all the particles are shifted in space by the same amount or the distance between

1. in practice we use a Galerkin scheme using Chebyshev basis instead of a nodal basis; however, this detail is unimportant for the discussion in this paper.

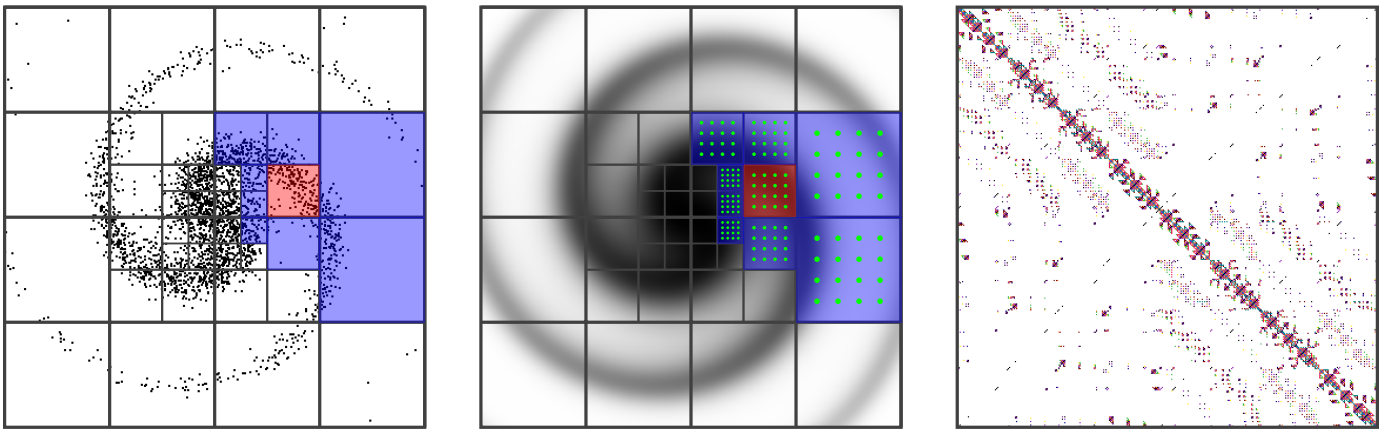


Fig. 1: *Left:* A gravitational N-body problem depicting a galaxy with several particles and the geometric partitioning of the problem in FMM. The particles in the red-box interact directly (using matrix-vector product) only with particles in the blue-boxes (in addition to the red-box itself). The remaining interactions are approximated using hierarchical low-rank approximations. *Center:* The continuous analogue of the particle N-body problem. The continuous mass density function is sampled at regularly spaced points (green) in each box. The interactions between particles in the red-box are computed using direct mat-vecs with particles in the red and blue boxes. This set of boxes is called the U-list of the red box and the interactions are called U-list interactions. *Right:* The $N \times N$ matrix for a volume FMM problem with order $q = 14$ and 358 boxes, each containing 680 particles for a total of $N = 2.4e5$ particles. Each pixel in the image represents a dense 680×680 block of the matrix corresponding to the interaction between two boxes. The colored pixels represent the U-list interactions and the colors denotes one of the 10 precomputed matrices from which it is derived.

the particles is scaled by a constant factor, then the potentials remain the same (up to a constant scaling factor). In the volume FMM, since the particles are regularly spaced in each box, the matrix block for the U-list interaction between two boxes depends only on the relative position of the two boxes. In our octree data structure, we ensure that adjacent leaf-nodes are within one level of each other. This restricts the number of unique U-list interaction matrix blocks to just 139 and allows us to precompute these matrix blocks instead of evaluating the matrix entries on the fly. Furthermore, these N-body problems are also rotationally invariant i.e. if the entire domain is rotated then the potentials still do not change. For the FMM algorithm this implies that each of the 139 U-list interaction matrix blocks (A^i for $i = 1, \cdots, 139$) can be derived from one of ten precomputed matrices by reordering and scaling its rows and columns: $A^i = M^i \hat{A}^{j_i} K^i$, where M^i and K^i are permutation and scaling operators (special type of sparse matrices) requiring $\mathcal{O}(q^3)$ storage each; and $\hat{A}^j$ (for $j = 1, \cdots, 10$) are precomputed dense matrices requiring $\mathcal{O}(q^6)$ storage each.

Figure 1-Right shows the U-list matrix blocks in a volume FMM problem with the colors denoting the precomputed matrix from which the matrix-block is derived. The repeating matrix blocks provide tremendous opportunity for maximize performance by exploiting data locality in computing these interactions. The PVFMM library [37], [38], which is the test bed for for all of our experiments, is highly-tuned for the x86 architecture, on which it has demonstrated up to 60% of the peak theoretical FLOP rate. It achieves this high performance using data locality in the computations.

This sparsity pattern and repeating matrix blocks (due to translational, scale and rotational invariance of the PDE) is also typical of other other PDE solvers relying on uniform meshes or trees based spatial discretization of the domain, such as finite element methods (FEM) and finite difference methods (FDM). Therefore, the results of our experiments also extend to these methods. Furthermore, the parameter q, which we vary in our experiments, determines the size of the matrix blocks in U-list computation and determines the arithmetic intensity (ratio of floating point operations to memory operations) of the computation. When $q \geq 8$, we are in a compute-bound regime and the workload resembles many practical FMM problems as well as high-order stencil codes. When $q < 8$, we are in a memory-bound regime and the workload resembles low-order block-structured stencil codes. Thus by varying q we can analyze both compute-bound and memory-bound applications.

3.2 Linear Algebra Processor (LAP)

We use the Linear Algebra Processor (LAP) in our studies as an example of a typical GEMM accelerator. With specialized microarchitecture optimizations, the LAP can provide massive parallel compute capabilities within tight power and area constraints. Previous work has demonstrated that a double-precision LAP with 15 internal cores can run GEMM with 600 GFLOPS at 1.4GHz with 90% utilization in an area of 120mm^2 in 45nm technology [41]. By comparison, modern GPUs reach a GEMM performance of 350 GFLOPS at 70% utilization in a total of 500mm^2 chip area [41]. All in all, the LAP achieves 25 GFLOPS/W GEMM efficieny, making it 10x more energy efficient than modern GPUs [41].

Figure 2 shows the overall microarchitecture of a single LAP core. It consists of a 2D array of 4×4 processing elements (PEs). Each PE has a multiply-accumulate (MAC) unit with a local accumulator and local storage separated into a bigger single-ported and a smaller dual-ported local memory. PEs are connected by low-overhead horizontal and

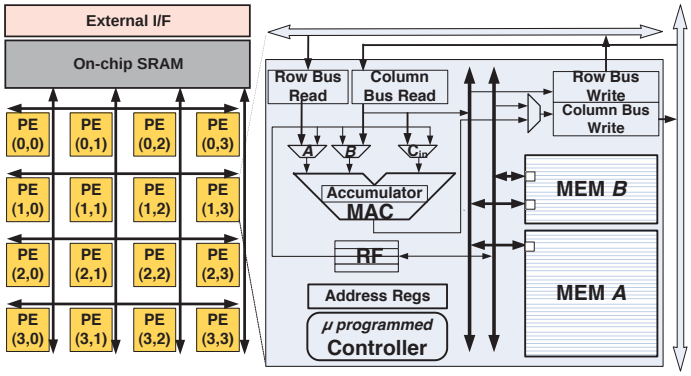


Fig. 2: Linear Algebra Processor (LAP) [41].

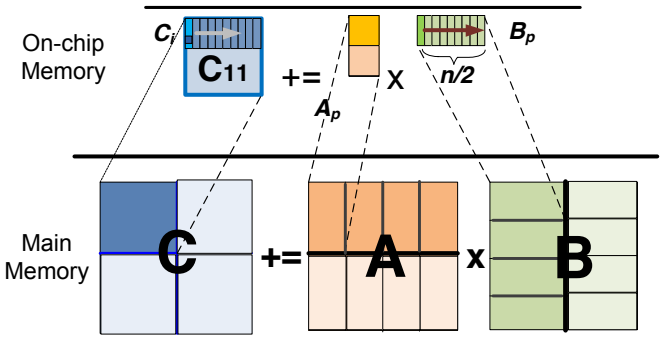


Fig. 3: Blocking mechanism of larger matrices [41].

vertical broadcast buses. MAC units perform the inner dot-product computations central to GEMM operation.

A LAP can integrate multiple cores together with a shared local scratchpad memory. When a GEMM kernel is offloaded to the LAP, its problem size can exceed LAP local memory capacity. To address this challenge, the LAP internally tiles/blocks larger problems into smaller chunks that fit into its local memory and that are processed successively. Figure 3 illustrates the LAP blocking scheme. The LAP fetches sub-blocks of matrices A, B and C into its local memory, denoted by $A_p, B_p, C_{i,j}$ respectively. The LAP cores then further block sub-matrices to distribute them across PEs. Multiple LAP cores can thereby operate on different sub-blocks concurrently. The LAP performs sub-block operations repeatedly until the GEMM is complete.

For details about the LAP architecture and its performance analyses see [41]. In our study, we employ a single-core LAP with 2MB of internal memory. We modified the application to offload the GEMM kernel to the LAP with the help of a device driver. The LAP then performs the GEMM, and interrupts the host CPU when it has finished.

4 Application Model

In this section, we use the FMM example to derive a generalized and representative HPC application model. We first describe FMM dataflow and computation patterns, and then introduce a corresponding application performance model.

4.1 Data Flow and Computation Patterns

The overall data flow and computation pattern is presented in Figure 4. It shows the three main parts in U-list computation, namely the input permutation, the main GEMM Kernel, and the output permutation. The preprocessing or the input permutation step applies the permutation and scaling operator K to the input array, labeled *"in"*, and assembles the result into the matrix B. This step mainly consists of light-weight computation along with data shuffling. The main GEMM computation kernel then computes the product of the precomputed matrix $\hat{A}$ with matrix B and produces matrix C as the result. The width of matrices B and C is equal to the number of times the matrix block $\hat{A}$ appears in the U-list matrix (shown in Figure 1-Right) and this can be hundreds or even thousands of columns wide. Since the GEMM requires a significant amount of compute resources, it is the main candidate for accelerator offloading. Once the GEMM is complete, the matrix C is sent to the host CPU which performs the post-processing computation to assemble the final result in the array, labeled *"out"*. This again involves a combination of light-weight computation with data reshuffling.

This workflow represents typical program patterns observed in HPC applications consisting of a mix of irregular data transformation and compute-intensive stages. Throughout this paper, we use the U-list as a representation of application patterns requiring significant data exchanges between irregular code executing on host CPUs and compute-intensive kernels offloaded to accelerators.

The main FMM parameter in our experiments is q, the polynomial order of the leaf nodes. For each q, we keep the total problem size fixed at $N = 2.4e5$ by adjusting the number of leaf nodes. The matrix B is in the GEMM (Figure 4), has height approximately $(q+2)^3/6$. The average width of B is approximately $2.35N/((q+2)^3/6)$ (the factor 2.35 is the number of times each $\hat{A}_k$ blocks repeats on average per block row in the FMM matrix in Figure 1-Right). Hence for problem size N=2.4e5 and q=10, the size of B is approximately 286-by-2K and for q=4 it is approximately 35-by-16K. The order q determines one dimension of the GEMM (the dimension of the square matrix A is approximately $(q+2)^3/6$, Figure 4). While the size of B (and C) remains constant, the smaller q is, the smaller A will be and the thinner the GEMM becomes.

4.2 Performance Model

We present a performance model for generalized applications that exhibit FMM-like dataflow/computation patterns. Application model parameters and their values for typical problem degrees are summarized in Table 1 and Table 2, respectively. As described earlier, the general application model consists of three sequential stages:

1) Input permutation: pre-processing to gather leaf node data with dimensions $N_{oct} \times N_q$ and assemble matrix B with dimensions $k \times n = N_q \times N_U$.
2) Computational kernel: perform GEMM operations to compute pieces $C_j = \hat{A}_j B_j = 1 \ldots 10$ each with problem size $m \times k \times n = N_q \times N_q \times N_U/10$.
3) Output permutation: post-process the combined matrix C with dimensions $m \times n = N_q \times N_U$ to scatter results back to leaf nodes.

As mentioned before, pre-/post- processing is executed on the CPU while the GEMM kernel is either executed on the CPU or offloaded the LAP.

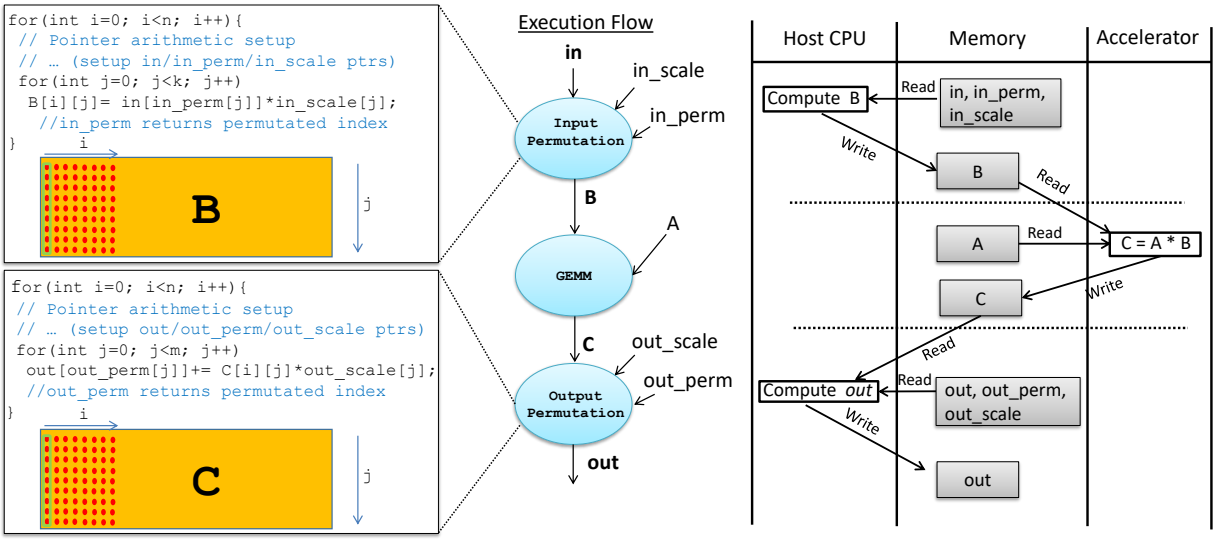


Fig. 4: Data flow graph (DFG) of U-list execution and data movement patterns.

The total execution time T is the sum of execution times T_1, T_2 and T_3 for pre-processing, GEMM and post-processing, respectively:

$$T = T_1 + T_2 + T_3 \tag{2}$$

The total data exchanged between pre-/post-processing and the GEMM invocation can reach more than 3x the total capacity of even the large 8MB LLC. This requires a significant amount of shared data written to the LLC by a producer to be evicted and spilled to DRAM, from where the consumer has to read it again. As such, shared data will be spillled and predominantly exchanged through DRAM. We can thus derive an upper bound on performance assuming that pre- and post-processing are memory-bound and not cache-contained, where their cost is dominated by the time required to read and write matrices from/to main memory. In case of pre-processing, this includes write-allocating matrix B into the cache, reading leaf node data, and writing B back to memory:

$$T_1 = (2N_U N_q + N_{oct} N_q)/w_C, \tag{3}$$

where w_C is the effective memory bandwidth for a combination of regular and irregular transfers made by the CPU.

Similarly, in case of post-processing, the cost is determined by the time to read leaf node data, read matrix C, and write leaf node data back to memory:

$$T_3 = (2N_{oct} N_q + N_U N_q)/w_C. \tag{4}$$

Finally, assuming that memory accesses and computations are perfectly overlapped, we can derive the cost of executing the ten GEMM operations as the maximum of the compute time for one multiply and one accumulate operation per element and the times to read matrices A and B and write-allocate and write C from/to memory:

$$\begin{aligned} T_2 &= 10 \max(2N_U N_q^2/10 f_C, N_q(N_q + 3N_U/10)/w_C) \\ &= \max(2N_U N_q^2/f_C,\ N_q(10N_q + 3N_U)/w_C), \end{aligned} \tag{5}$$

where f_C is the effective number of floating-point operations per second (FLOPS) when executing the GEMM.

TABLE 1: Application model parameters.

Symbol	Description
q	Chebyshev degree
N_q	$(q+1)(q+2)(q+3)/6$
N_{oct}	Number of leaf nodes
N_U	Total number of U-list interactions
f_C/ f_L	CPU/LAP FLOP-rate
w_C/ w_L	CPU/LAP memory bandwidth

TABLE 2: Application parameters as a function of q.

				GEMM sizes		
q	N_q	N_{oct}	N_U	m	k	n
4	35	6917	164913	35	35	16491
6	84	2885	68769	84	84	6877
8	165	1450	34172	165	165	3417
10	286	869	20129	286	286	2013
14	680	358	8336	680	680	834
18	1330	183	4309	1330	1330	431

We validated our performance model by comparing predicted performance against real performance measured on an Intel Core i7-920 CPU running at 2GHz with a peak flop-rate of 8GFLOPS and single-core maximum measured bandwidth of 6.4GB/s (theoretical peak is 12GB/s). We run experiments ten times and measure average performance. The observed variation across runs is less than $\pm 0.5\%$. Table 3 shows the comparison of measured execution times versus performance predicted by the model. Results generally confirm that the model captures runtime trends accurately. Output permutation runtimes show variations with q that are not tracked by our model. This is due to memory transfers becoming more irregular and hence more bandwidth-inefficient depending on q. We measured a bandwidth of 0.13GB/s for randomized transfers on the Intel machine and account for this by applying a reduced average CPU bandwidth of 0.32GB/s in our output permutation model.

TABLE 3: CPU performance model validation.

	Model [MCycles]			Real [MCycles]		
q	T_1	T_2	T_3	T_1	T_2	T_3
4	29	101	38	27	158	26
6	29	242	31	26	303	33
8	28	465	33	26	533	34
10	29	823	35	26	916	35
14	28	1928	34	25	2064	34

5 Architecture Model

Based on their granularity of coupling, hardware accelerators can be fundamentally classified as *tightly coupled* or *loosely coupled* [8]. Tightly coupled accelerators are attached to the main processor pipeline as one or more specialized functional units (FUs) located inside or very close to the main processor core. They typically realize fine-grain, short-latency computations, such as floating-point, vector or SIMD operations. As such, they generally share key resources with the processor core (e.g., register file, MMU and L1 data cache) while in some cases possessing their own additional internal storage. Similar to co-processors, tightly coupled accelerators require special instruction set architecture (ISA) extensions to manage their operation.

By contrast, loosely coupled accelerators are located outside of the main processor core and connected via cache, memory or I/O buses, or over on- or off-chip interconnect. Loosely coupled accelerators are often also called *coarse-grained*, due to the fact that they typically operate on coarser invocations of complete tasks, kernels or even an entire application. They typically operate asynchronously, where triggering of their operation and synchronization of completion and results is achieved through memory-mapped I/O, polling or interrupt mechanisms. Proper communication and synchronization with such accelerators is accomplished through device drivers for handling of interrupts, cache flushes and/or memory copies. GPUs and network/cryptography processors [35], [4] are examples that belong to this class.

In this work, we focus on integration tradeoffs of loosely coupled custom ASIC accelerators. We use the LAP [41] as an example for coarse-grain custom hardware acceleration of GEMM operations in the FMM U-list. We study different LAP coupling scenarios and their implications and system tradeoffs. Specifically, we investigate two different cases (Figure 5): (1) the accelerator is externally coupled as an off-chip device sharing DRAM with the host CPU, and (2) the accelerator is located on the same chip as the host CPU sharing the last level cache (LLC). In this study, we assume a modern, high-end x86 CPU with a typical aggressive wide-issue and out-of-order processor micro-architecture.

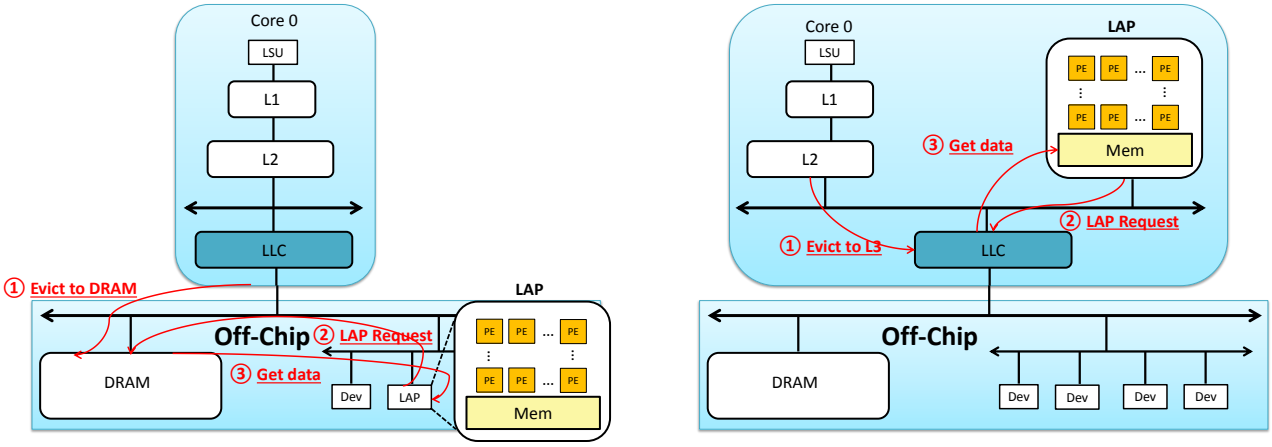
5.1 External Off-chip Coupling at Shared DRAM Level

In this scenario (Figure 5a), the LAP is connected to the external, off-chip system bus. The LAP accesses all its necessary input and output data in the DRAM through the system bus and a DMA controller. To avoid coherency protocol overhead, we assume that LAP and CPU are non-coherent. Instead, as shown in Figure 5a, before the LAP attempts to read inputs from the memory, if there exists any dirty data in the host CPU caches, the software has to explicitly manage coherency and tell the CPU to evict relevant data from its caches to DRAM. For example, this can be done via the *"clflush"* instruction in the x86 ISA. Once data is synchronized and the LAP has been triggered, it will operate directly on data in DRAM. The LAP will issue memory requests to DRAM, subject to DRAM latency. The LAP will prefetch data and overlap computation and communication through its internal scratchpad memory. At the end of one invocation, the LAP will have stored all output data back to DRAM, from where it can be accessed by the CPU. Before accessing the LAP results, the CPU driver will ensure that all previously cached copies are invalidated.

In a PIM setting, the accelerator would be coupled and integrated directly in or near DRAM. The main benefit is increased bandwidth for DRAM accesses by the accelerator. However, GEMM on the LAP is compute-bound and not bandwidth-limited, i.e., we would not expect to see any performance benefits from such in-memory integration. As such, we exclude this as a separate case in our studies. From an energy perspective, our simulation models currently do not explicitly account for off-chip interconnect energy. This underestimates energy costs for DRAM accesses. At the same time, in a PIM case, off-chip interconnect overhead would only be incurred for DRAM accesses by the CPU, which would in turn again be closer to our reported results. In literature, dynamic energy costs of 20pJ/bit are reported for off-chip DRAM interfaces in current 40nm technology [28] on top off around 60pJ/bit for actual DRAM accesses including all dynamic read/write and activation energy as modeled in our setup. In addition, both interconnect and DRAM incur similar proportions of static versus dynamic power [25]. Our reported DRAM energy results can thus be scaled accordingly to account for additional interconnect costs in either PIM or non-PIM scenarios.

5.2 Internal On-chip Coupling at Shared Cache Level

To avoid communication via external DRAM accesses, we couple the LAP at the LLC by connecting it to the internal bus network between L2 and LLC—in the same way as other cores are interconnected in multicore systems. As shown in Figure 5b, the LAP shares the LLC with other cores. It will access all data through the LLC. Since the LAP itself explicitly manages all memory and internal scratchpad accesses, we want to avoid the overhead of it having to participate in the L2 coherency protocol. Instead, the LAP expects necessary data to be in LLC or DRAM when it starts computing. However, since data might be cached in the CPU's L1 or L2, the CPU now has to explicitly evict data to the LLC. In our setup, caches are non-inclusive and write-back. The *clflush* instruction by default evicts the targeted block all the way to DRAM. If the block exists in the LLC, it will be invalidated and further evicted. In our scenario, we wish to manage the eviction and keep or store the block in the LLC by not invalidating and potentially updating an already existing, or by allocating a new LLC cache line that potentially replaces and evicts others. There are two plausible options to support this: (1) adding a special instruction in the ISA that handles eviction only to the LLC, or (2) setting a configurable register in the LLC controller to identify if a block should be invalidated or kept/stored



(a) Off-chip coupling at shared DRAM.

(b) On-chip coupling at shared LLC.

Fig. 5: Accelerator coupling at different levels of the memory hierarchy.

when software eviction happens. We opted for the latter implementation due to its simplicity and transparent change from the software perspective.

Figure 5b shows the data movements for this case. The CPU evicts necessary data to the LLC. Afterwards, the LAP accesses the LLC to get its data to operate on. If the LAP experiences an LLC miss, the request is forwarded and serviced by the main memory controller. Similarly, the LAP writes results back to the LLC, which, in case of capacity misses, can lead to previous results being evicted to DRAM.

A closer integration with potentially lower distance and latency would be to couple the accelerator to the CPU at the level of L1 or L2 caches and buses. While we considered this option, it is less appropriate for loosely coupled coarse-grain accelerators. Such accelerators usually afford a greater area budget than tightly coupled ones. This allows them to operate with complex data paths and internal memory for acceleration of complete tasks or kernels. Coupling a greater area budget close to the main CPU would pose challenges in achieving timing closure and meeting tight clock frequency constraints set for latency-sensitive L1 and L2 caches. Moreover, buses and interconnect of private L1 and L2 caches are usually not designed to support multiple connected cores. Furthermore, when task-level accelerators are allowed to access and share a private L1 or L2, the capacity pressure of the relatively small caches increases. Hence, potential performance degradation resulting from cache pollution and cache trashing can negate acceleration benefits, especially when executing on CPU cores and accelerators concurrently. A number of recent works have proposed acceleration engines placed in between L1 and L2 [2]. However, for reasons outlined above, such approaches do not target task/kernel level invocations, but are restricted to acceleration at finer instruction-level granularity.

5.3 Performance Model

In the following, we extend our performance model from Section 4.2 to incorporate different on-/off-chip accelerator coupling scenarios. We assume that once the CPU is done with pre-processing, the GEMM is offloaded to the accelerator without any further modifications. However, as mentioned in Section 4.2, since the total data exchanged by the CPU and the accelerator during one GEMM invocation can reach more than 3x the total capacity of the 8MB LLC, shared data will be spillled and predominantly exchanged through DRAM in both on- and off-chip coupling scenarios. In other words, a large fraction of data is exchanged through the DRAM in either case, i.e., even when the LAP is coupled to the LLC.

We can thus derive an upper bound on performance assuming that pre- and post-processing times T_1 and T_2 remain memory-bound and are not impacted by acceleration or different accelerator coupling scenarios. In contrast to the CPU performance model from Section 4.2, however, we can express the processing time of the GEMM kernel on the LAP as follows. As mentioned before, the LAP performs internal blocking to compute $C = AB$ in r sub-blocks with dimensions $m_c \times n_c$:

$$r = \lceil n/n_c \rceil * \lceil m/m_c \rceil \tag{6}$$

To compute each sub-block, the LAP needs to read the sub-block of C with size $m_c \times n_c$ as well as $m_c \times k$ and $k \times n_c$ sized panels of A and B into its local memory, perform the GEMM and then write the sub-block of C back. Again assuming that memory accesses and computations are perfectly overlapped, we can derive the cost of computing all r sub-blocks for each of the ten GEMMs on the LAP as the maximum of compute time and the time to read/write data from/to memory:

$$\begin{aligned} T_2 &= 10 \max(2mkn/f_L,\ r(2m_cn_c + m_ck + kn_c)/w_L) \\ &= \max(2N_U N_q^2/f_L,\ 10r(2m_cn_c + N_q(m_c + n_c))/w_L), \end{aligned} \tag{7}$$

where block sizes for a LAP with 2MB of local memory are determined to fit all data as $m_c = \min(\max(m, 64), 256)$ and $n_c = \min(\max(n, 64), 256)$.

6 HW/SW Co-Optimization

In this section, we discuss HW/SW co-optimization opportunities to exploit coupling options and optimally take

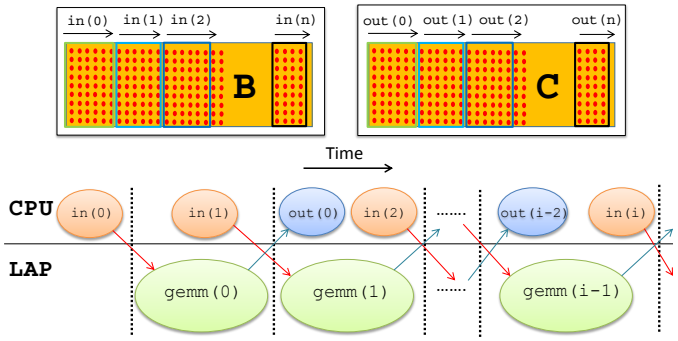


Fig. 6: Software blocking, pipelining and overlapping.

advantage of the underlying system architecture.

6.1 Cache-Aware Blocking

In order to minimize data movement and unnecessary round-trip transfers to DRAM with LLC coupling, all data should ideally be exchanged between CPU and the accelerator exclusively through the LLC. As Figure 4 shows, in the U-list, CPU and LAP communicate by exchanging matrices B and C during GEMM calls. Since the LAP operates on both matrices simultaneously in a streaming fashion, ideally both matrix B and matrix C should remain resident in the LLC for the duration of such calls. To achieve this, we modify the U-list such that the program operates on smaller, blocked versions of matrix B and C whose total size is equal or smaller than the LLC capacity.

6.2 Prefetching, Pipelining and Overlapping

When GEMM operations are executing on the LAP, the CPU is generally idling until the LAP signals completion and the output permutation can resume by processing the C result. However, we can note that auxiliary *M.scale* and *M.perm* metadata needed by output permutations is independent of GEMM execution and can thus be prefetched ahead. We modify the code flow to accommodate this optimization, such that the CPU prefetches the required metadata for output permutation while the LAP is performing the GEMM.

Moreover, with blocking of matrices into tasks with smaller granularity, we can apply software pipelining to exploit additional overlapping and parallelism between accelerator and CPU. As shown in Figure 6, with input permutation, GEMM, and output permutation tasks denoted as *in(i)*, *gemm(i)*, and *out(i)*, respectively, where *i* represents the iteration phase of the blocking, while the LAP is operating on one iteration of the GEMM, the CPU simultaneously performs output and input permutations for the portions of B and C from the previous and next iterations, respectively.

Initially when *i=0*, the CPU starts executing the first block of the input computation, *in(0)*. The first block of the input computation will produce a number of column vectors of matrix B, encapsulated by the green rectangle on the top part of figure 6. Afterwards, when the input permutation enter stage *in(1)*, the LAP executes *gemm(0)* which takes the input produced by the CPU at *in(0)*. Then on the next stage, the result of the GEMM computation will be ready to be used by the *out(0)*. Moreover, on this stage, the LAP is simultaneously ready to execute the next iteration *gemm(1)*.

6.3 Performance Model

We discuss how our performance model can be further extended to account for blocking and pipelining optimizations. Execution is blocked into b smaller sub-problems of size $m \times k \times n = N_q \times N_q \times N_U/b$, such that blocks of matrices B and C are exchanged between CPU and LAP through the LLC without the need to go to DRAM:

$$T = b\,(T_1 + T_2 + T_3), \tag{8}$$

where pre-processing incurs costs for read-allocating relevant leaf node data while flushing dirty lines being replaced back to DRAM, the GEMM is predominantly cache-contained, and post-processing incurs costs for read-alllocating leaf node data into the cache. The size of leaf node data accessed per block is thereby upper-bounded by the block dimension $n = N_U/b$:

$$T_1 = 2\min(N_{oct}, N_U/b)N_q/w_C \tag{9}$$
$$T_2 = 2N_U N_q^2/bf_L \tag{10}$$
$$T_3 = \min(N_{oct}, N_U/b)N_q/w_C \tag{11}$$

These b iterations are then further pipelined, where input and output permutation on the CPU are overlapped with the GEMM on the LAP. Assuming perfect overlapping between all memory transfers and computations, and ignoring filling and flushing of the pipeline, the pipelined execution time becomes:

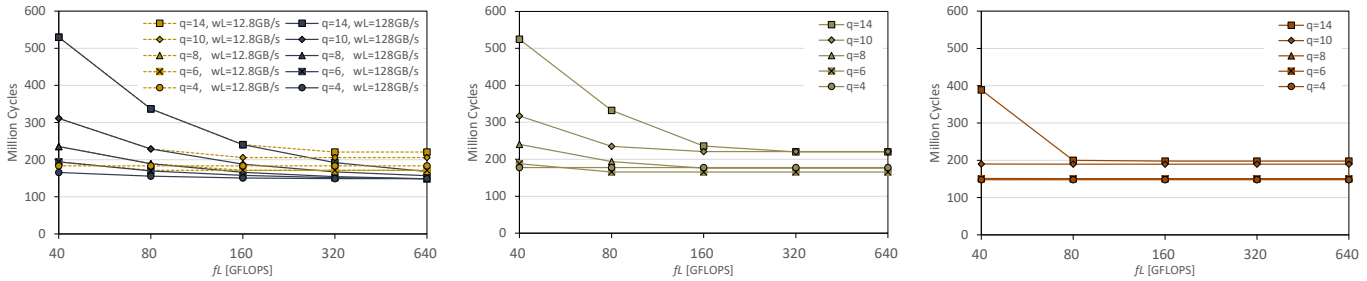
$$T = b\ \max(2N_U N_q^2/bf_L, 3\min(N_{oct}, N_U/b)N_q/w_C) \tag{12}$$

7 Model-Based Exploration

In this section, we use analytical performance models presented in previous sections to discuss algorithm/architecture exploration on various scenarios. We demonstrate different design points with respect to varying parameters such as *Chebyshev degree*, FLOPS, bandwidth and different HW/SW co-optimizations and coupling scenarios.

Figure 7a plots projected FMM runtime in total system cycles (CPU cycles at 2GHz) for varying q across different f_L with off-chip coupling following Equations (2)-(4) and Equation (7). We explore both a traditional DRAM coupling of the LAP (using peak DRAM bandwidth of $w_L = 12.8\,\text{GB/s}$) as well as a near-memory scenario with 10x more LAP bandwidth ($w_L = 128\,\text{GB/s}$). In general, runtime improves with f_L for both options until the GEMM on the LAP becomes memory-bound. For large q, runtime is dominated by the GEMM kernels and significant speedups can be achieved by acceleration. For $q < 8$, the GEMMs become smaller and are memory-dominated already at lower LAP FLOPS. A near-memory integration can provide additional performance, but gains are limited by a strict performance floor set by pre- and post-processing runtimes on the CPU. In memory-bound cases, near-memory integration can improve runtime by up to 20% for $q = 14$.

Figure 7b and Figure 7c show FMM runtime for on-chip coupling with blocking and pipeling optimizations (Equations (8) and (12)) , respectively. Comparing Figure 7a and Figure 7b, we can observe that blocking optimizations alone rarely improve baseline performance. Since matrices need to be brought into the cache at least once, pre- and



(a) Off-chip DRAM coupling. (b) On-chip LLC coupling with blocking. (c) On-chip LLC coupling with pipelining.

Fig. 7: Design space explorations with different optimizations and LAP coupling scenarios.

post-processing benefit from blocking optimizations only in a limited manner, while the GEMM is largely compute-dominated. By contrast, pipelining optimizations (Figure 7c) can further improve performance. With a properly balanced pipeline, the GEMMs are hidden and a constant execution time as determined by the sum of pre- and post-processing times is achieved irrespective of q or f_L. For $q = 14$ and low f_L, a pipeline imbalance leads to runtime being dominated by GEMMs again.

8 Simulation-Based Evaluation

In this section, we present our evaluation methodology and comprehensive results across different acceleration options. As discussed before, we adopt FMM U-list as a representative application exhibiting heterogeneous data exchanges between host CPU and an accelerator.

We use a cycle-accurate MARSSx86 [5] full-system simulator for architecture evaluation. The simulator itself has been calibrated to model a highly representative x86 CPU baseline targeting HPC applications. We specifically configured and calibrated the simulator to match an Intel Core i7-920 CPU. As an example of a hardware accelerator, the simulator from [5] integrates a cycle-accurate model of the LAP. We feed statistics obtained from MARSSx86 into McPAT [33] to model the total CPU energy. Finally, we attach DRAMSim2 [43] to model an accurate DRAM delay and energy. We measured a single-core maximum bandwidth of 3GB/s in the simulator and use that for performance models. Table 4 shows the detailed system configuration.

We run FMM using the simulator across different qs. We fast-forward execution in native emulation and only simulate the U-list using the cycle-accurate system model. Since execution statistics from the simulator are deterministic, we only use a single run for every q scenario.

TABLE 4: System configuration.

Parameter	Core Model
Clock frequency	2GHz
Fetch Width	4
Dispatch Width	4
Issue Width	5
Commit Width	4
Writeback Width	4
Cache Block Size	64B
L1-I Cache	32kB, 8-way
L1-D Cache	32kB, 8-way
L2 Cache	256kB, 8-way
LLC	8MB, 16-way
LAP	
FLOPS rate	40GFLOPS
Local Memory	2MB
DRAM	
DDR Type	DDR3-1600
Specification	Micron MT41J256M4

8.1 DRAM Accesses

We first evaluate impact of integration options on DRAM overhead, which is the main optimization goal affected by different accelerator couplings. Figure 8 shows the total number of DRAM accesses in the U-list under different HW/SW coupling options across varying q. We compare the original U-list with basic acceleration of GEMM calls (*Orig*) to a U-list with prefetching (*Orig + Pref*), with cache-aware blocking (*Blocking*), with blocking and prefetching (*Blocking + Pref*), and with blocking and software pipelining (*Pipelining*) for both DRAM and LLC coupled LAP accelerators. As expected, when the accelerator is coupled as an external device requiring data to be always exchanged via DRAM, none of the software optimizations significantly affect DRAM accesses. DRAM accesses are slightly increased with blocking due to reduced cache and scratchpad locality with smaller matrices.

By contrast, when the accelerator is coupled at the LLC, a cache-aware blocking approach reduces the number of DRAM accesses by 50%. The blocking case is optimized to ensure that all data exchanged between CPU and accelerator fits into the LLC, avoiding DRAM round trips for CPU-LAP communication. However, blocking benefits decrease with higher q. In the $q = 14$ case, due to algorithmic limitation, the smallest chunk that matrices B and C can be partitioned into already exceeds LLC capacity. Hence, significant capacity misses are unavoidable and result in additional DRAM accesses.

Prefetching does not generally affect DRAM access counts, but in the blocked case (*Blocking + Pref*), a minor decrease is observed. This is due to the Least-Recently Used (LRU) cache replacement policy. With prefetching, data already in the LLC is marked as used, which reduces the likelihood of it being evicted before being needed next as compared to other, less recently used lines. However,

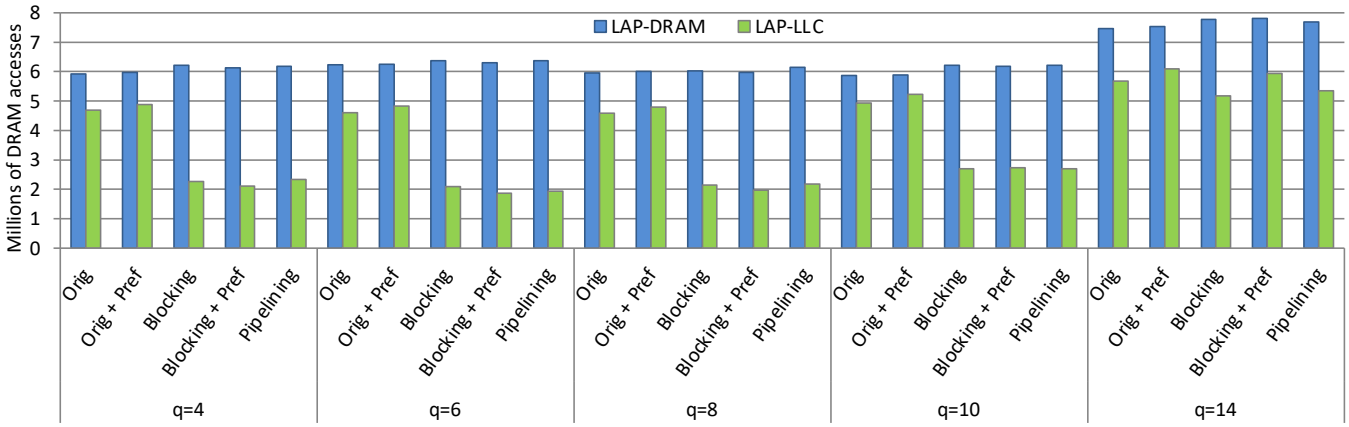


Fig. 8: DRAM accesses with different integration options across varying q.

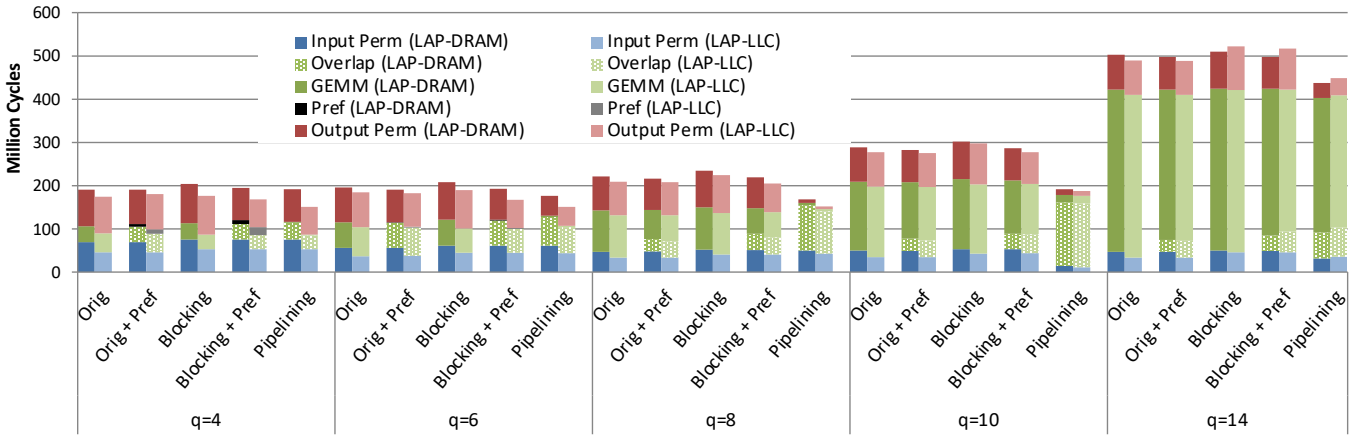


Fig. 9: Execution cycles with different integration options across varying q.

adding prefetching to a non-blocking approach instead increases the number of DRAM accesses (*Orig + Prefetch*). Due to the large number of capacity conflicts in the non-blocking case, prefetched data is likely be evicted again from the cache before it can even be used.

8.2 Execution Time

Figure 9 shows total system cycles for U-list execution under different optimization and coupling options across varying q. Total execution time is broken down into cycles spent on different parts, including *Overlap* periods in which the GEMM on the LAP runs in parallel with either prefetching or input and output permutations on the CPU. Note that *Pref* refers to extra cycles spent on prefetching when it can not be completely hidden behind a shorter GEMM.

Among HW/SW optimizations, due to additional overheads, performance is slightly decreased with blocking alone. Prefetching can in turn be applied to compensate such overheads. The best improvement is observed from pipelining and overlapping (*Pipelining*). At lower q, execution times are dominated by permutations and there are only small gains from overlapping them on the CPU with the GEMM on the LAP. As q grows, first input and then output permutations are hidden behind the increasingly dominating GEMM (except for non-overlapped invocations needed to fill and flush the software pipeline). At $q = 10$, both permutations are maximally overlapped and balanced with the GEMM. This yields a 37% better performance compared to the non-optimized solution (*Orig*). However, these speedups almost exclusively stem from exploiting available parallelism between accelerator and CPU, i.e. they are not a function of data movement optimizations and, ultimately, do not vary significantly across different accelerator couplings. Although the number of DRAM accesses in Figure 9 strongly depends on accelerator coupling and is significantly reduced with LLC blocking, the impact on performance is generally relatively minor, with a maximal gain of up to 20% comparing LLC to DRAM coupling. As discussed in Section 7, the GEMM is normally compute-bound and input/output permutation do not benefit significantly from blocking.

Closer observation shows that input permutations consume more execution cycles in *LAP-DRAM* than *LAP-LLC* due to flushing overhead to off-chip DRAM. In case of the GEMM, as it becomes smaller and thinner with smaller q, the computation to memory ratio and data locality in the LAP decrease, making it more memory bound. With ample parallelism, the LAP executing is not latency sensitive, but coupling it to the LLC with higher bandwidth benefits performance resulting in 10% faster GEMM execution in LLC vs. DRAM coupling at $q = 4$ (higher DRAM bandwidth, e.g. through in-memory LAP placement could potentially alleviate such differences). However, at bigger q, GEMM executions differ by less than 2%. Finally, output

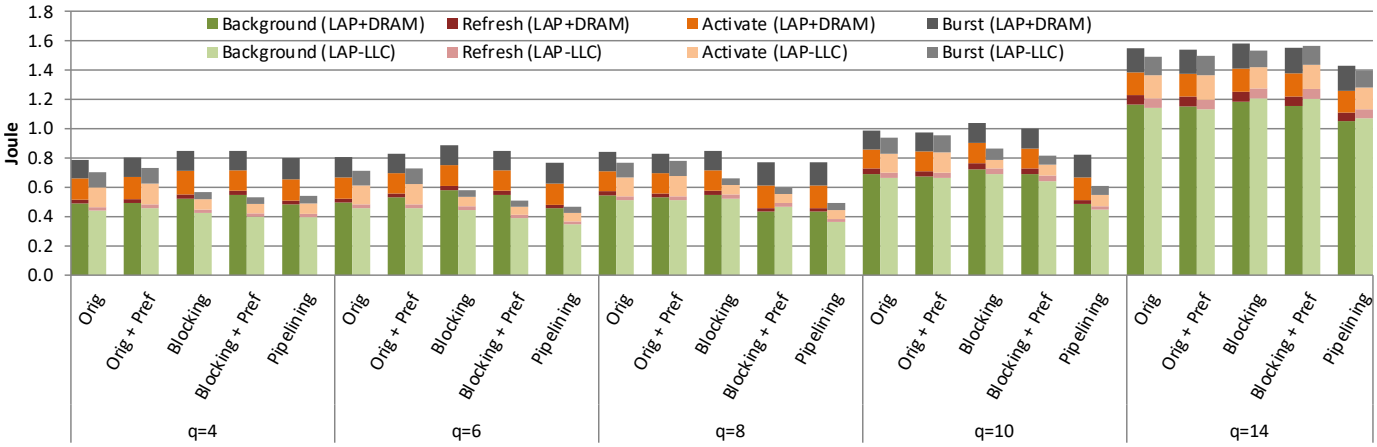


Fig. 10: DRAM energy with different integration options across varying q.

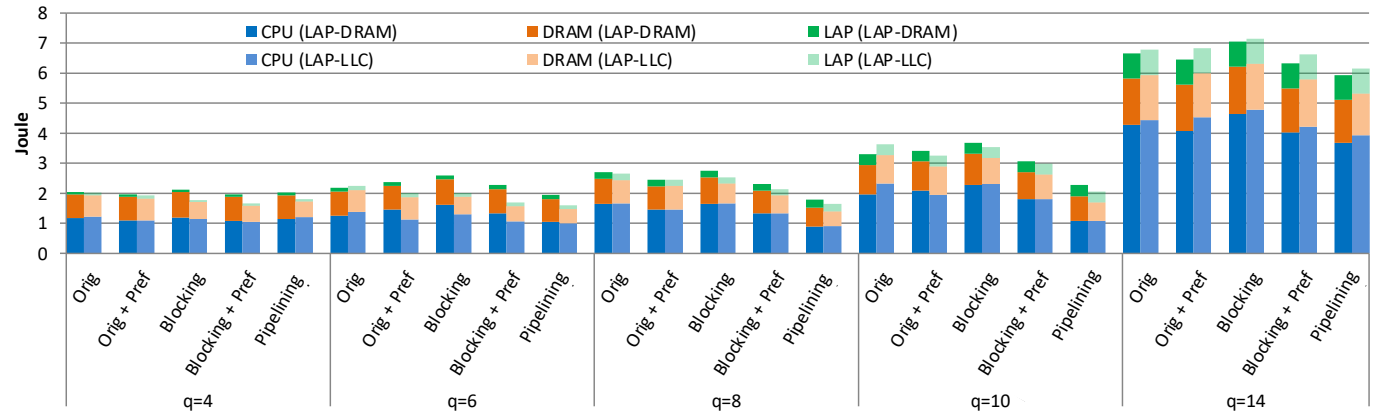


Fig. 11: Total energy with different integration options across varying q.

permutations are largely unaffected by coupling. However, due to increased overhead and cache pressure, performance actually decreases with LLC coupling and blocking at larger q. Coupled with previously discussed blocking limitations, this makes LLC worse than DRAM coupling at $q = 14$.

8.3 Energy

We further evaluate potential energy benefits of different architecture and algorithm options. We first focus on DRAM energy savings as the primary aspect that is affected by data movement and accelerator placement/coupling optimizations. Figure 10 shows the DRAM energy breakdown across different coupling options and varying q. We break energy consumption down into static background energy due to leakage, energy spent on periodic refresh, and dynamic energy consumption dependent on activation and read/write activity. As can be observed, static background and periodic refresh energy is the most contributing factor. Such idle baseline power contributes from 60% up to 80% of total energy consumption. As such, while dynamic energy reduces by 50%-60% proportionally to the reduced number of DRAM accesses shown in Figure 8 with blocking and LLC coupling, total energy improvements are limited due to small share of dynamic energy.

At the full system level (Figure 11), this translate into even smaller savings. The graph shows the total system energy broken down by contributions from individual system components. As can be seen, the CPU consumes the largest portion of energy spent, followed by the DRAM and the LAP with a comparably tiny fraction. Due to additional base overheads and relatively constant energy contributions by the CPU and the LAP, savings in dynamic DRAM energy across different integration options and data movement optimizations have limited effect on total energy. Variations in energy consumption are predominantly due to differences in static idle and leakage power accumulated over different total execution times. This is the case both at the DRAM and full system levels, where the CPU is also the dominant contributor of static leakage power (while its dynamic energy varies little given the same workload). In other words, energy savings are primarily driven by improvements in performance.

Through a combination of dynamic DRAM energy and execution time, i.e. static idle power savings, when coupling the LAP at LLC and applying aggressive software pipelining and overlapping optimizations, reductions in DRAM data movement between the LAP and CPU reduce the total system energy by up to 17% compared to coupling at the DRAM level. All in all, primarily driven by execution time savings, pipelining and overlapping yields up to 43% lower energy consumption compared to the original solution, but is, again, largely independent of the architecture coupling.

TABLE 5: Performance model validation.

	LAP-DRAM Orig [MCycles]		LAP-LLC Block [MCycles]		LAP-LLC Pipeline [MCycles]	
q	$Model$	Sim	$Model$	Sim	$Model$	Sim
4	183	190	177	176	148	150
6	193	195	188	189	150	150
8	235	220	240	223	151	152
10	311	288	317	297	190	188
14	529	502	524	522	388	448

8.4 Performance Model Validation

Table 5 shows a validation of our performance model against simulation results. The performance model tracks simulation results accurately across different q and optimization scenarios with less than 3% average error per model. Our pipelined model overestimates performance for $q = 14$. As mentioned before, the smallest chunk that matrices can be partitioned into in practice already exceeds LLC capacity for $q = 14$. This leads to performance losses when all memory transfers are overlapped in pipelined execution, which are not captured in our idealized model.

9 Summary and Conclusions

In this paper, we study system-level performance and energy saving benefits of hardware acceleration under different placement scenarios and software optimizations for a state-of-the-art Fast Multipole Method (FMM). We investigated accelerator coupling options at the architecture level and proposed several algorithm co-optimizations to maximize acceleration benefits. We showed that, in the context of a complex real-world application, simply placing an accelerator closer to the chip does not always lead to significant performance/energy wins. We demonstrated that an on-chip integration alone without supporting software optimizations can only provide limited gains. Only when more aggressive software/hardware co-optimizations are applied, additional gains can be unlocked. Moreover, when similar optimizations are employed in off-chip integration, on-chip integration presents up to 20% better performance with 17% less total energy consumption. In general, on-chip integration offers more performance opportunities when careful and aggressive co-optimizations are applied. Unlatching this, however, requires closer involvement and collaboration between both architects and programmers.

Such a study of ASIC acceleration and integration tradeoffs in the context of real, complex and well-optimized HPC applications has not been presented before. While absolute performance and energy results are specific to our targeted FMM application, U-list phase, x86 architecture and GEMM accelerator, we focus on analysis of relative tradeoffs, where these can be considered representative of typical HPC application and architecture patterns. In future work, we plan to extend such analyses to other application, architecture and acceleration options.

Acknowledgements

The material is based upon work supported by NSF awards CCF-1817048, CCF-1725743 and CCF-1337393, DOE awards DE-SC0019393 and DE-NA0003969, AFOSR award FA9550-17-1-0190, and computational resources from the Texas Advanced Computing Center (TACC).

References

[1] AMD Fusion. http://sites.amd.com/us/fusion/apu/Pages/fusion.aspx.
[2] S. Aga et al. Compute Caches. In *HPCA*, 2017.
[3] J. Ahn et al. A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing. In *ISCA*, 2015.
[4] R. Anderson et al. Cryptographic Processors: A Survey. *Proceedings of the IEEE*, 94(2):357–369, January 2006.
[5] M. Asri et al. Simulator calibration for accelerator-rich architecture studies. In *SAMOS*, July 2016.
[6] J. Board and K. Schulten. The Fast Multipole Algorithm. *Computing in Science and Engineering*, 2(1):76–79, February 2000.
[7] C. Burstedde, L. C. Wilcox, and O. Ghattas. p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, 2011.
[8] C. Cascaval et al. A taxonomy of accelerator architectures and their programming models. *IBM Journal of Research and Development*, 54(5):5:1–5:10, September 2010.
[9] Y. Chai, W. Shen, W. Xu, and Y. Zheng. Computing acceleration of fmm algorithm on the basis of fpga and gpu. *Advanced Materials Research*, 291-294:3272–3277, 07 2011.
[10] A. Chandramowlishwaran, K. Madduri, and R. Vuduc. Diagnosis, Tuning, and Redesign for Multicore Performance: A Case Study of the Fast Multipole Method. In *SC*, 2010.
[11] T. Chen et al. DianNao: a small-footprint high-throughput accelerator for ubiquitous machine learning. In *ASPLOS*, 2014.
[12] Y.-H. Chen et al. Eyeriss: An Energy-efficient reconfigurable accelerator for deep convolutional neural networks. In *ISSCC*, 2016.
[13] H. Cheng, J. Huang, and T. J. Leiterman. An adaptive fast solver for the modified helmholtz equation in two dimensions. *Journal of Computational Physics*, 211(2):616–637, Jan. 2006.
[14] J. Choi and R. W. Vuduc. Analyzing the energy efficiency of the fast multipole method using a DVFS-aware energy model. In *IPDPSW*, 2016.
[15] E. G. Cota et al. Exploiting Private Local Memories to Reduce the Opportunity Cost of Accelerator Integration. In *ICS*, 2016.
[16] V. Dang, Q. Nguyen, and O. Kilic. Fast multipole method for large-scale electromagnetic scattering problems on gpu cluster and fpga-accelerated platforms. *Applied Computational Electromagnetics Society Journal*, 28, 12 2013.
[17] J. Dongarra and F. Sullivan. The Top 10 Algorithms. *Computing in Science and Engineering*, 2(1):22–79, February 2000.
[18] H. Esmaeilzadeh et al. Dark Silicon and The End of Multicore Scaling. In *ISCA*, pages 365–376, June 2011.
[19] F. Ethridge and L. Greengard. A new fast-multipole accelerated poisson solver in two dimensions. *SIAM Journal on Scientific Computing*, 23(3):741–760, Jan. 2001.
[20] A. Gray and A. Moore. N-body problems in statistical learning. *Advances in neural information processing systems*, pages 521–527, 2001.
[21] L. Greengard and J.-Y. Lee. A direct adaptive poisson solver of arbitrary order accuracy. *Journal of Computational Physics*, 125(2):415–424, May 1996.
[22] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73(2):325–348, Dec. 1987.
[23] T. J. Ham et al. Graphicionado: A High-Performance and Energy-Efficient Accelerator for Graph Analytics. In *MICRO*, 2016.
[24] T. Huang, Y. Zhu, Y. Ha, X. Wang, and M. Qiu. A hardware pipeline with high energy and resource efficiency for fmm acceleration. *ACM Trans. Embed. Comput. Syst.*, 17(2), Jan. 2018.
[25] X. Jian, P. K. Hanumolu, and R. Kumar. Understanding and optimizing power consumption in memory networks. In *HPCA*, Feb 2017.
[26] X. Jiang et al. Architecture support for improving bulk memory copying and initialization performance. In *PACT*, 2009.
[27] N. P. Jouppi et al. In-datacenter performance analysis of a tensor processing unit. In *ISCA*, June 2017.
[28] S. W. Keckler et al. GPUs and the future of parallel computing. *IEEE Micro*, 31(5):7–17, Sept 2011.

[29] A. Krishna et al. Hardware acceleration in the IBM PowerEN processor: architecture and performance. In *PACT*, 2012.

[30] H. Langston, L. Greengard, and D. Zorin. A free-space adaptive FMM-based PDE solver in three dimensions. *Communications in Applied Mathematics and Computational Science*, 6(1):79–122, Aug. 2011.

[31] D. Lee et al. A distributed kernel summation framework for general-dimension machine learning. *Statistical Analysis and Data Mining*, 2013.

[32] J. Lee and H. Kim. TAP: A TLP-Aware Cache Management Policy for a CPU-GPU Heterogenous Architecture. 2012.

[33] S. Li et al. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO*, pages 469–480, Dec 2009.

[34] G. H. Loh et al. A processing-in-memory taxonomy and a case for studying fixed-function pim. 2013.

[35] M. H. Luke Durant, Olivier Giroux. Inside Volta: The world's most advanced data center GPU. In *Parallel Forall: NVIDIA Developer Blog*, Sept. 2017.

[36] J. Makino and H. Daisaka. GRAPE-8–an accelerator for gravitational N-body simulation with 20.5 Gflops/W performance. In *SC*, 2012.

[37] D. Malhotra and G. Biros. PVFMM: A Parallel Kernel Independent FMM for Particle and Volume Potentials. *Communications in Computational Physics*, 18(3):808–830, September 2015.

[38] D. Malhotra and G. Biros. Algorithm 967: A distributed-memory fast multipole method for volume potentials. *ACM Transactions on Mathematical Software*, 43(2):17:1–17:27, 2016.

[39] S. Mittal and J. S. Vetter. A survey of CPU-GPU heterogeneous computing techniques. *ACM Computing Surveys*, 47(4):69, 2015.

[40] T. Narumi et al. Fast calculation of electrostatic potentials on the gpu or the asic md-grape-3. *The Computer Journal*, 54(7):1181–1187, 2010.

[41] A. Pedram, A. Gerstlauer, and R. van de Geijn. Codesign Tradeoffs for High-Performance, Low-Power Linear Algebra Architectures. *IEEE TC*, 61(12):1724–1736, December 2012.

[42] W. Qadeer et al. Convolution Engine: Balancing Efficiency and Flexibility in Specialized Computing. In *ISCA*, 2013.

[43] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A cycle accurate memory system simulator. *IEEE CAL*, 10(1):16–19, 2011.

[44] Y. S. Shao et al. Co-designing accelerators and SoC interfaces using gem5-Aladdin. In *MICRO*, pages 1–12, 2016.

[45] D. E. Shaw et al. Anton 2: raising the bar for performance and programmability in a special-purpose molecular dynamics supercomputer. In *SC*, pages 41–53, 2014.

[46] D. Sozzo et al. A highly scalable and efficient parallel design of N-body simulation on FPGA. In *IPDPSW*, pages 241–246, 2017.

[47] G. Ventakesh et al. Conservation Cores: Reducing the Energy of Mature Computations. In *PACT*, 2012.

[48] Y. Yang et al. CPU-Assisted GPGPU on Fused CPU-GPU Architectures. In *HPCA*, 2012.

[49] R. Yokota, H. Ibeid, and D. Keyes. Fast multipole method as a matrix-free hierarchical low-rank approximation. *Computing Research Repository*, abs/1602.02244, 2016.

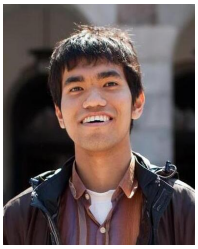
[50] D. Zhang et al. TOP-PIM: throughput-oriented programmable processing in memory. In *HPDC*, 2014.

[51] L. Zhao et al. Hardware support for accelerating data movement in server platform. *IEEE TC*, 56(6):740–753, 2007.

[52] L. Zheng et al. Heterogeneous reconfigurable design for TreePM N-body simulation. In *ICACT*, pages 442–446, 2017.

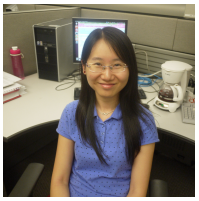
[53] Z. Zheng et al. Revealing feasibility of fmm on asic: Efficient implementation of n-body problem on fpga. In *IEEE International Conference on Computational Science and Engineering*, 2010.

[54] Y. Zhu and V. J. Reddi. Webcore: Architectural Support for Mobile Web Browsing. In *ISCA*, 2014.



Mochamad Asri is currently a Research Scientist at Facebook Reality Labs (FRL), Menlo Park. He received his Ph.D. in Electrical and Computer Engineering from The University of Texas at Austin in 2020. His research interests include heterogeneous system architectures, data movement optimizations, out-of-order micro-architecture, and high performance caching.

Dhairya Malhotra is a Research Scientist at the Flatiron Institute, New York. He received his Ph.D. in Computational Science, Engineering and Mathematics from the University of Texas at Austin in 2017. He was a postdoctoral associate at the Courant Institute of Mathematical Sciences from 2017 to 2020. His research interests include high performance computing, fast algorithms, numerical analysis and integral equations.



Jiajun Wang is a Hardware Engineer at Google. She received her Ph.D. in the Electrical and Computer Engineering (ECE) Department at the University of Texas at Austin in 2019. Her research interests include high performance codes evaluations and software hardware co-design to explore alternative algorithm/hardware scenarios.

George Biros is the W. A. "Tex" Moncrief Chair in Simulation-Based Engineering Sciences in the Oden Institute for Computational Engineering and Sciences and has Full Professor appointments with the departments of Mechanical Engineering and Computer Science (by courtesy) at the University of Texas at Austin. He received his Ph.D. in Computational Science and Engineering from Carnegie Mellon (2000). He was a postdoctoral associate at the Courant Institute of Mathematical Sciences from 2000 to 2003. Biros was among a team of researchers that won the IEEE/ACM SC03 and SC10 Gordon Bell Awards.

Lizy Kurian John is Cullen Trust for Higher Education Endowed Professor in the Department of Electrical and Computer Engineering at The University of Texas at Austin. She received her Ph.D. in computer engineering from The Pennsylvania State University. Her research is in the areas of computer architecture, multicore processors, memory systems, performance evaluation and benchmarking, workload characterization, and reconfigurable computing. She is an IEEE Fellow, ACM Fellow and a Fellow of the National Academy of Inventors (NAI).

Andreas Gerstlauer (S'97-M'04-SM'11) s a Professor of Electrical and Computer Engineering (ECE) at the University of Texas at Austin. He received a Ph.D. in Information and Computer Science from the University of California, Irvine in 2004, where he also was a Researcher before joining UT Austin in 2008. His research interests include system-level design, system modeling, design methodologies, and embedded hardware and software synthesis. His work has received several best paper nominations and best paper awards from, among others, DAC, DATE and HOST. He has been General and Program Chair for major conferences including ESWEEK, and he currently serves as Associate Editor for ACM TECS.