Horses for Courses: Understanding System Development Parameters for the Matrix Multiplication Kernel

***Abstract*—Matrix multiplication is a fundamental kernel in domains ranging from machine learning to high-performance computing to benchmarking computer architectures. Developers devote significant time and effort to optimize matrix multiplication kernels. In this paper, we present a systematic study of the performance, speed of development, and energy consumption of multiple flavours of implementations of the matrix multiplication kernel on multicore, manycore, SIMD-, and accelerator- based hardware. We implement hybrid-parallel codes which are a mix of hand-optimized and tool-generated codes and compare them with standard implementations of the matrix multiplication kernel. For various matrix sizes, performance, development effort, and energy consumption of AVX, AVX2, Cilk, OMP, CUDA, and hybrid parallel codes are measured. We implement the recursive matrix multiplication algorithm using task-parallel paradigms available for multicore systems. We make the recursion base case implementation tunable and plug hand-optimized AVX, AVX2, and CUDA code. In some implementation versions, the recursive algorithm is implemented using a code generation tool, which left the recursion base case empty to allow for plugging in of hand-optimized code. We then plug hand-optimized kernel in such implementations. Scalability experiments with different programming models on CPUs, multiple GPUs, Arm server on cloud, and Jetson Nano show that CUDA implementations not only offer the lowest latency but also are energy efficient in a multi-GPU setting, where the work is distributed. Interestingly, for extended-precision codes, we observe that tool-generated code augmented with hand-optimized CUDA kernel executed on multi-GPU systems performs better than native CUDA code in all aspects of latency, development speed, and energy consumption.**

I. Introduction

Matrix Multiplication is a central operation in algorithms that are used every day now. E.g., training a model in Machine learning, displaying sharper images in Computer Graphics, predicting weather patterns in Scientific Computing, analysis of Social Network relations, studying cache designs in Computer Architecture, 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 multiply-add operations [1], improvements in scheduling of those operations to yield lesser number of memory accesses to the slow memory in modern computing systems with hierarchical memory [2], [3], provisioning of dedicated hardware for simultaneous execution of several such operations form the broad spectrum of developments [4]. Orthogonally, researchers have created abstractions and tools for developers to *quickly* produce fast matrix multiplication implementations: dedicated APIs for matrix multiplication in library-based development environment [5]–[10], compilers to effectively map the operations to underlying hardware [11], and programming systems to abstract the specification of computation from orchestrating the computation on underlying (often) complex hardware [12]. This paper focuses on various factors involved in system development —development to deployment on specific systems–of matrix multiplication in an attempt to understand the tradeoffs involved. Specifically, on a variety of hardware, such as DNN accelerators, GPUs, x86- and Arm-based multicore CPU servers, performance characteristics at different scales within the system are considered: i) within a core: SIMD/vectorized and multithreaded codes, ii)across cores: shared-memory parallel programming models using Cilk, OpenMP tasks, iii) hybrid codes i.e. CPU+GPU including automatically generated codes. Latency, energy consumed, and speed of code development are measured for single- and double-precision data to understand the tradeoffs of execution time vs. {precision, energy consumption}, precision vs. energy consumption, speed of code development vs. {energy consumption, execution time, and precision}. The goal is to help application developers make an informed choice of the workhorse (system with appropriate programming abstraction and tools) for deploying their applications, which have matrix- multiplication as the underlying kernel.

Modern computing systems provision for parallelism within a core, across cores, across a host within the same compute node hosting one or more 'devices' or accelerators, and across a host with multiple nodes. A single optimized implementation that delivers the same level of performance with system configurations of different parallelism levels is challenging. Developers dedicate significant time and effort to optimize matrix multiplication kernels to squeeze out every ounce of performance from the hardware resources available. We create 'reasonably' efficient implementations of matrix multiplication *quickly* with the help of a tool, D2P [13], that produces parallel code exploiting parallelism across cores and across hosts on multiple compute nodes. Furthermore, D2P provisions for exploiting parallelism within a core and in a node with host-device mode of computing. This *elastic* parallel code is created from a specification, which we develop assuming dense matrix computation and a recursive divide-conquer (non-Strassen) matrix multiplication algorithm. A noteworthy feature of the

elastic code is that the recursion base case is left empty to allow for optimized kernels to be plugged-in. Consequently, we plug in hand-tuned, optimized kernels (CUDA, AVX, Arm Neon, locality-optimized) and augment to create efficient, performance portable matrix multiplication implementations quickly. It is important to note that the development of hand-tuned optimized code is a one-time effort (often available) and plugging-in and deploying augmented code on different systems is relatively easier.

Figure 1 shows performance comparison of the augmented code (indicated as hybrid code) and native CUDA-C code for multiplying two matrices. The hybrid code is a mix of MPI+Cilk+CUDA-C code. Interestingly, we see that in a compute node with multiple GPU cards, native CUDA-C implementation executes slower compared to the hybrid code. Performance counters from NVIDIA Nsight Compute indicate why the hybrid code performs better than the native code. The hybrid code records nearly 56 million DRAM active cycles, whereas the native code displays a much higher 9.54 billion cycles. The hybrid code also achieves better overall DRAM throughput (8.24% versus 5.11%) and memory throughput (21.95% compared to 18.54%). The L1 cache throughput for hybrid code is higher than native code (43.32% versus 37.08%) but the hybrid code records comparatively lesser L2 cache throughput (4.09% compared to 4.73%). On the other hand, the registers per thread used for hybrid code (36) are lesser than native code (46), leading to greater parallelism . With slightly improved GPU occupancy (100.49% versus 100.13%), the hybrid code maximizes resource utilization more effectively, leading to faster execution and superior performance overall. We attribute this to the dynamic parallelism created in hybrid code, which is able to utilize the additional GPU resources efficiently. The recursion base cases (with CUDA kernels) in the hybrid code are independent tasks that get executed with the help of distinct MPI ranks. Note that both hybrid code (in the recursion base case) and native code use the same optimized CUDA kernel for matrix multiplication. This experiment shows that performance portability across multiple GPU cards is achievable with minimal programmer effort and surprisingly, tool-produced code can be faster than hand-optimized code. Is the performance portability free of cost? does data precision have an impact on performance and/or energy consumption? to understand these, we do a systematic analysis of the performance (execution time), development speed, and energy consumption of different flavours of matrix multiplication kernel implementations on a variety of systems, including Jetson Nano and cloud-based server instances. The key contributions are:

- We create specifications for recursive matrix multiplication algorithm and produce corresponding hybrid-parallel code.
- We create a number of matrix multiplication implementations, including AVX256, AVX512, Arm Neon, OpenMP, Cilk, Jetson Nano, CUDA-C, Open BLAS-based, Tensorflow, and Pytorch based codes. We plug-in some of these codes into the recursion base case to produce efficient hybrid-parallel code.
- we measure and analyze performance, energy consumed, precision and speed of development to understand trade-offs involved in multiplying matrices with different precision data, on systems at different scales, with different programming systems, under different energy consumption scenarios.

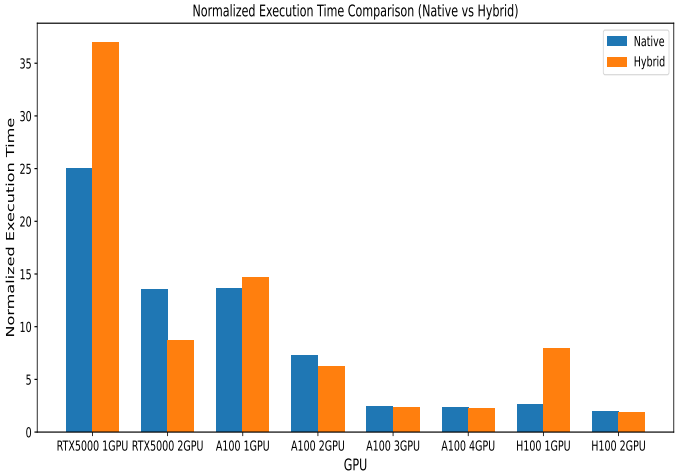


Fig. 1: Native CUDA-C vs. Augmented CUDA-C comparison.

Tests across various systems show that overall, CUDA implementations executing on multiple GPUs deliver the best performance, offering the lowest latency. W.r.t. energy consumption of distributed workloads, surprisingly, codes executing on RTX5000 GPUs show that energy consumption get smaller with work distributed across available cards. In contrast, energy consumption increase with work distributed across available A100 and H100 GPUs. As expected, increasing precision from single- to double- floating point computation is associated with longer execution time, which in turn is associated with increased energy consumption. The cuda codes on Jetson Nano w.r.t to performance are 7.5x slower than RTX5000, 28.48x slower than A100 and 37.47x slower than H100. And w.r.t. energy consumption, Jetson Nano consumes 5.2x less energy than RTX5000, 10.49x less energy than A100 and 8.4x less energy than H100. Interestingly, for CPU Parallel codes like OMP and Cilk, OMP and Cilk on x86 are 1.4x and approximately 1.5x slower than runs on ARM for both single- and double- precision. The SIMD codes on ARM, w.r.t. performance, are 1.9x slower than x86, whereas Hybrid SIMD codes on ARM are 20x slower than Hybrid SIMD x86 for single precision. For double precision, Native SIMD on ARM is 4x slower than x86, whereas SIMD codes on ARM are 23.4x slower than Hybrid SIMD x86 for single precision. A similar trend is observed in SIMD codes on ARM system w.r.t precision, additionally, native SIMD codes consistently perform better than Hybrid SIMD codes in single and double precisions. Also, ARM virtual machine instance SIMD codes are 1x slower than ARM Bare metal instances. Tests on Colab for TPU and GPU usage using TensorFlow and PyTorch show

that runs on TPU perform well when compared to GPU. TensorFlow code using TPU is faster than PyTorch using TPU for single and double precisions. In contrast, PyTorch GPU code is faster than TensorFlow GPU code. Due to the limited access to performance counters on cloud-based instances and Google Colab, energy consumption was not measured.

The remainder of the paper is organised as follows: Section 2 lays out the background of producing parallel code and augmenting; Section 3 discusses the analysis of system development parameters, Section 4 provides related work, and lastly, Section 5 concludes.

II. Producing Parallel Code and Augmenting

Table I presents a recursive approach and the corresponding specification to compute the product of two matrices A and B. The approach 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. This divide-conquer approach is represented with a method specification on the right. The specification requires that the first parameter of a method is the write parameter, and 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. The details of symbols `<`, `>`, blankspace or comma to separate arguments can be ignored and have no special meaning. The keyword `Parallel` is an annotation that tells that all the method calls on the same line may be executed in parallel. As we can see, there is a one-to-one correspondence between the specification and the divide-conquer approach on the left. The specification shows further details indicating multiplication of y_{ij} and z_{ij} and adding the result to x_{ij}. Note that the base case of recursive method is not part of the specification. Also, note that for matrix sizes that are powers of two, the method works very well without extensions. 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 locality optimized codes.

Starting from the specification, the MPI+Cilk code generation requires parallelism inference, data decomposition, task creation, and communication insertion. Importantly, the generated code will be correct only if the specification corresponding to the algorithm satisfies the *inclusive* and *intersection* properties. We omit these and provide a system-level overview here and refer the readers to [13] for details. The specification for recursive matrix multiplication can be written in multiple ways e.g., with the help of two recursive methods `A` and `B`, where method `A` implements the addition and method `B` implements the multiplication. However, some specifications may lead to inefficiencies when temporaries are used and some may not satisfy the inclusive and intersection properties. [13] showed the existence of those properties in all Dynamic Programming (DP) algorithms. Whereas, in this paper we consider a flavor of matrix multiplication algorithm, create the corresponding spec and generate efficient hybrid parallel code.

Figure 2 illustrates the system overview. The specification is input to D2P to generate MPI+Cilk parallel code. This code has recursive method implementations with empty recursion base case, which is filled by the programmers. The resulting augmented code is a hybrid-parallel code that is executed on SIMD, multicore, manycore, and GPUs.

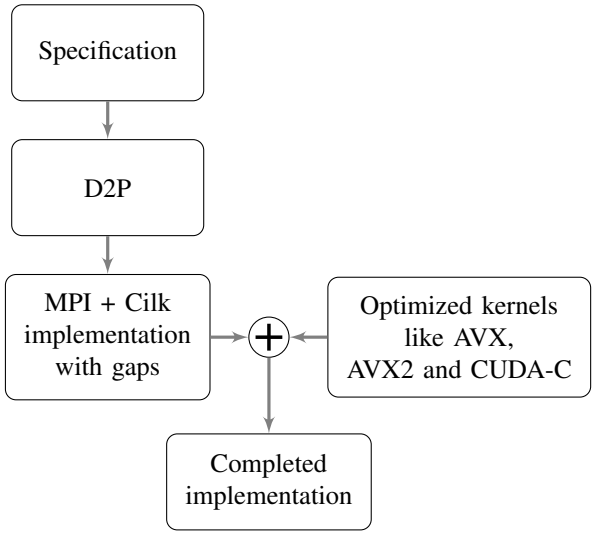


Fig. 2: System overview

Figures 3 and 4 show how handwritten SIMD codes for x86 and Arm systems are integrated with D2P generated code to produce hybrid code. We use intrinsics—C-style functions provided to manipulate the vector registers without writing assembly code—available from Intel and Arm. The same technique is applied to create a number of flavors of hybrid parallel code where the base case implementations to multiply two matrices (of smaller sizes) are different. The below list shows a complete list of different flavors of matrix multiplication implementations:

- `d2p_avx, d2p_avx2, avx, avx2`: code using the 256-bit and 512-bit vector registers on x86 architecture. Meant for execution on multicores, manycores across compute nodes.
- `d2p_neon, neon`: code using Arm's Neon vector registers. Meant for execution on multicores, manycores across compute nodes.
- `d2p_cuda, cuda`: code using single GPU to accelerate matrix multiplication. Meant for execution on GPU servers.
- `d2p_cublas, cublas`: code using single GPU and a BLAS-like library API available to multiply matrices. Meant for execution on GPU servers.
- `d2p_cudamultigpu, cudamultigpu`: code using multiple GPUs to accelerate matrix multiplication. Meant for execution on GPU servers.
- `omp` recursive implementation of matrix multiplication using OpenMP tasks. Meant for execution on multicore CPU servers.

TABLE I: Algorithm (left), Specification (right)

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

```
A(<x,y,z>)
Parallel: A(<x00 y00 z00>) A(<x01 y00 z01>)
          A(<x10 y10 z00>) A(<x11 y10 z01>)
Parallel: A(<x00 y01 z10>) A(<x01 y01 z11>)
          A(<x10 y11 z10>) A(<x11 y11 z11>)
```

```
void A( /* Method Signature */ ) {
    if ((x->coords[0] == x->coords[2])
    && (x->coords[1] == x->coords[3])) {
```

Optimized terminating case with AVX.

```
    // Loop over rows of A
       for (int i = 0; i < size; i++) {
           // Loop over columns of B
           for (int j = 0; j < size; j++) {
           // Initialize the result vector to zero
               __m256i result = _mm256
                                _setzero_si256();
               // (process 8 elements at a time)
               for (int k = 0; k < size; k += 8) {
                   // Load 8 elements from B and C
                   __m256i b = _mm256_loadu_si256
                   ((__m256i*)&B[i * size + k]);
                   //similarly for c
                   // Multiply and accumulate
                   result = _mm256_add_epi32(result
                   , _mm256_mullo_epi32(b, c));
               }
               //Accumulate result into A
           }
       }
```

```
     // Auto Generated Code to handle
    A_unroll(&x00, xData, parentTileIDx * 4 + 0,
    &y00, yData, parentTileIDy * 4 + 0, &z00,
    zData, parentTileIDz * 4 + 0,
    callStackDepth + 1);//similarly for x01,x10,x11
    A_unroll(&x00, xData, parentTileIDx  * 4 + 0,
    &y01, yData,parentTileIDy * 4 + 1, &z10,
    zData, parentTileIDz  * 4 + 2,
    callStackDepth + 1);//similarly for x01,x10,x11
}
```

Fig. 3: Augmenting with AVX code.

```
void A( /* Method Signature */ ) {
  if (x->coords[0] == x->coords[2])
  && (x->coords[1] == x->coords[3])) {
```

Optimized terminating case with NEON.

```
    // Loop over rows of A
     for (int i = 0; i < size; i++) {
       // Loop over columns of B
       for (int j = 0; j < size; j++) {
         // Initialize the result vector to zero
         float32x4_t result =
                          vdupq_n_f32(0.0f);
         // process 4 elements at a time)
         for (int k = 0; k < size; k += 4) {
           // Load 4 elements from B and C
           float32x4_t b = vld1q_f32
                           (&B[i * size + k]);
           // Similarly for C and then Multiply
           //and accumulate
           result = vmlaq_f32(result, b, c);
         }
         // Accumulate the result into A
       }
     }
```

```
// auto-generated code to handle
  A_unroll(&x00, xData, parentTileIDx * 4 + 0,
  &y00, yData, parentTileIDy * 4 + 0, &z00,
  zData, parentTileIDz * 4 + 0,
  callStackDepth + 1);//similarly for x01,x10,x11
  A_unroll(&x00, xData, parentTileIDx * 4 + 0,
  &y01, yData,parentTileIDy * 4 + 1, &z10,
  zData, parentTileIDz * 4 + 2,
  callStackDepth + 1);//similarly for x01,x10,x11
}
```

Fig. 4: Augmenting with Arm Neon code.

- `Cilk` recursive implementation of matrix multiplication using Cilk. Meant for execution on multicore CPU servers.
- `pytorch_cpu`, `pytorch_gpu`, `pytorch_tpu` implementation using the Pytorch framework on multicore CPU, GPU, and TPU.
- `tensorflow_cpu`, `tensorflow_gpu`, `tensorflow_tpu` implementation using the Tensorflow framework on multicore CPU, GPU, and TPU.
- `jetson_nano` iterative implementation of matrix multiplication using CUDA on Jetson nano.

In the above list, flavors that do not have `d2p_` prefix are implementations that are not hybrid parallel i.e. are not codes generated by D2P. They are handwritten optimized implementations using recursive algorithm. Also, each of the above flavors have two sub-flavors one each corresponding to single- and double-precision data.

III. Analysis of System Development Parameters

We consider the most significant parameters involved in the system development: performance in terms of execution time, energy consumption, speed of implementing the code. These three broadly summarize the cost of ownership of a particular implementation. Speed of implementation is a metric that is custom defined for software systems e.g. based on the number of features implemented, based on number of lines of code written, etc. Here, we go with a broad notion that vector codes (using intrinsics or assembly) are the slowest and most difficult to develop and tool generated codes are the fastest and least time consuming to develop. In particular the following is considered w.r.t. speed of development: `SIMD` $<$ `tiled` $<$ `GPU` $\leq$ `jetson_nano` $\leq$ `OMP` $\leq$ `Cilk` $<$ `pytorch` $\leq$ `tensorflow` $<$ `d2pxx`. Cost of ownership [14] is usually defined for data center designs. We introduce it here as a metric for evaluating the efficiency

of matrix multiplication as deployed. Considering the pervasiveness of running Gen-AI workloads (and the matrix-multiplication operation running underneath) on systems on Cloud-based, on premise-servers, HPC clusters, workstations, edge devices, and on personal computing systems, we believe the following choice of hardware and the implementation flavors mentioned earlier help developers and end-users to decide on a suitable system, framework, and precision to develop and deploy their applications.

A. System Configuration

We have conducted tests on multiple systems:

- Intel(R) Xeon(R) Gold 6240C CPU, 36 physical cores in total in a 2-socket configuration, 2.2MiB L1 cache, 36MiB L2 cache and 49.5MiB L3 cache. Core base frequency is kept at 2.60GHz and the TDP is 150W (Thermal Design Power, which is the average power in watts, that the processor consumes while running at base frequency with all cores active under a complex workload). This system also has two NVIDIA QUADRO RTX 5000 GPUs, each with 16 GB memory and 3,072 CUDA cores, with a maximum power draw of 230W. Evaluated SIMD, CPU-only, Tensorflow-GPU, Tensorflow-CPU, Pytorch-CPU, Pytorch-GPU, BLAS-based, CUDA BLAS based codes on this server. Hybrid parallel codes are also evaluated on this server.
- A cluster of two compute nodes and a master node. One of the compute nodes has four A100 cards and the other has two H100 cards. Each with a memory capacity of 80GB. Single GPU and Multigpu codes including using CUDA BLAS libraries are evaluated on these servers.
- Jetson Nano has 128 Cuda cores, each with a memory of 4GB and a maximum power draw of 10W.
- We also employed an AWS Graviton3 processor `c16_large`, with 64 cores. This is a cloud-based instance offering VM environment. We have used this to run Arm Neon codes. We have also run experiments on `c16_metal`, the baremetal version and observed that the overhead of VM environment is insignificant.
- Google Colab to test TPU codes in TensorFlow and PyTorch frameworks.

During execution, we unrolled the recursion to a depth of 1 to create parallellism for tasks in D2P. This resulted in 8 tasks and we used 4 processes (MPI ranks) to measure the parameters mentioned. 4 processes were considered after we empirically found that the 4-process configuration yielded the best performance (lowest latency). We have used perf events for measuring the energy consumption (in Joules) of CPU-based implementations. For measuring GPU energy consumption, we have mainly used Nvidia-smi to monitor GPU power consumption, multiplied it with total execution time to obtain energy consumed by the GPU, and perf to measure the CPU's overall package energy. Then, we added both energies to calculate the total energy consumed by the GPU and CPU during the code's execution. Our attempt to measure energy consumption on the the ARM instance on AWS was unsuccessful due to the limited access to performance counters.

B. Discussion

For most codes outside the numerical computing world, double-precision may be an overkill. Even single-precision `float` data may be over provisioning. In such scenarios, a latency vs. development speed or energy efficiency vs. development speed tradeoff would be more appropriate. Our study shows on multicore execution with `float` data, OpenMP (OMP) codes deliver almost the same level of performance as that with SIMD codes. In Figure 5, the rhs part shows this result. OMP codes are much easier to develop and compiler optimizations vectorize the code effectively. Also, we observe that tool-generated hybrid code augmented with AVX performs slightly better than the native code. If high-precision is necessary, `double` inputs are to be considered. The lhs part of the Figure shows the results.

Figure 5 also shows the results with GPU execution. Executions on single and multiple-GPUs, with or without CuBLAS provided APIs, D2P augmented and native are shown. The left half of each part (lhs or rhs) in the figure shows this. We observe that GPU executions consume less energy compared to CPU executions. This is mostly because of the much lesser execution time (latency) on GPUs. It is interesting to note that executions on multiple GPUs consume less energy compared to executions on single GPU. This can be counter-intuitive at first but is due to the overall reduction in execution time resulting from the distribution of workload on multiple GPUs. Also as expected, increased precision is associated with increased latency and energy consumption.

In a win-win scenario, we see that tool assisted code implementation yields not only faster development speed but also faster execution. We see that the hybrid parallel code implementation for multi-GPU systems is not only performance portable but executes faster compared to native CUDA code. Figure 6 shows the results. Also, because of the reduced execution time, hybrid parallel code execution consumes lesser energy. D2P multigpu code was executed with 4 MPI ranks and the base case of the recursion had CUDA-kernel to distribute the workload across available GPU cards on the system. The native code was a plain CUDA code and had kernel to distribute the workload across available GPU cards. The executions were performed with varying number of GPU cards on RTX5000, A100, and H100 systems (Figure 1 shows the results with RTX5000 card). We see that the hybrid parallel code outperformed native code on these systems for double precision numbers by achieving higher total memory throughput while having slightly lower compute throughput. There was no significant performance impact from this slight variation in compute throughput. Whereas for single precision values, the native code outperformed the hybrid code in terms of memory and computation throughput. As a result, a similar behaviour as doubles is not seen in floats. Another interesting observation is that the difference in the latency between the cudaSgemm and CudasingleGPU implementations vanishes

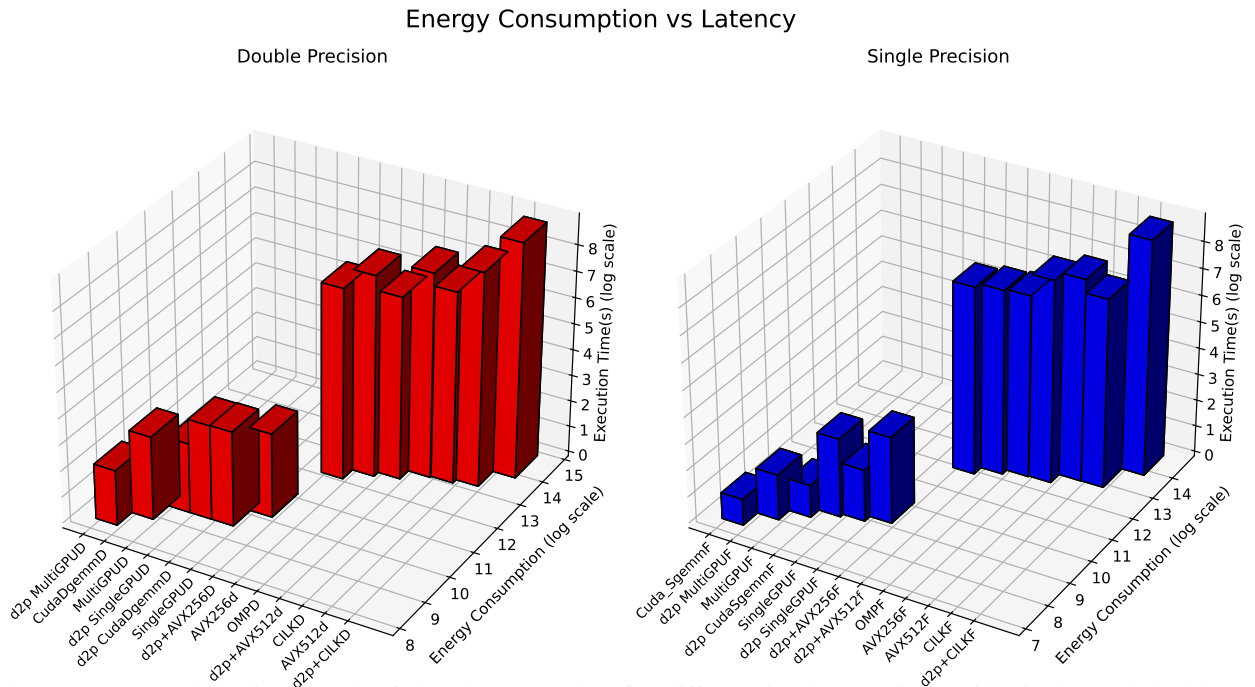


Fig. 5: Energy-Execution_Time-speed_of_development plots for different implementations with single- and double-precison inputs.

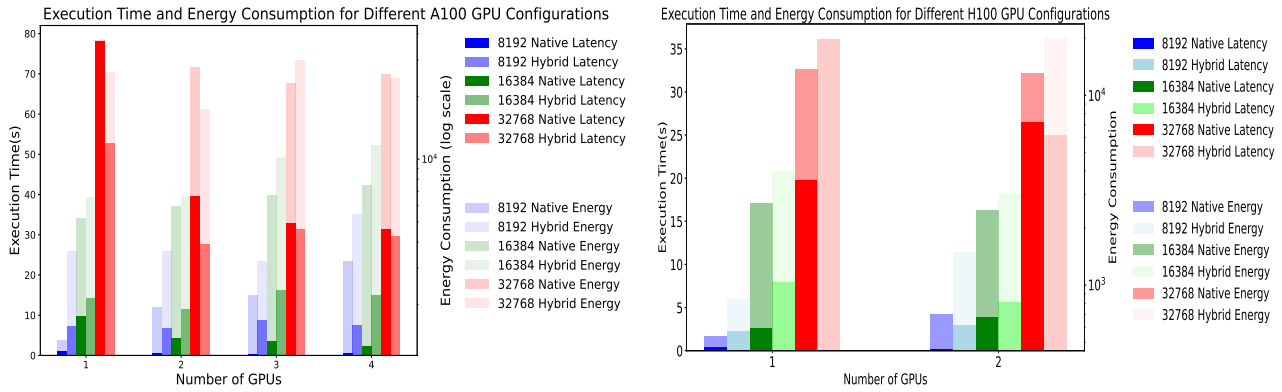


Fig. 6: Native CUDA-C vs. Augmented CUDA-C comparison on A100 and H100 GPU servers

from 4k to 8k.To understand this behaviour, we profiled the codes using profilers mentioned in previous section. Through the profiled information, we found that the computational intensity of both codes almost doubles from 4k to 8k to 16k, and for 16k-sized matrices, we see that cudaSgemm executes faster than cudasingleGPU. From this, we infer that the Cuda library-based approach is beneficial for matrix sizes greater than or equal to 8k. We continue the discussion on GPU executions with a focus on energy consumption next.

a) Energy consumption on GPUs and Jetson Nano: Figure 7 shows the energy consumption when we execute native matrix-multiplication code with 8k input matrix size. We use a single GPU card on each of the GPU-servers in these experiments. The figure shows the data for both single- and double-precision inputs. With single-precision input data, Jetson Nano execution is 37× slower than the most powerful H100 GPU execution. In contrast, the Jetson Nano is energy efficient compared to H100 by 8.5×. With double precision input data, Jetson Nano code is slower by 48.7× compared to RTX100, 133× compared to A100, and only 8.4× compared to H100. We observe that H100 performance falls below that of A100 for double-precision data and are doing further study to understand the reason. Energy consumption-wise, Jetson Nano is 1.7× more efficient compared to the best performing A100 GPU for double-precision inputs. Beyond 8k input matrix sizes, we start seeing overflow in Jetson Nano. This experiment shows that in a scenario with lower precision requirement and no strict latency requirement, Jetson Nano can be very energy efficient when single-precision `float` data is chosen.

b) TPU executions: We explored TPU performance using TensorFlow and PyTorch for matrix multiplication on Google Colab, comparing it to GPU v4 and CPU for matrix

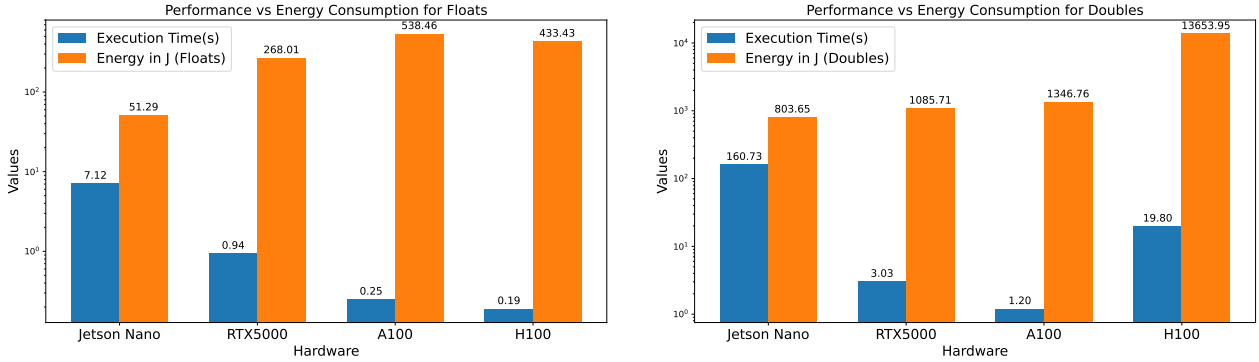
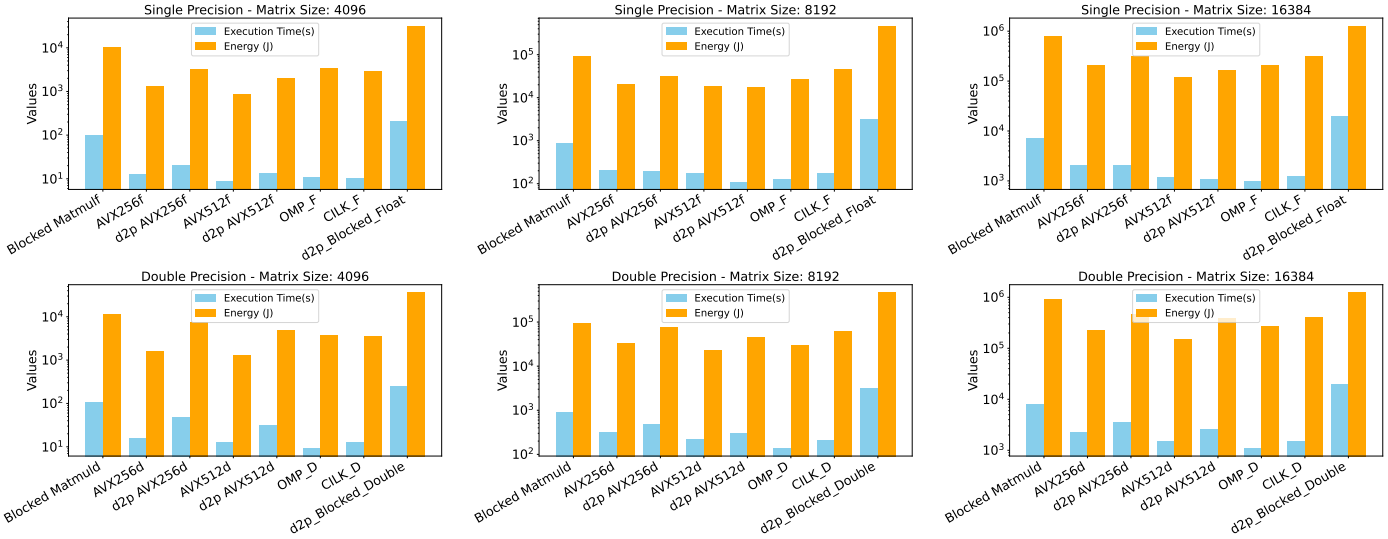


Fig. 7: Performance and Energy consumption of Cuda codes on different Hardwares



(a) 4k (b) 8k (c) 16k

Fig. 8: Performance characteristics on CPU cores for sizes 4k to 16k

sizes varying from 4k to 16k as before. Due to restricted hardware access, power consumption details were not available. For a 16k matrix, PyTorch showed that the TPU was 74x faster than the CPU and 1.2x faster than the GPU for single precision, and 84x faster than the CPU and 10x faster than the GPU for double precision. TensorFlow results indicated that the TPU was 6x faster than the GPU and 435x faster than the CPU for single precision, and 280x faster than the CPU and 18x faster than the GPU for double precision. The findings focus on the trade-offs among latency and energy consumption in multi-GPU setups and the performance of TPUs for matrix operations. Figure 9 shows the results.

c) multicore execution on x86 and Arm: We begin with multicore execution on x86 architectures. We have evaluated blocked (tiled), OMP, Cilk, AVX, AVX2, D2P with (tiled, AVX, and AVX2) variants on the RTX5000 system (we have named this system as RTX5000 but not used the GPU card in these experiments.). Figure 8 shows the details.

As we increase the matrix size from 4k to 16k, the better-performing and energy-efficient programming model shifts. In the case of single precision, AVX512f is energy-efficient at 4k, whereas the OMP and D2P integrated with AVX512 become faster at 16k. On the other hand, for double precision, the OMP code constantly has lower latency, whereas AVX512d remains energy-efficient across all matrix sizes. Figure 10 shows the results of executing on Arm based instance on AWS. Figure 11 shows the trend observed on ARM systems, i.e., Native NEON code performs better than Hybrid NEON code for all the matrix sizes for single and extended precision.In figure 10, observe that OMP and CILK codes on ARM perform better than x86.

We have also explored library-based methods like CBLAS and MKL implementations, where MKL outperformed with significantly better execution time and lower energy consumption than CBLAS. We also implemented the hybrid implementations, d2p_cblas and d2p_MKL, which showed improved performance and energy consumption relative to CBLAS. However, they still are less efficient than MKL. The AVX-based implementations combined with Cilk or OpenMP, exhibited the highest execution times and energy consumption

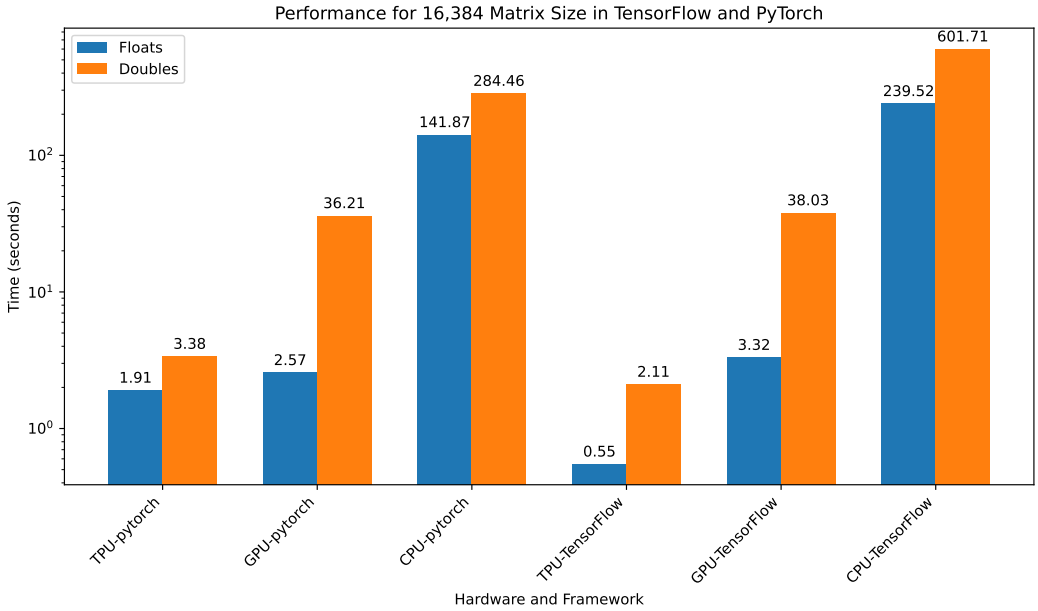


Fig. 9: Colab TPU results using TensorFlow and PyTorch

among all methods. The table II displays the detailed analysis for each implementation. Apart from the above methods, we have also leveraged Kokkos framework to explore performance portability aspect. We observed that KOKKOS introduces an abstraction overhead, which results in it being 2.3x slower than native CPU and 15x slower than native CUDA codes, it is discussed in detail in the [15].

Codes	Execution Time (s)	Energy Consumed (J)
CBLAS_D	15.83	4753.77
MKL_D	9.15	2452.98
d2p_cblasD	14.82	3940.68
d2p_MKLD	14.95	3953.05
AVX+ CILKD	39.83	12037.66
AVX+ OPENMPD	40.31	11156.25

TABLE II: Comparison of execution time and energy consumption across different implementations for 16k-sized matrices.

Table III displays the cost of ownership details.Traditionally, the cost of ownership is computed in dollars.In our study, we adopted a metric that is generally used in the semi-conductor industry : picoJoules per bit. Aiming to provide an estimation of the amount of energy spent per bit of computation. This metric provides a standardised approach that takes into account the various geographical representations and cost structures. Using this metric provides a common representation that is universally applicable for a comparison of energy efficiency across diverse regions. The data in Table III shows that harnessing accelerators like the Jetson Nano and exploiting the benefits of multi-GPU setups offer higher energy efficiency. In contrast, other implementations like AVX-Native, CILK, and OMP have high energy consumption. Optimized implementations like D2p AVX substantially reduce energy usage.

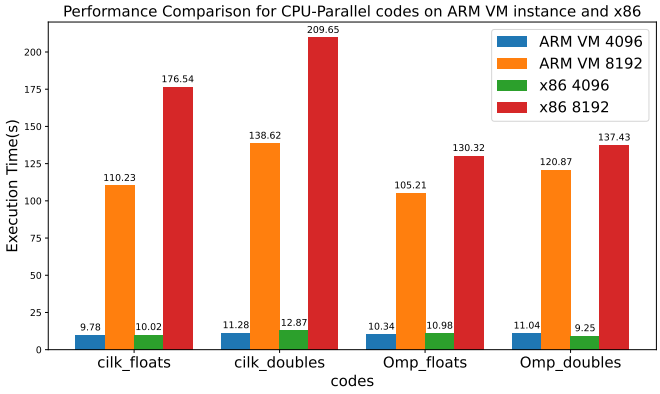
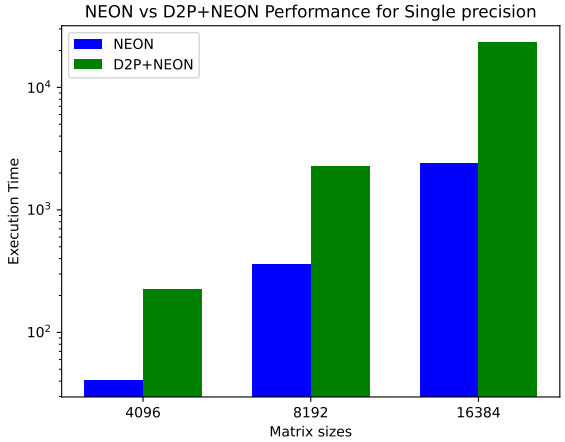


Fig. 10: Performance of CPU Parallel codes on ARM vs x86.

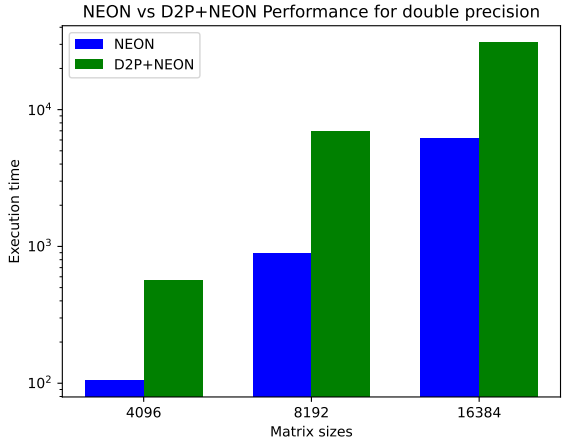
The performance peak percentages across the different hardware architectures display apparent differences in their ability to maximise efficiency as shown in Table IV. The A100 demonstrates strong scalability in hybrid code, especially with multiple GPUs, reaching a peak of 24.47% and outperforming its native counterpart. On the other hand, H100 performs well for native single-GPU execution, achieving 36.64%, but it reduces for multiple gpus, indicating less scalability in multi-GPU setups. In contrast, the Quadro RTX 5000 performs poorly in single-GPU execution (3.14%) but benefits significantly from hybrid parallelism in multi-GPU configurations, achieving 9.07% peak performance. A common takeaway from these results is that hybrid parallelism yields better performance in multi-GPU setups, as seen with all the hardware, while single-GPU performance is maximised in native execution, especially with the H100.

As our future work, we will extend our experimentation on

Fig. 11: Native Vs Hybrid using NEON intrinsics



(a) Single precision



(b) Double precision

clusters with GPU nodes and power measurements on Arm based systems and TPUs.

Versions	picoJoules/bit
D2p AVX	1019.98
D2p cuda	69.51
D2p Sgemm	66.88
D2p multigpu	40.03
OMP	1501.29
CILK	2587.76
Jetson Nano	2.91
AVX-Native	6915.32

TABLE III: Cost of ownership

Hardware	GPUs	Native (%)	Hybrid (%)
A100	1	9.29	13.76
A100	2	18.37	26.14
A100	3	21.97	23.04
A100	4	23.06	24.47
H100	1	36.64	20.11
H100	2	8.87	9.40
Quadro RTX	1	3.136	2.125
Quadro RTX	2	5.822	9.069

TABLE IV: Peak Performance Percentages for Native and Hybrid Codes on Different Hardware

IV. Related Work

Traditional approaches that exploited the memory hierarchy include [2], [3]. They aim at improving the data locality by blocking and loop interchange. Earlier efforts in optimizing matrix multiplication kernels that have primarily focused on library development, are shown in various studies [2], [5], [7]–[9] and for GPUs there are optimised frameworks such as [16]. Other studies, including [17]–[20], have implemented new algorithms to optimize sparse matrix-matrix and matrix-vector multiplication kernels on GPUs. On the other hand for dense matrix-vector computations we have have [21], [22] Furthermore, a GPU and CPU-GPU systems framework has been developed, as outlined in [23]. There are compiler and language approaches that address the optimization of the kernels. One of them is exo compilation [11], in which compiler backend decisions are externalised and are brought into the realm of the programmers. This approach, employs a Python-like code that rewrites library functions into hardware-level optimised code. This approach is similar to that employed in domain-specific language Halide [12], which mainly targets image processing kernels, where matrix-vector operations dominate. Halide targets single-node, multi-node, and GPU-based systems. Vectorisation is also an approach to optimise the performance, some of the works include [24] which presents effective vectorisation techniques on Xeon Phi processors. Tool based approaches reduce the programmer effort by doing automatic dependency inference, parallelism extraction, and/or code generation/execution [13], [25]. [13] only targets Dynamic programming algorithms and [25] provides an API based approach and developers are required to re-engineer the code using the APIs. Hierarchical tiling for improved superscalar performance [26] and automatic variable blocking for dense linear algebra algorithms [27] highlight the advanced techniques that improve the execution efficiency of matrix operations by optimising how data is accessed and stored in memory. Further research has been done in [28] on nonlinear array layouts for hierarchical memory systems. Furthermore, an experimental comparison of cache-oblivious and cache-conscious programs demonstrates the benefits of these approaches in parallel algorithm design [29]. The automatic generation of block-recursive codes, emphasizing memory optimization, is also studied in [30].Some studies have compared the energy efficiency of GPUs with FPGAs [31] and examined how the efficiency of matrix multiplication methods varies with the density of multi-GPU systems [32]. Besides, there is also study on performance evaluation on level-3 operations in CUBLAS [33]. Additionally, comparisons between FPGAs and GPUs for memory intensive and task that have less memory usage are shown in [34].In addition, previous research

on scientific and engineering subroutine libraries for vector processors has given important insights that guide modern optimization techniques [35].

V. Conclusions

Our tool-generated implementation approach offers a simpler approach to achieve performance portability and energy efficiency across multiple cores, accelerators and GPU cards. Although domain experts may find it difficult at first to create a specification as input, we find that the recursive algorithms' close resemblance to the spec effectively overcomes this obstacle. Experimenting with various Hardware configurations, such as DNN accelerators and GPU-enabled single and multi-card servers, demonstrated the efficacy of our tool-based approach. Our work provides a peek into the energy consumption, latency, and speed of development to help developers deploy their matrix-multiplication based applications on an appropriate workhorse.

References

[1] S. Huss-Lederman, E. M. Jacobson, A. Tsao, T. Turnbull, and J. R. Johnson, "Implementation of strassen's algorithm for matrix multiplication," in *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, 1996, pp. 32–es.

[2] T. M. Low, R. A. Van de Geijn, and F. W. Note, *An API for manipulating matrices stored by blocks*. Computer Science Department, University of Texas at Austin, 2004.

[3] R. A. Van De Geijn and J. Watts, "Summa: Scalable universal matrix multiplication algorithm," *Concurrency: Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1997.

[4] M. D. Schatz, R. A. van de Geijn, and J. Poulson, "Parallel matrix multiplication: A systematic journey," *SIAM J. Sci. Comput.*, jan 2016. [Online]. Available: https://doi.org/10.1137/140993478

[5] Math kernel library. [Online]. Available: "https://en.wikipedia.org/wiki/Math_Kernel_Library"

[6] P. Alpatov, G. S. Baker, H. C. Edwards, J. A. Gunnels, G. Morrow, J. Overfelt, and R. A. van de Geijn, "Plapack parallel linear algebra package design overview," *ACM/IEEE SC 1997 Conference (SC'97)*, pp. 29–29, 1997.

[7] K. Goto and R. Van De Geijn, "High-performance implementation of the level-3 blas," *ACM Transactions on Mathematical Software (TOMS)*, vol. 35, 2008.

[8] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker, "Scalapack: A scalable linear algebra library for distributed memory concurrent computers," in *The Fourth Symposium on the Frontiers of Massively Parallel Computation*. IEEE Computer Society, 1992, pp. 120–121.

[9] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney *et al.*, *LAPACK users' guide*. SIAM, 1999.

[10] F. G. V. Zee, E. Chan, R. A. van de Geijn, E. S. Quintana-Ort'ı, and G. Quintana-Ort'ı, "The libflame library for dense matrix computations," *Comput. Sci. Eng.*, vol. 11, no. 6, pp. 56–63, 2009. [Online]. Available: https://doi.org/10.1109/MCSE.2009.207

[11] Y. Ikarashi, G. L. Bernstein, A. Reinking, H. Genc, and J. Ragan-Kelley, "Exocompilation for productive programming of hardware accelerators," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2022, pp. 703–718.

[12] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *Acm Sigplan Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[13] N. Hegde, Q. Chang, and M. Kulkarni, "D2p: From recursive formulations to distributed-memory codes," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–22.

[14] L. A. Barroso, U. Hölzle, and P. Ranganathan, *The datacenter as a computer: Designing warehouse-scale machines*. Springer Nature, 2019.

[15] M. Atif, M. Battacharya, P. Calafiura, T. Childers, M. Dewing, Z. Dong, O. Gutsche, S. Habib, K. Knoepfel, M. Kortelainen *et al.*, "Evaluating portable parallelization strategies for heterogeneous architectures in high energy physics," *arXiv preprint arXiv:2306.15869*, 2023.

[16] A. Abdelfattah, D. Keyes, and H. Ltaief, "Kblas: An optimized library for dense matrix-vector multiplication on gpu accelerators," *ACM Transactions on Mathematical Software (TOMS)*, vol. 42, no. 3, pp. 1–31, 2016.

[17] K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the efficiency of gpu algorithms for matrix-matrix multiplication," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2004, pp. 133–137.

[18] R. Dorrance, F. Ren, and D. Marković, "A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on fpgas," in *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, 2014, pp. 161–170.

[19] S. Dalton, L. Olson, and N. Bell, "Optimizing sparse matrix—matrix multiplication for the gpu," *ACM Transactions on Mathematical Software (TOMS)*, vol. 41, no. 4, pp. 1–20, 2015.

[20] Y. Nagasaka, A. Nukada, and S. Matsuoka, "High-performance and memory-saving sparse general matrix-matrix multiplication for nvidia pascal gpu," in *2017 46th International Conference on Parallel Processing (ICPP)*. IEEE, 2017, pp. 101–110.

[21] N. Fujimoto, "Dense matrix-vector multiplication on the cuda architecture," *Parallel Processing Letters*, vol. 18, no. 04, pp. 511–530, 2008.

[22] R. Nath, S. Tomov, T. T. Dong, and J. Dongarra, "Optimizing symmetric dense matrix-vector multiplication on gpus," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–10.

[23] W. Liu and B. Vinter, "A framework for general sparse matrix–matrix multiplication on gpus and heterogeneous processors," *Journal of Parallel and Distributed Computing*, vol. 85, pp. 47–61, 2015.

[24] X. Tian, H. Saito, S. V. Preis, E. N. Garcia, S. S. Kozhukhov, M. Masten, A. G. Cherkasov, and N. Panchenko, "Effective simd vectorization for intel xeon phi coprocessors," *Scientific Programming*, vol. 2015, no. 1, p. 269764, 2015.

[25] F. G. Van Zee, E. Chan, R. A. Van de Geijn, E. S. Quintana-Orti, and G. Quintana-Orti, "The libflame library for dense matrix computations," *Computing in science & engineering*, vol. 11, no. 6, pp. 56–63, 2009.

[26] L. Carter, J. Ferrante, and S. F. Hummel, "Hierarchical tiling for improved superscalar performance," in *Proceedings of 9th International Parallel Processing Symposium*. IEEE, 1995, pp. 239–245.

[27] F. G. Gustavson, "Recursion leads to automatic variable blocking for dense linear-algebra algorithms," *IBM Journal of Research and Development*, vol. 41, no. 6, pp. 737–755, 1997.

[28] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi, "Nonlinear array layouts for hierarchical memory systems," in *Proceedings of the 13th international conference on Supercomputing*, 1999, pp. 444–453.

[29] K. Yotov, T. Roeder, K. Pingali, J. Gunnels, and F. Gustavson, "An experimental comparison of cache-oblivious and cache-conscious programs," in *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, 2007, pp. 93–104.

[30] N. Ahmed and K. Pingali, "Automatic generation of block-recursive codes," in *European Conference on Parallel Processing*. Springer, 2000, pp. 368–378.

[31] S. Mittal and J. S. Vetter, "A survey of methods for analyzing and improving gpu energy efficiency," *ACM Computing Surveys (CSUR)*, vol. 47, no. 2, pp. 1–23, 2014.

[32] P. Zhang and Y. Gao, "Matrix multiplication on high-density multi-gpu architectures: theoretical and experimental investigations," in *High Performance Computing: 30th International Conference, ISC High Performance 2015, Frankfurt, Germany, July 12-16, 2015, Proceedings 30*. Springer, 2015, pp. 17–30.

[33] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana-Orti, "Evaluation and tuning of the level 3 cublas for graphics processors," in *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 2008, pp. 1–8.

[34] B. Betkaoui, D. B. Thomas, and W. Luk, "Comparing performance and energy efficiency of fpgas and gpus for high productivity computing,"

in *2010 International Conference on Field-Programmable Technology*. IEEE, 2010, pp. 94–101.

[35] R. C. Agarwal, F. G. Gustavson, J. McComb, and S. Schmidt, "Engineering and scientific subroutine library release 3 for ibm es/3090 vector multiprocessors," *IBM Systems Journal*, vol. 28, no. 2, pp. 345–350, 1989.