Cross-Platform, Cross-Framework Development of Hybrid-Parallel Matrix-Multiplication Codes

Vyuhita Bonthu
Indian Institute of Technology Dharwad
India
vyuhita.bonthu@iitdh.ac.in

Nikhil Hegde
Indian Institute of Technology Dharwad
India
nikhilh@iitdh.ac.in

Abstract

Matrix-multiplication is an important kernel in domains ranging from machine learning to high-performance computing. Developers devote significant time and effort to optimizing the matrix-multiplication kernel. In this paper, we simplify the development of optimized matrix-multiplication codes for various platforms and heterogeneous systems. We target codes for CPUs on x86 (AVX, AVX2) and ARM (Neon) platforms, as well as Nvidia GPUs and Jetson Nano. To achieve this, we employ a tool to generate a novel, hybrid-parallel, implementation of matrix-multiplication that exploits parallelism within a core, across cores, across nodes, and across GPU devices.

We create a specification of the recursive matrix-multiplication algorithm and feed it to the tool, which emits the implementation with an empty recursion base case. The base case is then filled in (manually) with pre-existing hand-optimized codes using AVX, AVX2, ARM Neon, OpenCilk, OMP, BLAS APIs for CPUs and GPUs. Experiments with different programming models (OpenCilk, OpenMP, library-based), and multiple floating-point precision inputs on CPU-based, GPU-based, and Jetson Nano show that: (i) The hybrid-parallel multi-GPU codes show an energy efficiency of up to 5.5× and also execute at 1.5× lower peak temperatures compared to the cuBLAS library-based GPU implementations. Furthermore, for double-precision data, the hybrid multi-GPU code on RTX5000 achieves better performance compared to the cuBLAS implementation. (ii) The recursive CPU implementations are faster, more energy-efficient, and also execute at lower peak temperatures compared to iterative CPU codes. We introduce a cost-of-ownership metric to analyse hybrid-parallel code in a multi-platform, multi-framework setting and show that executions on Jetson Nano are the most cost-effective.

CCS Concepts

• **Computing methodologies** → **Parallel programming languages**; **Distributed programming languages**.

Keywords

Matrix-multiplication, Parallel code generation, Heterogeneous computing, Energy efficiency, Cost of ownership

ICPE '26, Florence, Italy

ACM ISBN 979-8-4007-2325-4/2026/05
https://doi.org/10.1145/3777884.3797819

ACM Reference Format:
Vyuhita Bonthu and Nikhil Hegde. 2026. Cross-Platform, Cross-Framework Development of Hybrid-Parallel Matrix-Multiplication Codes. In *Proceedings of the 17th ACM/SPEC International Conference on Performance Engineering (ICPE '26), May 04–08, 2026, Florence, Italy.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3777884.3797819

1 Introduction

Matrix-Matrix multiplication is a core operation in algorithms that are widely used currently. Training a model in Machine learning, displaying sharper images in Computer Graphics, analyzing Social Network relations, in Scientific Computing, benchmarking the speed of a Supercomputer are a few important examples where matrix-multiplication is employed. Improving the speed of computation of matrix-multiplication has been the focus of mathematicians and computer scientists for many decades. Algorithmic improvements that yield lesser number of underlying multiply-add operations [12], scheduling improvements of underlying operations to yield lesser number of memory accesses to slow memory in modern computing systems with hierarchical memory [15, 19], provisioning of dedicated hardware for simultaneous execution of several such operations [18] are important directions in this regard. Orthogonal to these improvements, utilizing heterogeneous hardware resources, such as GPUs, tensor cores, TPUs, and vector units, in modern computing systems effectively lead to creation of frameworks such as domain-specific languages, runtime systems, and libraries.

Recent state-of-the-art frameworks such as Exo [13] and Exo2 [14] support the creation of hardware-accelerated, high-performance codes for diverse and rapidly evolving hardware. Developers *quickly* produce fast and efficient matrix-multiplication implementations using approaches such as: i) Python-like programming environments, ii) dedicated APIs with library-based development environment for GPUs and CPUs [1, 2, 5, 6, 10, 20, 21], iii) compilers to effectively map the matrix-multiplication operations to underlying hardware, and iv) programming systems to abstract the specification of computation from orchestrating the computation on underlying (often) complex hardware [17].

However, there is no single framework or tool that enables quick creation of high-performant matrix-multiplication codes targeted at a variety of platforms. Hence, this paper focuses on producing a cross-platform, multi-framework implementation of matrix-multiplication, which is efficient, which can be deployed on multiple platforms, and which can be developed quickly.

A single optimized implementation that delivers the same level of performance on a system with different hardware parallelism levels—within a core, across cores, between a host and one or more

targets, across multiple nodes—is challenging. In a novel application of a tool, D2P [11]—which researchers have shown to produce MPI+Cilk parallel codes from intuitive specifications for Dynamic-Programming (DP) algorithms only—we create a specification for matrix-multiplication and use the specification with D2P to produce corresponding parallel implementation *quickly*. The specification for matrix-multiplication assumes dense matrix computation and a recursive divide-conquer (non-Strassen like) matrix-multiplication algorithm. The generated parallel code has recursion base case empty and allows optimized kernels to be plugged-in. We refer to such a code as *hybrid-parallel* code. We augment hybrid-parallel code by plugging in hand-tuned, optimized kernels (CUDA, AVX, Arm Neon, OpenMP, OpenCilk, and codes using BLAS APIs for CPUs and GPUs) to create efficient, cross-platform, performance portable matrix-multiplication implementations quickly. We thus accelerate development of efficient matrix-multiplication implementations through hybrid-parallel codes. This is a novel approach compared to library-based [10, 21], High-level Programming Language based [13], and runtime-based [20] approaches to accelerate the development of high performant matrix-multiplication codes.

To understand the cost of this productivity gain, we adopt a multi-pronged evaluation approach factoring in runtime performance, energy consumption, and observed peak temperature while executing hybrid-parallel codes with single- and double-precision data. On a variety of hardware, such as DNN accelerators, GPUs, x86- and Arm-based multi-core CPU servers, Jetson Nano, and cloud-based server instances, we consider performance characteristics of matrix-multiplication codes exploiting parallelism at different scales within the system: i) within a core: SIMD/vectorized and multi-threaded codes, ii) across cores: shared-memory parallel programming models using OpenCilk, OpenMP tasks, iii) across nodes i.e. CPU and GPU. Through this, we present an analysis of: i) how precision affects runtime performance and energy consumption, ii) how runtime affects energy consumption and, iii) the impact of different frameworks aimed at rapid code development on energy consumption, execution time, and precision. This helps application developers make an informed choice of the system with appropriate programming abstraction and tools, for deploying their applications, which have matrix-multiplication as the underlying kernel. The key contributions are:

- We create specification for recursive matrix multiplication algorithm and use this specification with a tool, D2P, to generate partially complete parallel implementation. A complete, hybrid-parallel, implementation is then obtained with the optimized codes that we plug in.
- We measure the execution time, energy consumed, and peak temperature with different implementations executing on a variety of systems. We also present a novel cost-of-ownership analysis to understand the tradeoffs involved in multiplying matrices with single- and double-precision floating point inputs, with and without hybrid-parallel codes.
- We compare hybrid-parallel codes against reference implementations Exo [14], a DSL and associated compiler. That allows rapid development of efficient matrix-multiplication codes. We also consider state-of-the-art, reference CUDA codes [5].

Tests show that CUDA implementation (i.e. hybrid code obtained using D2P) of matrix-multiplication executing on multiple GPUs delivers the best performance on the RTX5000 GPU server with multiple GPU cards. On NVidia V100, A100, and H100-based servers, the cuBLAS-based codes perform better. However, energy consumption- and peak temperature-wise, hybrid-parallel codes are better on all the systems. They provide up to 5.5× energy saving, making it suitable for energy-conscious workloads. On hardware accelerator like Jetson Nano, CUDA implementations are up to 10× more energy efficient, despite being slower than high-end GPUs like the H100. Overall, the experiments on a variety of GPU servers indicate that hybrid-parallel codes are performance-portable and the energy consumption gets smaller with work distributed across available cards.

Increasing precision from single- to double-type floating point computation is associated with longer execution time as expected. This increase is also associated with increased energy consumption. Our study finds that Hybrid code achieves significantly better energy efficiency and also keeps peak temperatures similar or even lower than other implementations. The CPU-based experiments with hand-written and hybrid-parallel implementations show that, the peak temperatures of iterative codes are higher than that of recursive codes. The hand-optimized vector codes, OpenMP, and Cilk parallel codes exhibit higher peak temperatures than hybrid-parallel implementations. Overall, recursive codes show lower peak temperatures than iterative codes. Comparison with Exo, shows that hybrid codes show better scalability and also execute faster. Cost-of-ownership analysis with different systems show that the executions on Jetson Nano are the most cost-effective with least energy spent, in picoJoules, per bit computed. These findings highlight key trade-offs between development effort, execution speed, energy efficiency, peak temperature, and an overall cost-of-ownership across a range of hardware platforms.

2 Producing Parallel Code and Augmenting

Autogen and D2P: Autogen [7] is a framework that automatically discovers a recursive algorithm (and produces corresponding specification) from an iterative DP code, targeting a subclass of problems in the Fractal-DP class.

D2P [11] takes the recursive specification produced by Autogen, customizes it, and produces a template with MPI+OpenCilk code. D2P unrolls the recursion to create tasks dynamically, computes the read and write data regions associated with each task, and inserts the communication channel (among tasks) required to satisfy dependencies. The generated output is an MPI+Cilk parallel skeleton where the overall recursive structure and task decomposition are handled automatically. Autogen and D2P together create an end-to-end system where the user only needs to provide the iterative loop structure to obtain the distributed-memory parallel code. The base case of the recursion is left empty so that the user can plug-in an optimized kernel. In this paper, we find that the matrix multiplication problem can be casted as a problem similar to the Fractal-DP class and show that recursive matrix multiplication can also be expressed with specifications amenable for D2P's automatic parallel code generation.

$$\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \times \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix} = \begin{bmatrix} A_{00}B_{00} + A_{01}B_{10} & A_{00}B_{01} + A_{01}B_{11} \\ A_{10}B_{00} + A_{11}B_{10} & A_{10}B_{01} + A_{11}B_{11} \end{bmatrix}$$

```
1 A(<x, y, z>)
2 Parallel: A(<x00 y00 z00>)
3           A(<x01 y00 z01>)
4           A(<x10 y10 z00>)
5           A(<x11 y10 z01>)
6 Parallel: A(<x00 y01 z10>)
7           A(<x01 y01 z11>)
8           A(<x10 y11 z10>)
9           A(<x11 y11 z11>)
```

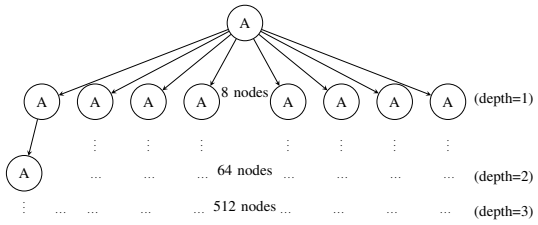

Figure 1: Recursive Matrix-Multiplication formulation (left), Specification for D2P (middle), Task decomposition (right)

Figure 1 presents a recursive formulation of matrix-multiplication, the corresponding specification (for D2P) to compute the product of two matrices A and B, and the recursion tree structure. The recursive formulation on the left, partitions the matrices into block matrices with four blocks of roughly equal size. It then computes the product of blocks to form the final product. The equivalent specification of this divide-conquer approach is in the middle. The specification requires that the first parameter of a method is the write parameter, and the remaining are the read parameters. Hence, the recursive method represented as `A` computes the product of two matrices `y` and `z` and adds the result to the data stored in matrix `x`. Essentially, method `A` represents a multiply-add operation[1]. The keyword `Parallel` is an annotation that tells that all the following method calls may be executed in parallel. This specification is represented with a task decomposition tree on the right. The nodes are tasks / recursive function calls and edges represent caller-callee relationship when the recursive method `A` is unfolded to a specific depth, three, to create tasks.

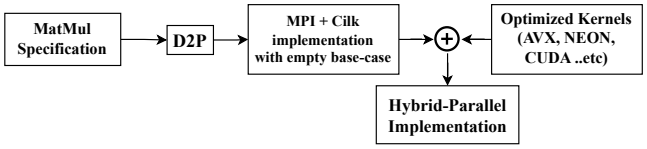

Figure 2: System overview

Starting from the specification, recursive MPI+Cilk parallel code is generated. Note that the base case of the recursive method is not part of the specification. In the code that is emitted from this specification, the base case implementation is left blank and is expected to be augmented with optimized kernels such as SIMD codes, CUDA kernels, or tiled codes. Also, note that for matrix sizes that are powers of two, the current specification works very well without extensions. With slight modifications to the specification, it can also be extended to support rectangular matrices. Not all specifications yield correct partial parallel codes and/or may be efficient. This study focuses on matrix multiplication only. However, the approach has been shown to work on recursive LU decomposition additionally and can be extended to any recursive linear algebra algorithms with the mentioned properties.

E.g., the specification for recursive matrix-multiplication can be written with the help of two recursive methods `A` and `B`, where method `A` implements the addition and method `B` implements the multiplication. This leads to inefficiencies when temporaries are used in code generation. D2P [11] showed that for all Dynamic Programming (DP) algorithms, specifications can be written and

[1]The syntactic details of symbols <, >, blankspace or comma can be ignored and have no special meaning.

```
void A(/* Method Signature */) {
  if (x->coords[0] == x->coords[2] && x->coords[1] == x->coords[3]) {
    // Optimized kernel: dense block using AVX-512
    for (int i = 0; i < size; i++) {
      for (int j = 0; j < size; j++) {
        float temp[16] = {0.0f};
        for (int k = 0; k < size; k += 16) {
          __m512 b   = _mm512_loadu_ps(&B[i*size + k]);
          __m512 c   = _mm512_loadu_ps(&C[k*size + j]);
          __m512 acc = _mm512_loadu_ps(temp);
          acc = _mm512_fmadd_ps(b, c, acc);
          _mm512_storeu_ps(temp, acc);
        }
        for (int k = 0; k < 16; k++)
          A[i*size + j] += temp[k];
      }
    }
    // Auto-generated recursive calls
    A_unroll(&x00, xData, parentTileIDx*4 + 0,
             &y00, yData, parentTileIDy*4 + 0,
             &z00, zData, parentTileIDz*4 + 0,
             callStackDepth + 1);
    ...
}
```

Figure 3: AVX code integrated with auto-generated code.

parallel codes can be generated. Figure 2 illustrates the system overview. The recursive matrix-multiplication specification is an input for D2P, which generates corresponding MPI+Cilk parallel code. The generated code has recursive method implementations with empty recursion base case, which is filled by the programmers with hand-optimized codes such as AVX, AVX2, and CUDA-C. The final augmented code is a hybrid-parallel code that is executed on SIMD-, multicore-, many-core machines, and GPUs.

Figure 3 shows how handwritten SIMD code for x86 (AVX512) is integrated with D2P generated code in the recursion base case. The code outside the box is D2P generated and we show only a part of the generated code for simplicity. Calls to `X_unroll` in the Figure, unrolls the recursion for the purpose of improving available parallelism. Following a similar approach, we use CUDA code to offload the computations to the GPU. This way we create a number of flavors of hybrid-parallel code where the base case implementations to multiply two matrices (of smaller sizes) are different. The complete list is shown in the Table 1.

Implementations that have `d2p_` prefix are hybrid-parallel codes i.e. generated by D2P and augmented with optimized base cases. Implementations that have prefix `hw_` are recursive, hand-written codes developed without assistance from D2P. Additionally, flavors that end with the suffix `_iter` represent iterative codes. Each of the above flavors have two sub-flavors corresponding to single- and

Table 1: Matrix-multiplication(MatMul) implementation variants evaluated in this study.

Implementation	Platform / Model	Description	Precision Variants
d2p_AVX256,d2p_AVX512, hw_AVX256, hw_AVX512	x86 SIMD CPU	Uses 256-bit and 512-bit vector registers; executes on multicore and many-core CPU nodes.	d2p_AVX256F,hw_AVX256F, d2p_AVX512D, hw_AVX512D
d2p_NEON, hw_NEON	Arm SIMD CPU	Uses Arm NEON vector registers; executes on multicore and many-core CPU nodes.	Single and double precision
d2p_CUDA, ref_CUDA	Single GPU	Uses CUDA kernel-based MatMul on GPU servers.	d2p_CUDAF, ref_CUDAF, d2p_CUDAD, ref_CUDAD
d2p_cuBLAS, ref_cuBLAS	Single GPU (Library-based)	Uses cuBLAS BLAS-like API for GPU.	d2p_cuBLASF,ref_cuBLASF, d2p_cuBLASD,ref_cuBLASD
d2p_MGPU, ref_MGPU	Multi-GPU	Uses CUDA across multiple GPUs.	d2p_MGPUF, ref_MGPUF, d2p_MGPUD, ref_MGPUD
ref_cuBLAS_MGPU	Multi-GPU (Library-based)	Uses cuBLAS for Multi-GPU.	ref_cuBLAS_MGPUF, ref_cuBLAS_MGPUD
Jetson_Nano	Embedded GPU	Uses Iterative CUDA implementation for Jetson Nano.	Single and double precision
d2p_cblas, cblas	CPU BLAS Library	Uses OpenBLAS API.	Single and double precision
d2p_MKL, MKL	CPU BLAS Library	Uses Intel MKL API.	Single and double precision
hw_CILK_AVX	CPU Parallel Runtime	Uses cilk_spawn for task parallelism with AVX vectorized recursion base case.	hw_CILK_AVXF, hw_CILK_AVXD
hw_OMP_AVX	CPU Parallel Runtime	Uses OpenMP task parallelism with AVX vectorized recursion base case.	hw_OMP_AVXF, hw_OMP_AVXD
hw_(ijk, kij, ikj, jik)_iter	CPU Iterative	Different loop order permutations.	Single and double precision
hw_Rec_Blocked	CPU Recursive	Blocking/tiling-based recursive implementation.	Single and double precision
NumPy	CPU (Python Library)	NumPy dot / matmul routines backed by optimized BLAS libraries.	Single and double precision
exo	DSL Compiler for x86 (CPU)	Python-like Exo code compiled to optimized C with AVX instructions.	Single and double precision

Table 2: Experimental hardware platforms.

Platform	Configuration
RTX Server	36 cores(2-socket) Intel Xeon Gold 6240C, 2.2MiB L1, 36MiB L2, 49.5MiB L3 cache, 2.60 GHz base, 150 W TDP; 2× NVIDIA Quadro RTX5000 GPUs (16 GB, 3,072 CUDA cores, 230 W).
DGX Server	8× NVIDIA V100 GPUs, 32 GB memory each; used for multi-GPU experiments.
GPU Cluster	Two compute nodes + master node: 4× A100 GPUs and 2× H100 GPUs (80 GB each); used for single- and multi-GPU experiments.
Jetson Nano	128 CUDA cores, 4 GB memory, 10 W maximum power draw.

double-precision inputs (having postfix F and D respectively.). Reference state-of-the-art implementations are ref_CUDAF, ref_CUDAD, ref_cuBLASF, ref_cuBLASD [5], exo [14].

3 Evaluation

We consider programming model, execution time, energy consumption, and peak temperature observed, and use some of these parameters to summarize with a cost-of-ownership metric for a particular implementation. Cost-of-ownership, typically employed as a data-centre design metric [4], is adopted here for evaluating the efficiency of deployed matrix-multiplication codes. AI workloads, which have matrix-multiplication as a kernel underneath, pervade resource consumption on Cloud-based and on-premise servers, HPC clusters, workstations, and edge devices. We believe that the choice of experimental systems shown in Table 2 together with the evaluation of 30 implementation flavors mentioned earlier helps developers and end-users to decide on a suitable system, abstraction, and precision to develop and deploy their applications.

Methodology. With d2p_X codes, we vary the recursion depth from 1 to 3 to create parallel tasks and vary the number of Cilk workers from 1 to 16 to execute these tasks. Depending on the recursion depth, we get 8, 64, and 512 tasks. We also vary number of MPI processes from 1 to 16 for launching multi-GPU based jobs. We observe that 8-tasks, 4-processes, and 4-cilk workers configuration yields the best execution times and energy consumption for both single- and double-precision input datasets. We measure energy consumption of CPU-based executions using perf events, which internally uses Intel RAPL [8] interfaces. For GPU-based executions, we use Nvidia-smi to measure GPU power consumption values. We multiply these values with the total execution time to obtain energy consumed by the GPU. We add to this the energy combined by the CPU to calculate the total energy consumed by both the GPU and CPU[3, 9]. We measure the peak temperature during execution using the sensors command for the CPU and Nvidia-smi for the GPU. Although power meters provide more accurate measurements, the readings from Intel RAPL and Nvidia-smi are sufficient in our studies as we focus on comparing these values across models. We ran all our experiments in an isolated environment, on bare-metal, where there were no active users (and other workloads), keeping the system conditions stable across different seasons spanning six months. This helped us minimize the variations that could come from changes in ambient temperature. Each experiment was run five times, and the standard deviation of the execution times was always under 3% of the mean.

3.1 Framework-Execution time-energy consumption trade-off

Figure 4 shows execution time and energy consumption measurements of different implementations for input of 16384 rows executing on RTX Server. In a win-win scenario, we observe that hybrid-parallel GPU implementation, d2p_MGPUD, i.e. D2P generated code with augmented base case for distributed execution on multiple GPUs, not only executes faster but also consumes less energy than reference multi-GPU ref_MGPUD, ref_cuBLAS_MGPUD counterparts (Figure 4, right). This is due to d2p_MGPUD distributing the workload more effectively on multiple GPUs and improved hardware utilization. We execute d2p_MGPUD with 4 MPI processes and base case of the recursion invokes CUDA kernels on distinct GPU cards available on the system. Analysis of d2p_MGPUD and reference implementations using performance counters and NVIDIA

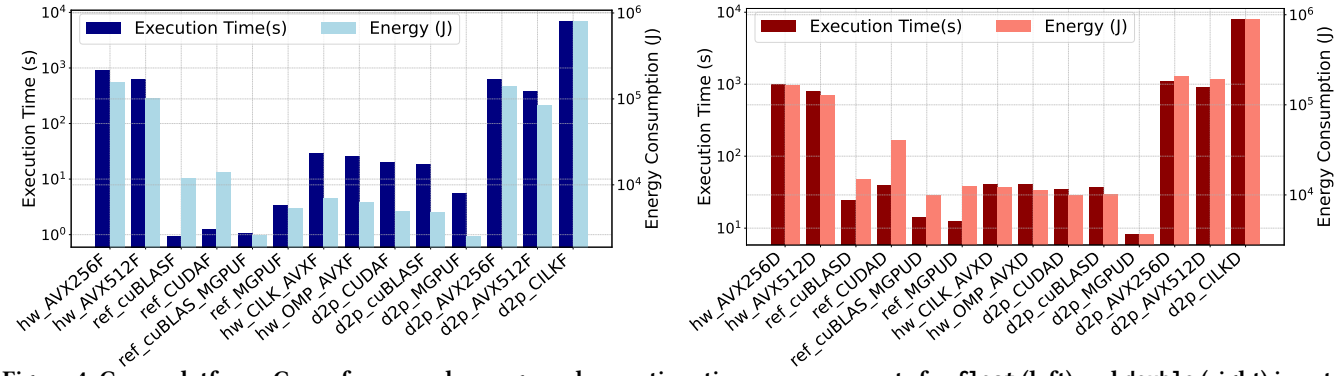

Figure 4: Cross-platform, Cross-framework energy and execution_time measurements for `float` (left) and `double` (right) inputs.

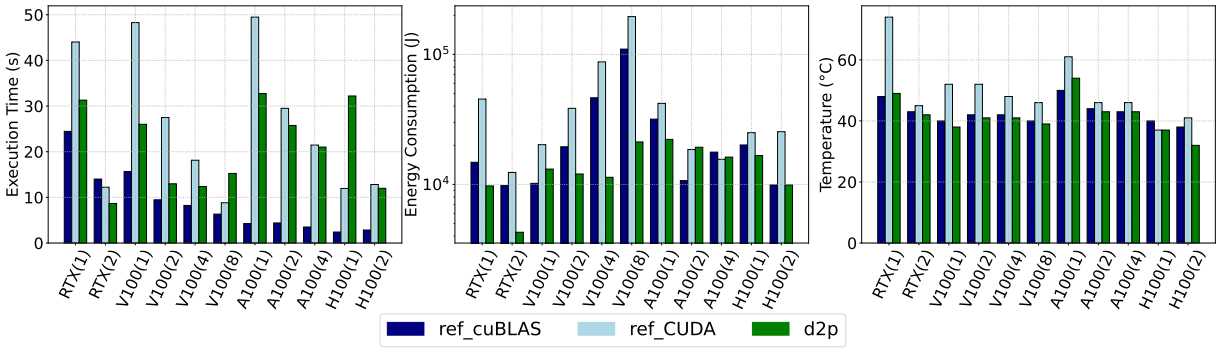

Figure 5: Comparison of Execution time, Energy and Temperature for Single- and Multi-GPU implementations.

Nsight Compute (`ncu`) shows that `d2p_MGPUD` has 1.2× higher memory throughput, 1.2× better L1 cache utilization, and improved GPU occupancy, indicating better resource usage and smaller execution time compared to native implementation. The smaller execution time in turn results in lower energy consumption values.

CPU-based hybrid-parallel codes also show interesting trends. With `float` inputs, AVX-based implementations (`d2p_AVX256F`, `d2p_AVX512F`) execute faster and consume lesser energy compared to codes developed without assistance from D2P i.e. hand-optimized native versions (`hw_AVX256F`, `hw_AVX512F`). This is due to dynamic task creation in hybrid-parallel codes, allowing better utilization of CPU vector units and reduction of CPU idle time. With `double` inputs, the opposite is true, where hybrid-parallel codes underperform due to higher memory bandwidth demands, showing over 23× more cache misses.

3.2 Comparing with Reference Implementations

This Section compares hybrid-parallel implementations with reference GPU and CPU implementations. In the GPU-based experiments, the reference codes `ref_cuBLAS, ref_CUDA` are used as baselines. The original reference CUDA implementation [5] uses only one GPU card even when multiple GPU cards are present on the server. Hence, we make minor changes to this version to create `ref_MGPUD, ref_cuBLAS_MGPUD`, implementations that distribute workload across available GPU cards. With `d2p`, the same hybrid-parallel code works with single and multiple GPU executions. We observe that `d2p` offers performance portability across multiple GPUs and outperforms hand-optimized `ref_CUDA` implementation. `ref_cuBLAS` outperforms both `d2p` and `ref_CUDA`. However, it is outperformed by `d2p` executing on RTX Server. This is because V100, A100 and H100 GPU based systems are optimized for throughput, whereas RTX5000 GPU is optimized for latency and has fastest clock among the GPU systems considered. `d2p` incurs runtime overheads and therefore performs slower than the reference codes on throughput-optimized systems. It is worth noting that `d2p` is the most energy efficient and shows lower peak temperature compared to the reference codes. These results are shown in Figure 5.

In the CPU-based experiments executed on the RTX server (without using the GPU cards), we compare hybrid-parallel codes with `exo` [13] and `NumPy` [16]. We consider D2P generated codes augmented with calls to MKL and OpenBLAS library APIs for comparison. Exo offers a Python-like development environment with support for calling intrinsics and specifying optimization policies, enabling rapid prototyping and creation of performant codes. Exo does not exhibit scaling improvement with increasing matrix size from 4k to 16k, although it shows good scalability for smaller matrix sizes as reported in [13]. Hybrid-parallel implementations perform better than Exo at larger scales and consume less energy. With single-precision inputs and smaller matrix sizes, `NumPy` shows the best throughput compared to the theoretical peak throughput achievable on the system. At 8192 rows and beyond, for both single- and double-precision inputs, `d2p_cblas` and `d2p_mkl` show the best throughput even outperforming their handwritten counterparts. These results are shown in Figure 6.

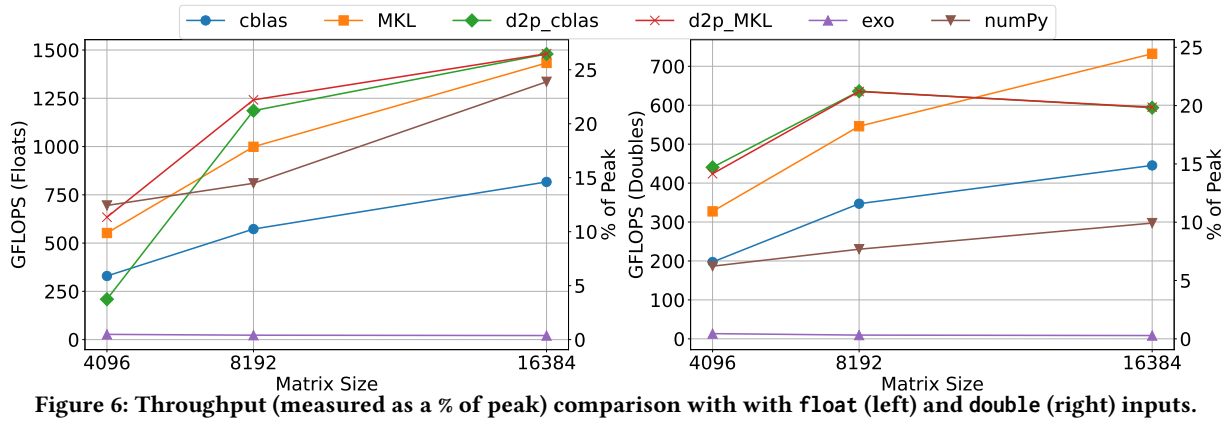

Figure 6: Throughput (measured as a % of peak) comparison with with `float` (left) and `double` (right) inputs.

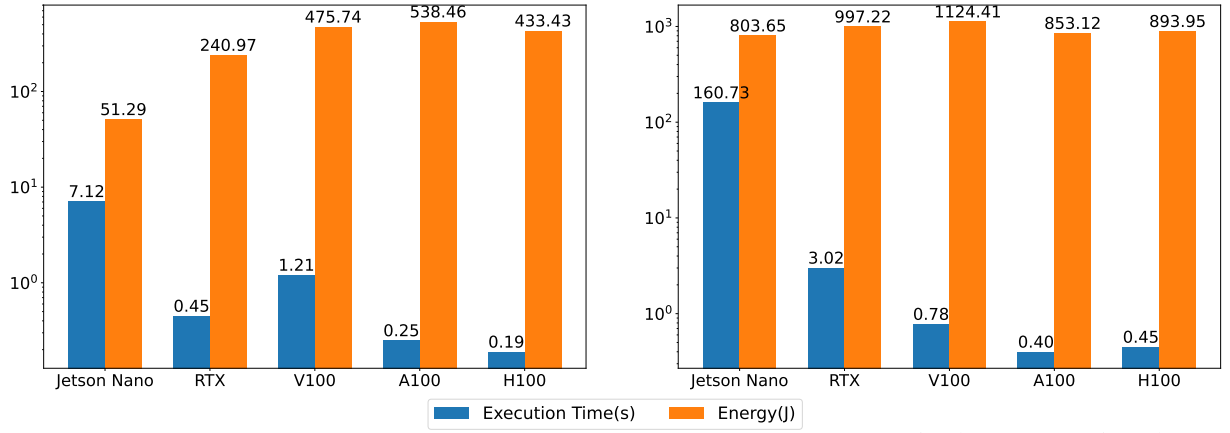

Figure 7: Execution time and Energy consumption of reference CUDA codes for `float` (left) and `double` (right) inputs.

Energy consumption on Jetson Nano: We also added the Jetson Nano to our tests so we could see how a hardware accelerator stacks up against powerful GPUs when it comes to energy usage. In these experiments, we used reference CUDA codes (`ref_cuBLAS`) on all platforms. Figure 7 shows the bar plot of energy consumed and execution time by the same native `ref_cuBLAS` code on 8192-sized matrices. For clarity, we show the actual numbers in seconds and Joules on top of the bars. As expected, the Jetson Nano execution is slow, about 37× slower than that on H100, in single-precision. However, Jetson Nano consumes 8.5× lesser energy. With double-precision inputs, Jetson Nano execution is slower, by around 400×, than that on A100. Energy consumption-wise, however, Jetson Nano outperforms the A100 by about 1.1×. Beyond 8k input matrix sizes, the Jetson Nano encounters memory overflow issues. This experiment shows that in scenarios with lower precision requirements and no strict latency requirements, the Jetson Nano is highly energy efficient when single-precision `float` data is used.

3.3 Comparing with Non-recursive codes on the CPU

In this Section we compare execution of recursive codes with iterative codes. For iterative codes, we consider the standard triple-nested-for loop with tiles/blocking. The block sizes were chosen based on the cache size of the system. Observation of peak temperature on the CPU tells an interesting story: In general, we observe significantly lesser temperatures with executions involving recursive implementations. We conduct these experiments on RTX Server (without using GPUs). As Figure 8 shows, `hw_AVX256f` to `d2p_AVX512f` have lower temperatures compared to `X_iter` implementations. The same trend is observed with double-precision inputs (RHS of the Figure). The recursive implementations also execute faster and consume lesser energy than their iterative counterparts. Figure 9 shows results. Analysis of the measurements using `perf` shows that this behavior is due to the cache-oblivious nature of recursive implementations and also balanced workload partitioning (divide-conquer), which creates smaller cache-friendly tasks reducing memory stalls and idle cycles. These tasks improve cache reuse and keep vector units active, thereby lowering execution time, energy consumption, and peak temperature. In contrast, the iterative implementations process larger fixed size blocks, resulting in unbalanced workloads, causing higher memory contention and idle cycles, which increases energy usage and temperature.

Another interesting observation is that doubles run at higher peak temperatures compared to the floats. This is observed because double-precision computations require twice the data width, increasing memory bandwidth demand and cache pressure, resulting

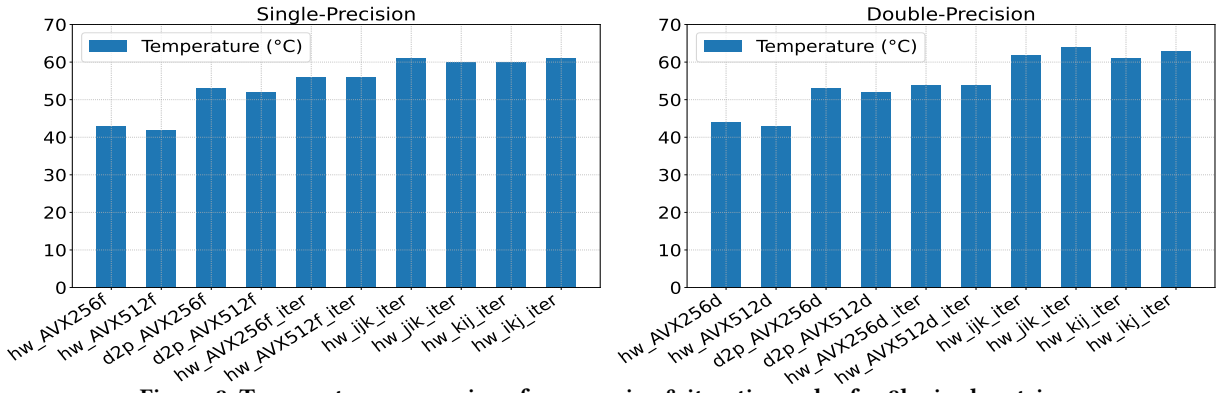

Figure 8: Temperature comparison for recursive & iterative codes for 8k-sized matrices.

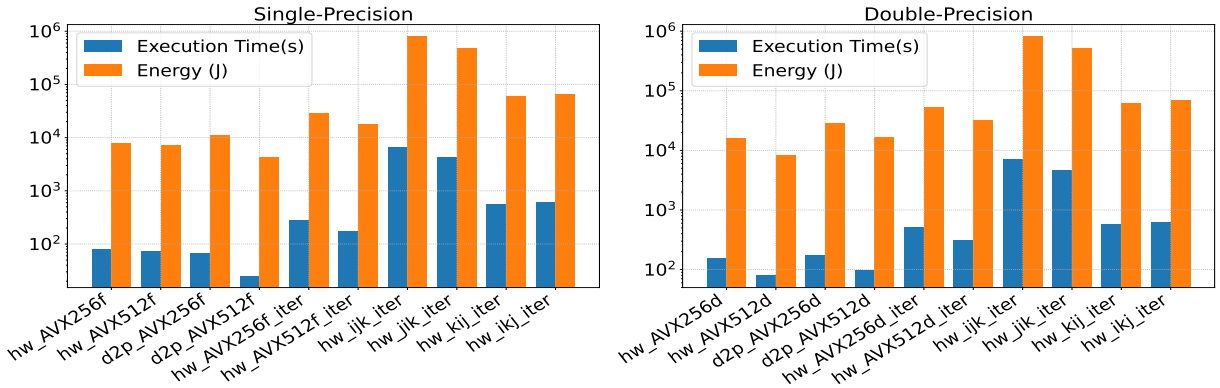

Figure 9: Performance & Energy comparison for recursive & iterative codes for 8k-sized matrices.

in higher CPU utilization, power draw, and peak temperature compared to floats. We also observe that, `d2p_AVX512f` is the fastest and the most energy-efficient for single-precision, while `hw_AVX512d` is the best in terms of both performance and energy-efficiency for double-precision.

Additionally, we also ran experiments with codes developed using Kokkos and ARM-Neon intrinsics. With Kokkos, we observe an overhead and makes it about 2.3× slower than native CPU code (and about 15× slower than hybrid-parallel CUDA codes). With ARM-Neon, we observe that the native NEON (`hw_NEON`) implementation is 7.1× faster than the hybrid-parallel, `d2p_NEON`, code for single-precision, and 6× faster for double-precision across all matrix sizes. Due to platform and permission limitations on cloud-based instances of ARM servers, we could not measure energy and temperature for these systems.

3.4 Cost-of-ownership (CO)

We now present a cost-of-ownership analysis that illustrates the amount of energy spent in computing a bit operation. Table 3 shows the results computed using Equation 1.

$$\text{CO} = \left(\frac{E}{N^3 \times p}\right) \times 10^{12} \quad \text{(pJ per bit operation)} \tag{1}$$

where, E is the total energy consumed (in joules), N is the dimension (input size) of matrix, p is the precision (in bits) of input data.

Table 3: Cost of ownership

Versions	picoJoules/bit
Jetson Nano	2.915
d2p_MGPU	11.681
d2p_AVX	30.356
d2p_CUDA	34.598
d2p_cuBLAS	35.521
hw_OMP_AVX	39.224
hw_CILK_AVX	42.020
hw_MGPU	43.929
hw_AVX	52.066
hw_cuBLAS	52.610
hw_CUDA	161.166
exo	2392.838

Traditionally, the cost-of-ownership is computed in dollars. In our study, we adopt a metric that is generally used in the semiconductor industry, picoJoules per bit. Aiming to provide an estimation of the amount of energy spent per bit of computation. This metric is free from geographical representations and cost structures (E.g., the selling price of a GPU server, manufactured in Asia, is different depending upon where it is sold.). Using picoJoules per bit as a metric provides a common representation that is universally applicable for a comparison of energy-efficiency across diverse regions. The data in Table 3 is computed for a p value of 32 (single-precision input), and N value of 8192. E is measured as explained

earlier in the methodology part of this Section. A lower value indicates more cost-efficient computation. The table shows that Jetson Nano and multi-GPU executions (`d2p_MGPU`) offer higher energy-efficiency. The hybrid single-GPU executions such as `d2p_CUDA` and `d2p_cuBLAS` have higher energy-efficiency compared to their native single-GPU CUDA implementations such as `hw_CUDA` and `hw_cuBLAS`. The CPU implementation like `d2p_AVX` exhibit reduced energy usage compared to other non-`d2p` AVX-Native counterparts like (`hw_OMP_AVX`, `hw_CILK_AVX` and `hw_AVX`). As our future work, we will extend our experiments for measuring temperature and energy consumption on Arm-based systems, FPGAs and TPUs.

4 Related Work

Approaches for optimizing matrix-multiplication that exploited the memory hierarchy include [19]. These techniques improve data locality through blocking and loop interchange. Earlier efforts in developing optimized matrix-multiplication kernels, especially as part of numerical libraries, appear in [2, 10]. While these approaches rely on hand-tuned kernels, our work looks at how much performance and efficiency can be achieved starting from a simple recursive formulation and generating hybrid-parallel code automatically. Compiler and language approaches such as Exo compilation [13] enable hardware-level optimization through higher-level abstractions. Exo allows custom hardware instructions, memories, and accelerator configurations to be specified in a Python-like code, enabling rapid development of matrix-matrix multiplication implementations. While our goal aligns with Exo, our approach differs in that the starting point for hybrid-parallel codes is an intuitive recursive specification of the matrix-multiply algorithm. Tool-based approaches reduce programmer effort by performing automatic dependency inference, parallelism extraction, and code generation [11]. However, [11] targets Dynamic Programming algorithms, whereas our work applies a similar automatic code generation approach to dense matrix-multiplication using a recursive formulation. While prior work primarily evaluates performance, our work evaluates execution time, energy consumption, and peak temperature together on CPUs, CUDA GPUs, and hybrid-parallel codes, providing a more complete comparison across the implementations considered in this paper.

5 Conclusions

Hybrid codes are attractive when quick deployment is needed as they drastically reduce the time it takes to get an efficient implementation. In many cases, they are not just faster to develop but they also deliver better performance, lower energy consumption, and reduced peak temperatures compared to handwritten alternatives. These advantages, combined with their ease of deployment across CPUs and GPUs, make hybrid codes a practical choice with a low cost-of-ownership. While hybrid codes may not always match the performance of cuBLAS and hand optimized CUDA implementations, their portability, ease of development, energy efficiency, and lower observed peak temperatures make it a practical alternative.

References

[1] Philip Alpatov, Greg Baker, Carter Edwards, John Gunnels, Greg Morrow, James Overfelt, Robert van de Geijn, and Yuan-Jye J. Wu. 1997. PLAPACK: parallel linear algebra package design overview. In *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing* (San Jose, CA) *(SC '97)*. Association for Computing Machinery, New York, NY, USA, 1–16. doi:10.1145/509593.509622

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. 1992. *LAPACK's user's guide.* Society for Industrial and Applied Mathematics, USA.

[3] Mauricio Fadel Argerich and Marta Patiño-Martínez. 2024. Measuring and Improving the Energy Efficiency of Large Language Models Inference. *IEEE Access* 12 (2024), 80194–80207. doi:10.1109/ACCESS.2024.3409745

[4] Luiz Andre Barroso and Urs Hoelzle. 2009. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines* (1st ed.). Morgan and Claypool Publishers.

[5] Simon Bøhm. 2023. *SGEMM CUDA: High-Performance Matrix Multiplication in CUDA.* https://github.com/siboehm/SGEMM_CUDA

[6] Jaeyoung Choi, Jack J. Dongarra, Roldan Pozo, and David W. Walker. 1992. *LAPACK Working Note 55: ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers.* Technical Report. USA.

[7] Rezaul Chowdhury, Pramod Ganapathi, Jesmin Jahan Tithi, Charles Bachmeier, Bradley C. Kuszmaul, Charles E. Leiserson, Armando Solar-Lezama, and Yuan Tang. 2016. AUTOGEN: automatic discovery of cache-oblivious parallel recursive algorithms for solving dynamic programs. *SIGPLAN Not.* 51, 8, Article 10 (Feb. 2016), 12 pages. doi:10.1145/3016078.2851167

[8] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. 2010. RAPL: memory power estimation and capping. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design* (Austin, Texas, USA) *(ISLPED '10)*. Association for Computing Machinery, New York, NY, USA, 189–194. doi:10.1145/1840845.1840883

[9] Eva García-Martín, Crefeda Faviola Rodrigues, Graham Riley, and Håkan Grahn. 2019. Estimation of energy consumption in machine learning. *J. Parallel Distrib. Comput.* 134, C (Dec. 2019), 75–88. doi:10.1016/j.jpdc.2019.07.007

[10] Kazushige Goto and Robert Van De Geijn. 2008. High-performance implementation of the level-3 BLAS. *ACM Trans. Math. Softw.* 35, 1, Article 4 (July 2008), 14 pages. doi:10.1145/1377603.1377607

[11] Nikhil Hegde, Qifan Chang, and Milind Kulkarni. 2019. D2P: from recursive formulations to distributed-memory codes. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 22, 22 pages. doi:10.1145/3295500.3356205

[12] Steven Huss-Lederman, Elaine M. Jacobson, Anna Tsao, Thomas Turnbull, and Jeremy R. Johnson. 1996. Implementation of Strassen's algorithm for matrix multiplication. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing* (Pittsburgh, Pennsylvania, USA) *(Supercomputing '96)*. IEEE Computer Society, USA, 32–es. doi:10.1145/369028.369096

[13] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. 2022. Exocompilation for productive programming of hardware accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 703–718. doi:10.1145/3519939.3523446

[14] Yuka Ikarashi, Kevin Qian, Samir Droubi, Alex Reinking, Gilbert Louis Bernstein, and Jonathan Ragan-Kelley. 2025. Exo 2: Growing a Scheduling Language. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 426–444. doi:10.1145/3669940.3707218

[15] Tze Meng Low and Robert A. van de Geijn. 2004. *An API for Manipulating Matrices Stored by Blocks.* Technical Report. Department of Computer Sciences, The University of Texas at Austin, Austin, TX, USA.

[16] Travis E. Oliphant. 2015. *Guide to NumPy* (2nd ed.). CreateSpace Independent Publishing Platform, North Charleston, SC, USA.

[17] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 519–530. doi:10.1145/2491956.2462176

[18] Martin D. Schatz, Robert A. van de Geijn, and Jack Poulson. 2016. Parallel Matrix Multiplication: A Systematic Journey. *SIAM J. Sci. Comput.* 38, 6 (Jan. 2016), C748–C781. doi:10.1137/140993478

[19] Robert A. van de Geijn and Jerrell Watts. 1997. SUMMA: Scalable Universal Matrix Multiplication Algorithm. *Concurrency: Practice and Experience* 9, 4 (1997), 255–274.

[20] Field G. Van Zee, Ernie Chan, Robert A. van de Geijn, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. 2009. The libflame Library for Dense Matrix Computations. *Computing in Science and Engg.* 11, 6 (Nov. 2009), 56–63.

[21] Wikipedia contributors. 2026. *Intel Math Kernel Library.* https://en.wikipedia.org/wiki/Math_Kernel_Library