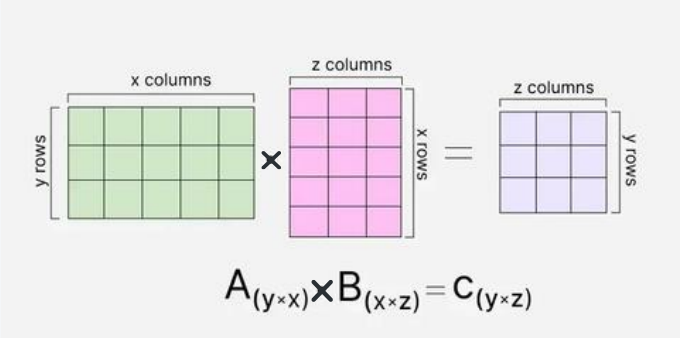
ICPE '26 · Florence, Italy · May 04–08, 2026

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

Vyuhita Bonthu · Nikhil Hegde
Indian Institute of Technology Dharwad, India

Why Matrix-Multiplication?

Matrix-multiplication is a core operation in modern computing — from ML training to HPC benchmarks.



- ***Machine Learning***
- ***Computer Graphics***
- ***Social Networks***
- ***Scientific Computing***
- ***HPC Benchmarking***
- ***Data Analytics***

What is the Problem we are trying to Solve?

Matrix-multiplication is a core operation in modern computing — from ML training to HPC benchmarks.

No single framework delivers optimized MatMul across all Hardwares & exploiting multiple levels of Parallelism

What is the Problem we are trying to Solve?

Matrix-multiplication is a core operation in modern computing — from ML training to HPC benchmarks.

No single framework delivers optimized MatMul across all Hardwares & exploiting multiple levels of Parallelism

So, you need an automated approach

What is the Problem we are trying to Solve?

Matrix-multiplication is a core operation in modern computing — from ML training to HPC benchmarks.

No single framework delivers optimized MatMul across all Hardwares & exploiting multiple levels of Parallelism.

So, you need an automated approach

Energy efficiency
Peak Temperature
Deployment Cost**(data centers, AI workloads, edge devices)**

What is the Problem we are trying to Solve?

Matrix-multiplication is a core operation in modern computing — from ML training to HPC benchmarks.

No single framework delivers optimized MatMul across all Hardwares & exploiting multiple levels of Parallelism.

So, you need an automated approach

Energy efficiency
Thermal behavior
Deployment Cost**(data centers, AI workloads, edge devices)**

Like Nvidia said "Performance per watt, not the price per chip, determines revenue" because power, not money, is the ultimate limitation in data-center growth.

What's Missing in Existing Approaches?

Library-Based

e.g. cuBLAS, MKL, OpenBLAS

Fast but non-portable across platforms

Compiler / DSL

e.g. Exo, Exo2, Halide

Python-like, easy to write but do not scale well

Runtime Systems

e.g. libflame, Kokkos

Flexible but high overhead (~2.3× slower) & limited heterogeneity

Hand-Tuned Code

Fastest but takes weeks of expert effort per platform

Key Contributions

- We create a framework that automates the generation of optimized Matrix-Multiplication codes.

Key Contributions

- We create a framework that automates the generation of optimized Matrix-Multiplication codes.
- We support execution on CPUs, GPUs & Hybrid systems.

Key Contributions

- We create a framework that automates the generation of optimized Matrix-Multiplication codes.
- We support execution on CPUs, GPUs & Hybrid systems.
- We Compare our generated kernels against the SOTA Cuda [2] kernels & Frameworks like Exo [3].

We analyse the approaches on three fronts…

PERFORMANCE & PORTABILITY

How fast?

Parallelism:

Within a core (SIMD/vectors)

Across cores (multi-thread)

Multi-node / multi-GPU scaling

EFFICIENCY

At What cost?

Power draw

Peak temperature

Cost-of-ownership (pJ/bit metric)

PRODUCTIVITY

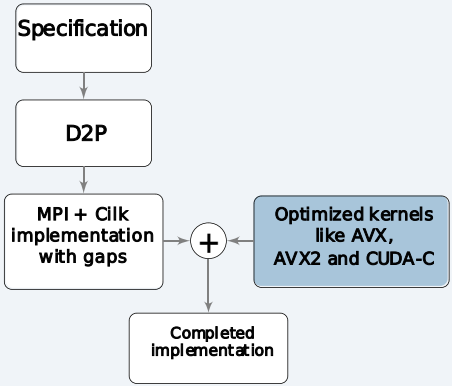
How quickly?

Lines of code vs. performance

Cross-platform portability

Ease of development

How do we produce code?



System Overview

We use **D2P** [1], a tool to generate the code to automate the creation of parallel codes for both shared- and distributed-memory architectures based on recursive algorithm specification.

How do we produce code?

a) Mathematical Formulation:

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

b) D2P Specification:

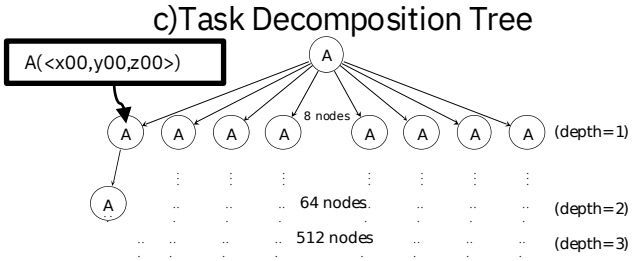
A(<x,y,z>)

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>)

- The method 'A' represents the multiply-add operation.
- 'A' is a recursive method that computes the product of 'y' & 'z' and then adds the result to 'x'.
- Symbols <, >, blankspace & comma can be ignored.
- The Parallel keyword is an annotation indicating that all the method calls on the same line may be executed in parallel.

How do we produce code?



Matrix size $\in 2^n \in \mathbb{N}$.

Implementation Variants

30+ flavours across CPU, GPU, and Edge — single- & double-precision sub-variants

CPU · SIMD Vector	
d2p_AVX512 / hw_AVX512	x86 512-bit
d2p_NEON / hw_NEON	ARM NEON
d2p_AVX256 / hw_AVX256	x86 256-bit

Implementation Variants

30+ flavours across CPU, GPU, and Edge — single- & double-precision sub-variants

CPU · SIMD Vector

d2p_AVX512 / hw_AVX512	x86 512-bit
d2p_NEON / hw_NEON	ARM NEON
d2p_AVX256 / hw_AVX256	x86 256-bit

CPU · Parallel Runtime

hw_CILK_AVX	— OpenCilk + AVX
hw_OMP_AVX	— OpenMP + AVX
hw_Rec_Blocked	— Recursive blocked

Implementation Variants

30+ flavours across CPU, GPU, and Edge — single- & double-precision sub-variants

CPU · SIMD Vector

d2p_AVX512 / hw_AVX512 x86 512-bit
d2p_NEON / hw_NEON ARM NEON
d2p_AVX256 / hw_AVX256 x86 256-bit

CPU · Parallel Runtime

hw_CILK_AVX — OpenCilk + AVX
hw_OMP_AVX — OpenMP + AVX
hw_Rec_Blocked — Recursive blocked

CPU · Iterative

ijk, kij, ikj, jik — loop-orders

Implementation Variants

30+ flavours across CPU, GPU, and Edge — single- & double-precision sub-variants

CPU · SIMD Vector

d2p_AVX512 / hw_AVX512 x86 512-bit
d2p_NEON / hw_NEON ARM NEON
d2p_AVX256 / hw_AVX256 x86 256-bit

CPU · DSL / Compiler

Exo SOTA — Python-like DSL

CPU · Parallel Runtime

hw_CILK_AVX — OpenCilk + AVX
hw_OMP_AVX — OpenMP + AVX
hw_Rec_Blocked — Recursive blocked

CPU · Iterative

ijk, kij, ikj, jik — loop-orders

Implementation Variants

30+ flavours across CPU, GPU, and Edge — single- & double-precision sub-variants

CPU · SIMD Vector

d2p_AVX512 / hw_AVX512 x86 512-bit
d2p_NEON / hw_NEON ARM NEON
d2p_AVX256 / hw_AVX256 x86 256-bit

CPU · Parallel Runtime

hw_CILK_AVX — OpenCilk + AVX
hw_OMP_AVX — OpenMP + AVX
hw_Rec_Blocked — Recursive blocked

CPU · Iterative

ijk, kij, ikj, jik — loop-orders

CPU · DSL / Compiler

Exo SOTA — Python-like DSL

CPU · BLAS Library

d2p_OpenBLAS / hw_OpenBLAS
d2p_MKL / hw_MKL — Intel MKL

Implementation Variants

30+ flavours across CPU, GPU, and Edge — single- & double-precision sub-variants

CPU · SIMD Vector

d2p_AVX512 / hw_AVX512 x86 512-bit
d2p_NEON / hw_NEON ARM NEON
d2p_AVX256 / hw_AVX256 x86 256-bit

CPU · Parallel Runtime

hw_CILK_AVX — OpenCilk + AVX
hw_OMP_AVX — OpenMP + AVX
hw_Rec_Blocked — Recursive blocked

CPU · Iterative

ijk, kij, ikj, jik — loop-orders

CPU · DSL / Compiler

Exo SOTA — Python-like DSL

CPU · BLAS Library

d2p_OpenBLAS / hw_OpenBLAS
d2p_MKL / hw_MKL — Intel MKL

GPU · Single

d2p_CUDA / ref_CUDA
d2p_cuBLAS / ref_cuBLAS

Implementation Variants

30+ flavours across CPU, GPU, and Edge — single- & double-precision sub-variants

CPU · SIMD Vector

d2p_AVX512 / hw_AVX512 x86 512-bit
d2p_NEON / hw_NEON ARM NEON
d2p_AVX256 / hw_AVX256 x86 256-bit

CPU · Parallel Runtime

hw_CILK_AVX — OpenCilk + AVX
hw_OMP_AVX — OpenMP + AVX
hw_Rec_Blocked — Recursive blocked

CPU · Iterative

ijk, kij, ikj, jik — loop-orders

CPU · DSL / Compiler

Exo SOTA — Python-like DSL

CPU · BLAS Library

d2p_OpenBLAS / hw_OpenBLAS
d2p_MKL / hw_MKL — Intel MKL

GPU · Single

d2p_CUDA / ref_CUDA
d2p_cuBLAS / ref_cuBLAS

GPU · Multi

d2p_MGPU / ref_MGPU -- multiple GPUs
ref_cuBLAS_MGPU — Using cuBLAS

Implementation Variants

30+ flavours across CPU, GPU, and Edge — single- & double-precision sub-variants

CPU · SIMD Vector

d2p_AVX512 / hw_AVX512 x86 512-bit
d2p_NEON / hw_NEON ARM NEON
d2p_AVX256 / hw_AVX256 x86 256-bit

CPU · Parallel Runtime

hw_CILK_AVX — OpenCilk + AVX
hw_OMP_AVX — OpenMP + AVX
hw_Rec_Blocked — Recursive blocked

CPU · Iterative

ijk, kij, ikj, jik — loop-orders

CPU · DSL / Compiler

Exo SOTA — Python-like DSL

CPU · BLAS Library

d2p_OpenBLAS / hw_OpenBLAS
d2p_MKL / hw_MKL — Intel MKL

GPU · Single

d2p_CUDA / ref_CUDA
d2p_cuBLAS / ref_cuBLAS

GPU · Multi

d2p_MGPU / ref_MGPU -- multiple GPUs
ref_cuBLAS_MGPU — Using cuBLAS

GPU · Edge / Embedded

Jetson_Nano — Iterative CUDA, 128-core, 10 W

Snippets of Hybrid Parallel

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

Optimized terminating case with AVX.

```
      // Annotated pseudocode for illustration
      // 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
}
```

Handwritten SIMD codes for x86 and Multi-GPU platforms combined with D2P [3] produced code to create “hybrid code.”

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

Optimized terminating case with CUDA multi-GPU.

```
    // Annotated pseudocode for illustration
    //not meant for execution
   // Multi-GPU Wrapper
   A  call to function A_wrapper(xData, yData,
   zData, Y, Z, S, GPUs):
     for i in 0..GPUs-1:
       set GPU i, create stream
       alloc d_A[i], d_B[i], d_C[i]
      // copy data to each GPU
       if i == 0:
         copy yData to d_B[i], zData to d_C[i]
       else:
         peer-copy d_B[0], d_C[0] to d_B[i], d_C[i]
     split S across GPUs
     for i in 0..GPUs-1:
       copy xData slice to d_A[i]
       launch A_kernel on GPU i
     for i in 0..GPUs-1:
       copy d_A[i] to xData slice
       free GPU memory and stream
```

```
// 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
}
```

How do we evaluate?

- Execution time
- Energy consumption
- Peak temperature
- Cost of Ownership (Novel metric)

Cost of Ownership

Measures **Energy Efficiency** independent of hardware cost or region.

Formula:

$$CO = \left(\frac{E}{N^3 \times p} \right) \times 10^{12}$$

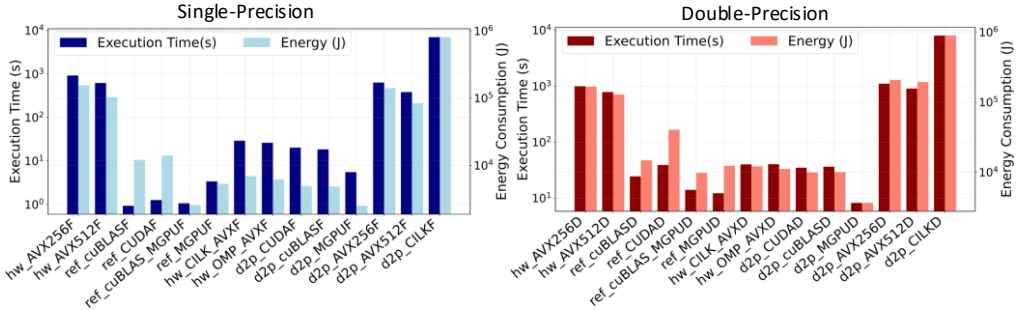
- E: Total energy consumed (Joules)
- N: Matrix dimension
- p: Data precision (bits)

Lower CO (picoJoules/bit) signifies **higher energy efficiency**.

System Configuration

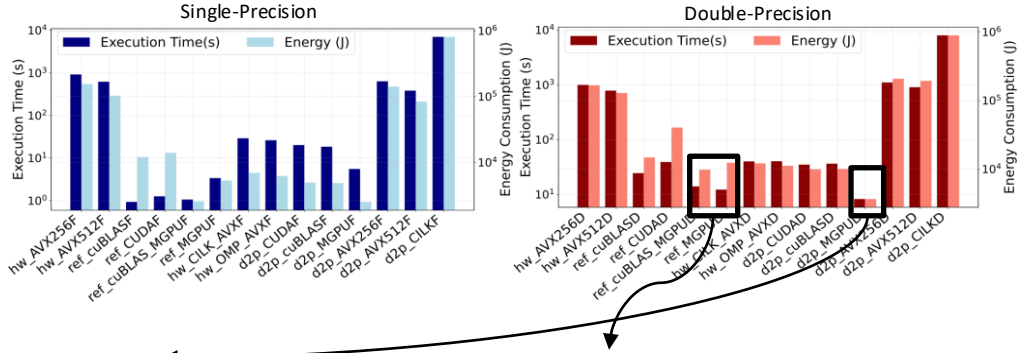
Platform	Configuration
RTX Server	36 cores(2-socket) Intel Xeon Gold 6240C, 2.2MiB L1, 36MiB L2, 49.5MiB L3 cache, 2.60 GHz base, 150 W TDP; 2× NVIDIA Quadro RTX5000 GPUs (16 GB, 3,072 CUDA cores, 230 W).
DGX Server	8× NVIDIA V100 GPUs, 32 GB memory each; used for multi-GPU experiments.
GPU Cluster	Two 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.

Performance & Energy Comparison on RTX Server



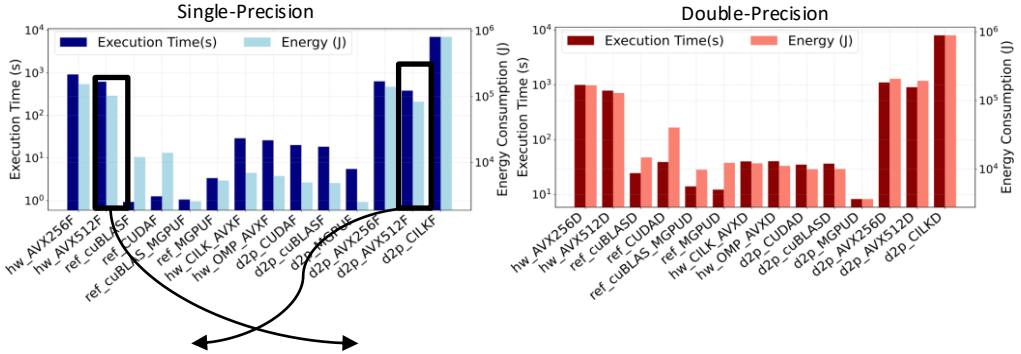
Cross-platform, Cross-framework energy and Execution-time measurements

Performance & Energy Comparison on RTX Server



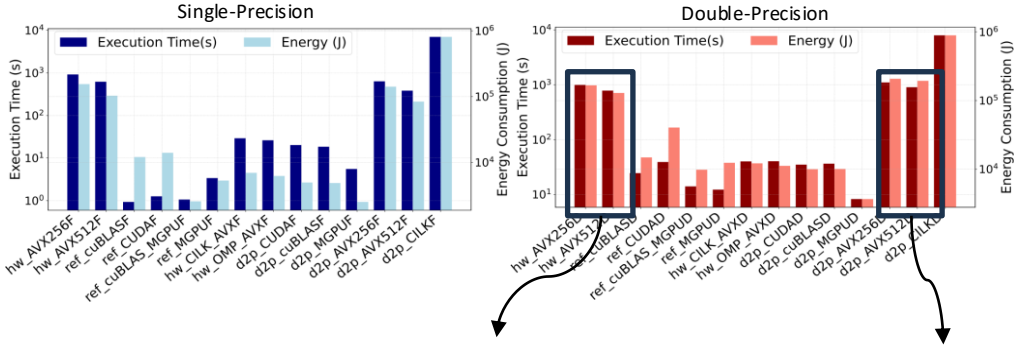
(**d2p_MGPUD**) outperforms (**ref_MGPUD & ref_cuBLAS_MGPUD**) in speed and energy.

Performance & Energy Comparison on RTX Server



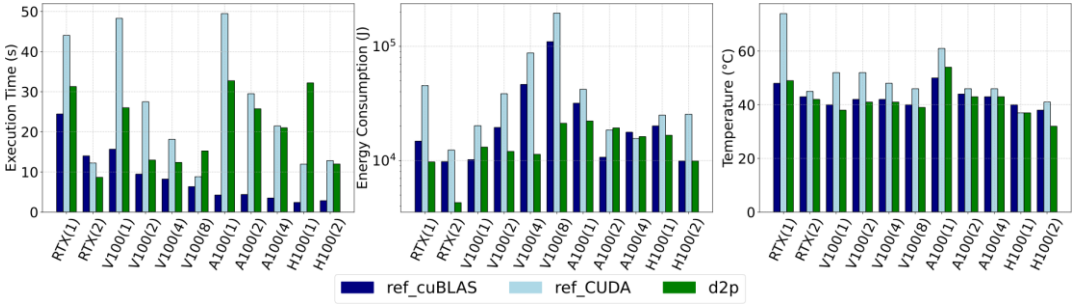
d2p_AVX512F beats **hw_AVX512F** in time and energy.

Performance & Energy Comparison on RTX Server



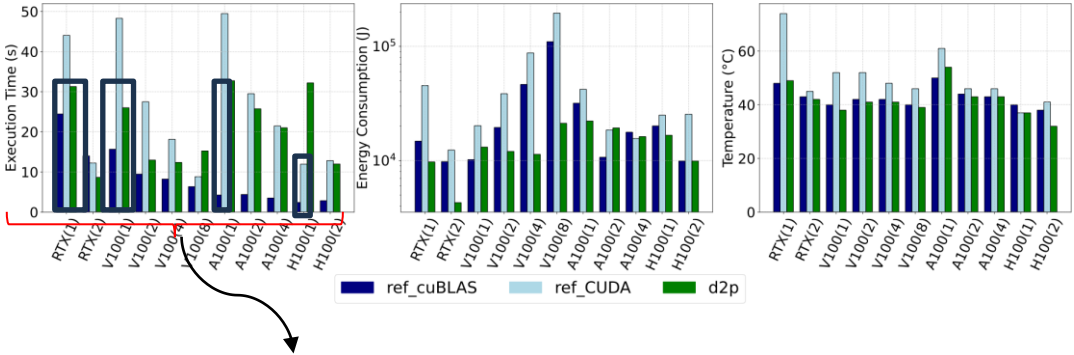
Hand-written (**hw_AVX256D, hw_AVX512D**) doubles outperform **d2p_AVX256D, d2p_AVX512D** in time and energy.

Energy Comparisons across a range of GPU Servers



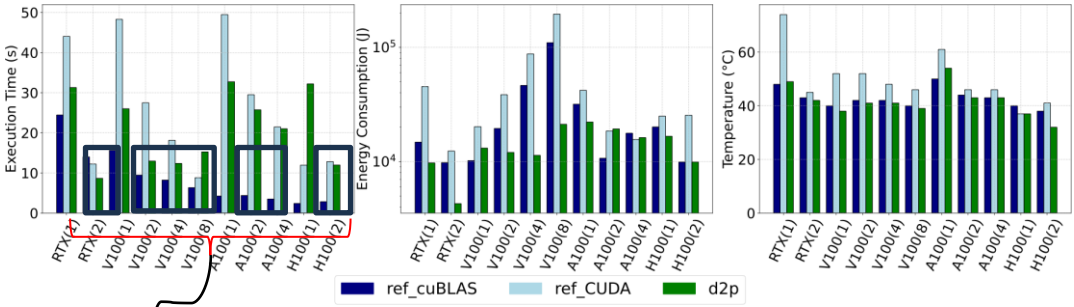
Comparison of Execution time, Energy and Temperature

Energy Comparisons across a range of GPU Servers



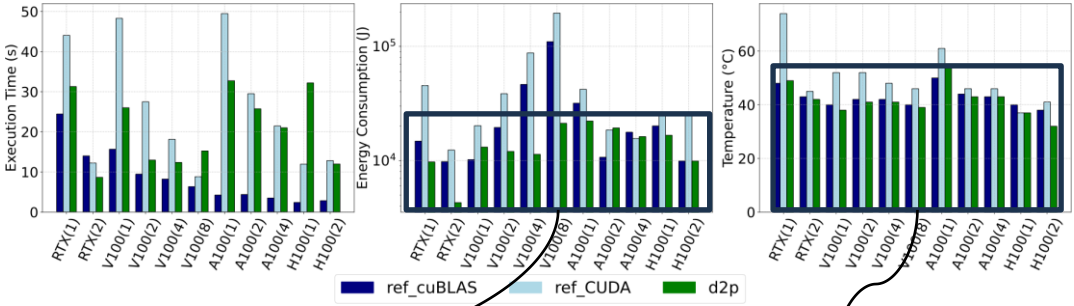
ref_cuBLAS outperforms in single-GPU across all GPUs.

Energy Comparisons across a range of GPU Servers



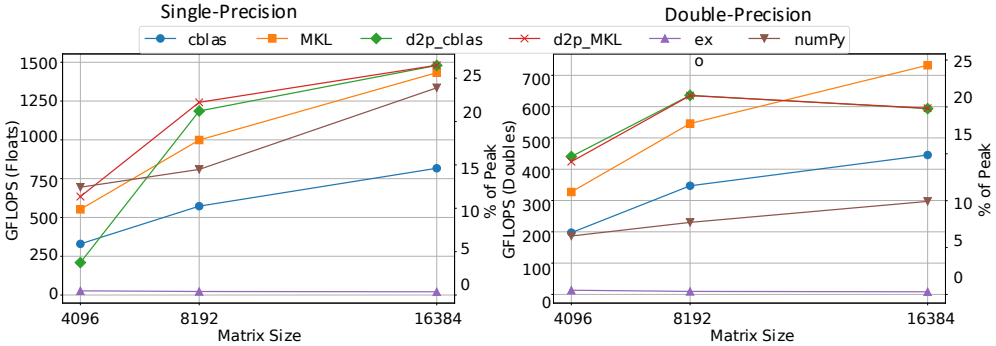
d2p_multi performs well on RTX5000 in multi-GPU while **ref_cuBLAS_multi** outperforms on V100, A100, H100

Energy Comparisons across a range of GPU Servers



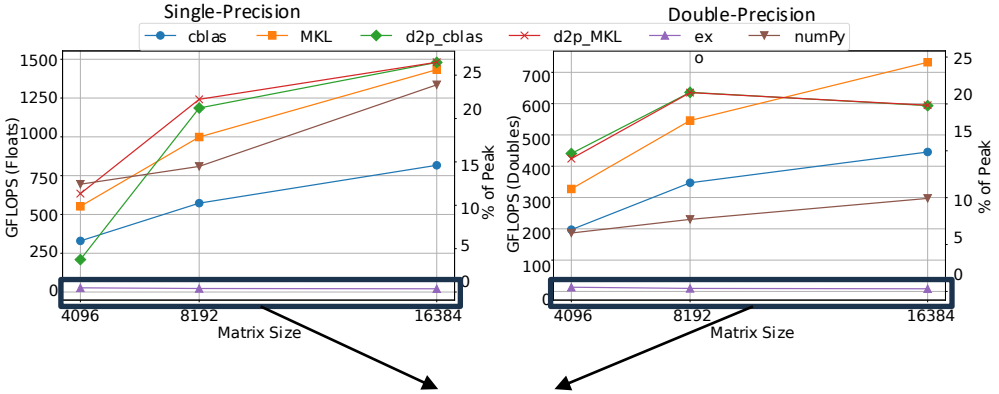
Hybrid codes consume less energy and lower **peak temperatures** than references across GPU systems

Hybrid & DSL based Comparison

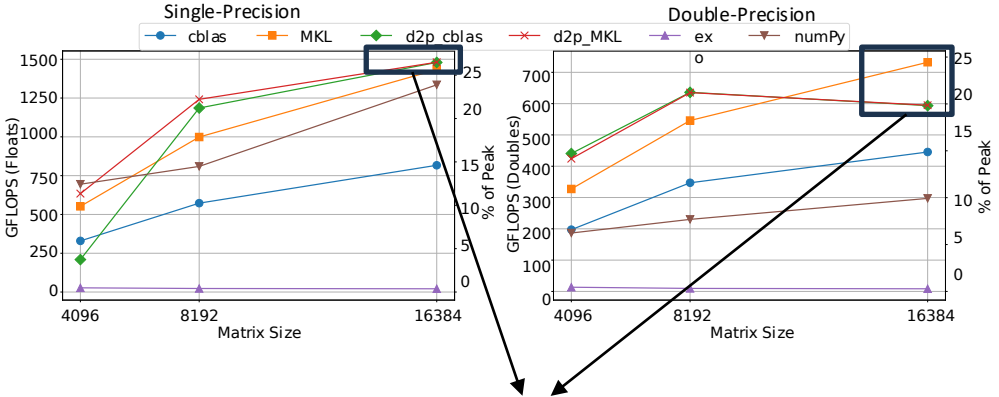


Throughput (measured as a % of peak) comparison

Hybrid & DSL based Comparison

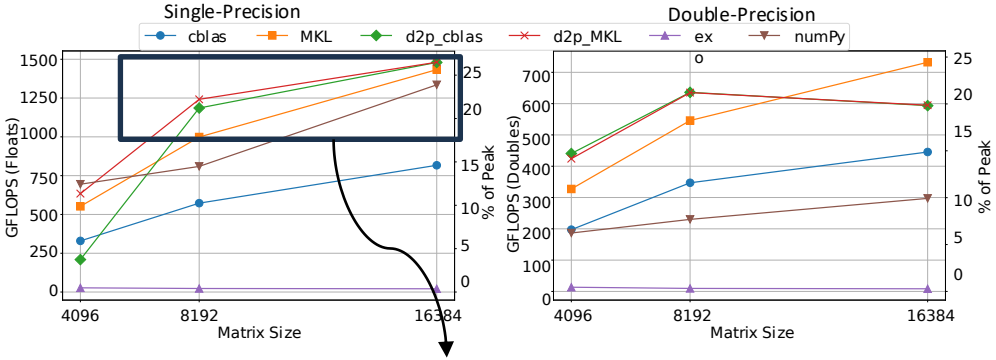


Hybrid & DSL based Comparison



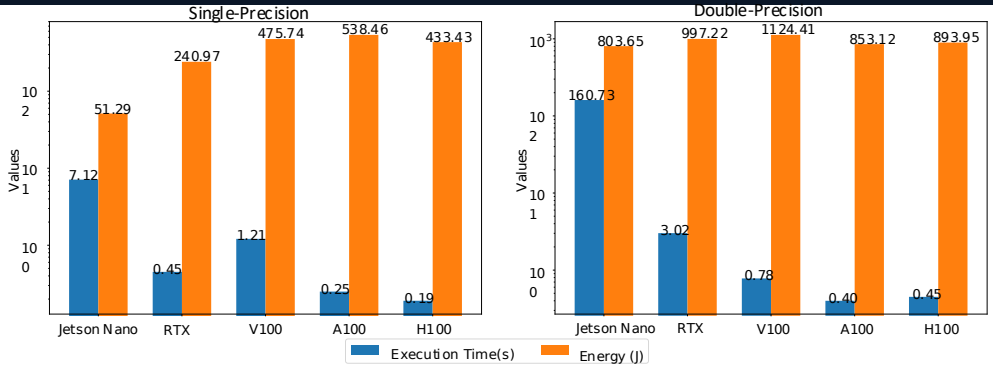
Hybrid D2P scales well, outperforming Exo at large sizes.

Hybrid & DSL based Comparison



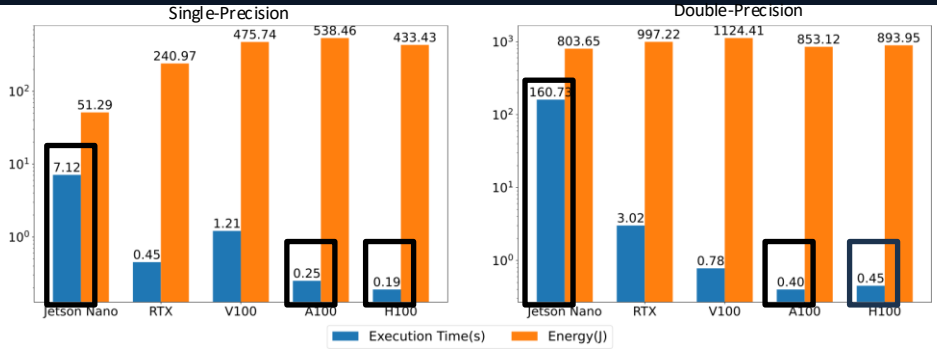
d2p_MKL/CBLAS beat handwritten CPU codes in single precision

Energy Comparisons across Different Hardwares (including Resource constraints)



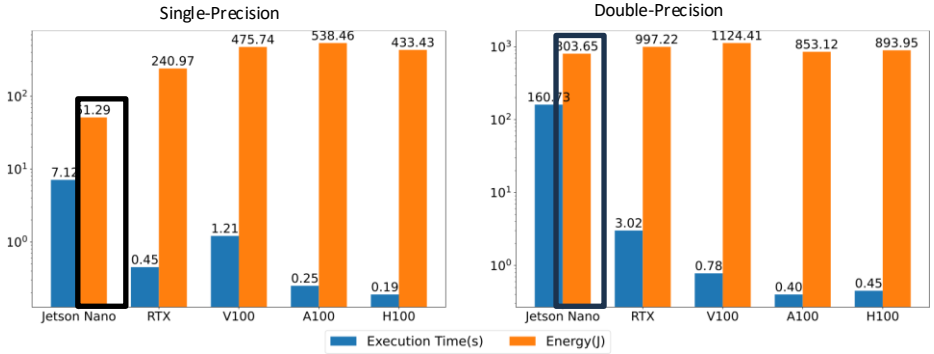
Execution time and Energy consumption of reference CUDA codes

Energy Comparisons across Different Hardwares (including Resource constraints)



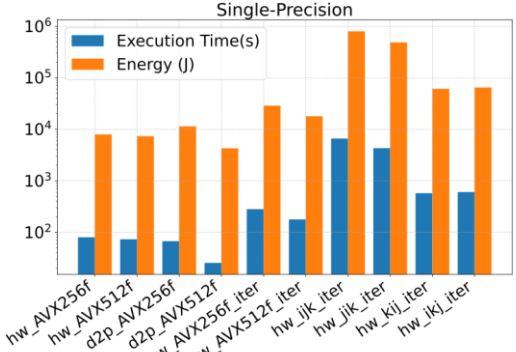
Jetson Nano is 37× slower than H100 single precision & 133× slower than A100 double precision.

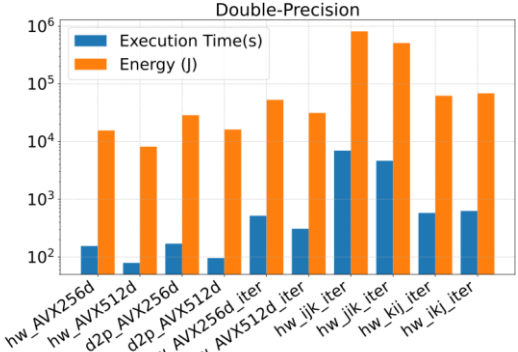
Energy Comparisons across Different Hardwares (including Resource constraints)



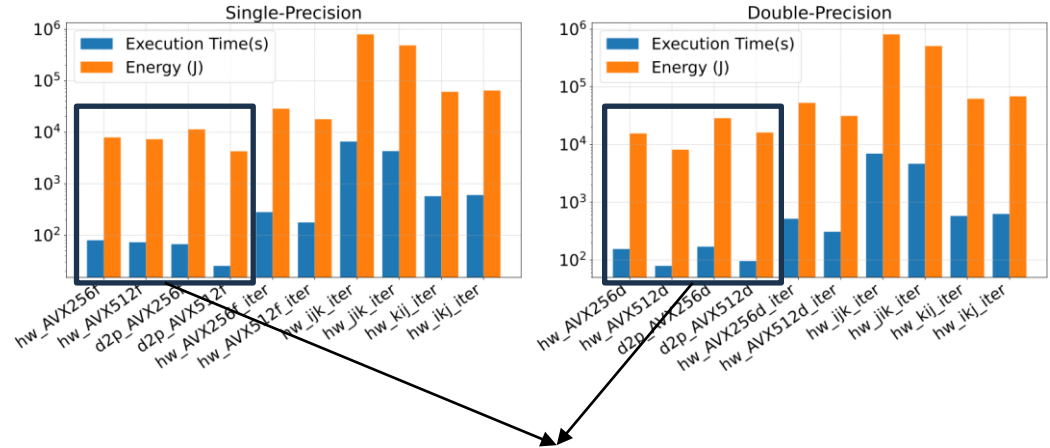
Jetson Nano consumes 8.5× less than H100 single precision & 1.7× more energy efficient than A100 in double precision.

Energy Measurements on CPU Codes



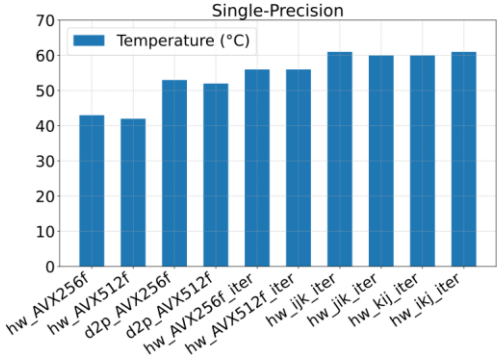


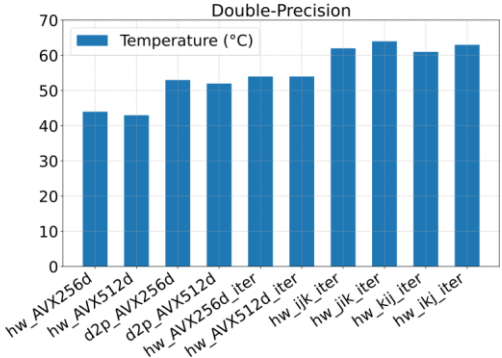
Energy Measurements on CPU Codes



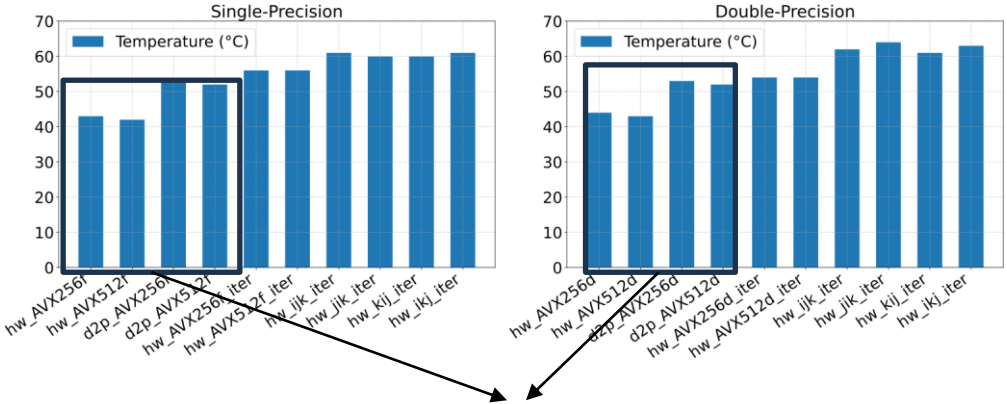
Recursive codes run faster and use less energy than iterative ones.

Peak Temperature measurements on CPU Codes



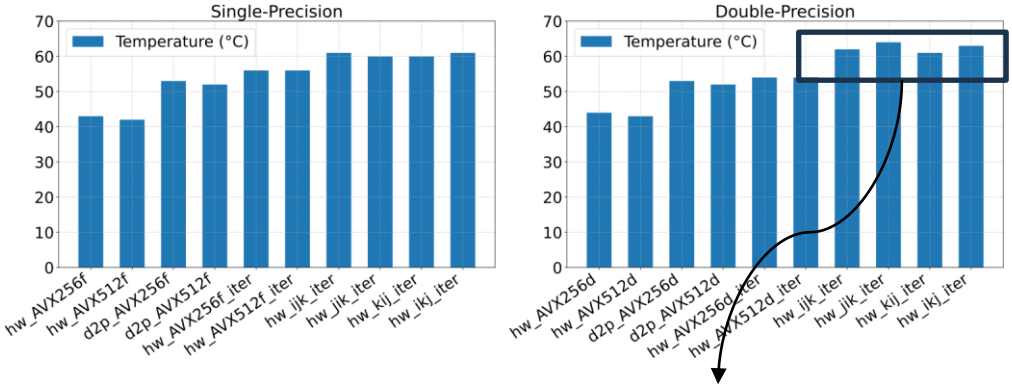


Peak Temperature measurements on CPU Codes



Recursive implementations have lower peak temperatures than iterative.

Peak Temperature measurements on CPU Codes



Double precision causes higher temperatures than single precision.

Cost-of-Ownership (CO)

Implementation	CO (pJ/bit)
Jetson Nano	2.915
d2p_MGPU	11.681
d2p_AVX	30.356
d2p_CUDA	34.598
d2p_cuBLAS	35.521
OMP+AVX	39.224
CILK+AVX	42.020
hw_MGPU	43.929
hw_AVX	52.066
hw_cuBLAS	52.610
hw_CUDA	161.166
exo	2392.838

Findings:

Jetson Nano and hybrid/multi-GPU setups have the lowest CO values.

Cost-of-Ownership (CO)

Implementation	CO (pJ/bit)
Jetson Nano	2.915
d2p_MGPU	11.681
d2p_AVX	30.356
d2p_CUDA	34.598
d2p_cuBLAS	35.521
OMP+AVX	39.224
CILK+AVX	42.020
hw_MGPU	43.929
hw_AVX	52.066
hw_cuBLAS	52.610
hw_CUDA	161.166
exo	2392.838

Findings:

Optimized hybrid codes (d2p_MGPU, d2p_AVX, etc.) outperform traditional hand-tuned implementations.

Conclusion

- Simplified development of optimized matrix multiplication
- Built a hybrid-parallel approach:

  Works across CPUs, GPUs, and multiple systems

Future Scope

- Extend to other linear algebra operations
- Support mixed- and half-precision computation for improved energy efficiency and higher throughput in datacenter and AI workloads
- Support more hardware:
  - TPUs, FPGAs

Key References

[1] D2p: From recursive formulations to distributed-memory codes.In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–22, 2019.

[2] Exo-compilation for productive programming of hardware accelerators. In Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, pages 703–718, 2022.

[3] Sgemm-cuda - high-performance matrix multiplication in cuda.2023. Accessed: 2025-02-19.

Thank You!