

Fast GPU Linear Algebra via Compile Time Expression Fusion

Ryan R. Curtin¹, Marcus Edel², and Conrad Sanderson^{3,4}

¹ Independent Researcher, Atlanta, USA
ryan@ratml.org

² Collabora, Montreal, Canada

³ CSIRO, Brisbane, Australia

⁴ Griffith University, Brisbane, Australia

Abstract We describe the Bandicoot GPU linear algebra toolkit for C++, which prioritises ease of use without compromising efficiency. Bandicoot’s API aims for compatibility with the popular Armadillo CPU linear algebra library, enabling easy transition for existing CPU-based codebases. Unlike other GPU-focused toolkits, Bandicoot uses template metaprogramming to generate fused GPU kernels directly at *compile-time*, yielding efficient kernels that can saturate memory bandwidth. This removes the need for run-time overhead or JIT infrastructure. Empirical results show that Bandicoot outperforms (sometimes by considerable margins) commonly-used linear algebra toolkits including PyTorch, TensorFlow, and JAX.

1 Introduction

As workload sizes increase, modern scientific computing and linear algebra have increasingly come to depend on the use of graphics accelerators / graphics processing units (GPUs) to accelerate computation. The use of GPUs is widespread, most notably in the field of machine learning [20] where the use of GPUs for inference in large language models is all but required [19], though GPUs are also used in a wide variety of other fields, including astrophysics [4], computational fluid dynamics [15], weather simulations [14] and healthcare [21].

GPUs are built on the idea of *single-instruction-multiple-threads* (SIMT) and thus are particularly well-suited for linear algebra tasks, where identical or similar computations are often performed over many matrix elements. For instance, embarrassingly parallel linear algebra operations like the element-wise Schur product $Z = X \odot Y$ can be performed trivially, with one thread assigned to compute each element of Z . In situations like this, GPUs can produce speedups of over 100x compared to a CPU-based implementation.

For a practitioner, there are many options for implementing GPU-accelerated linear algebra or adapting existing code to run on GPUs (Fig. 1). At the low level, vendor-specific toolkits such as CUDA or

<pre>// In C++ using cuBLAS float a = 2.0f, b = 0.0f; cublasSgemm(h, CUBLAS_OP_N, CUBLAS_OP_T, m, n, k, &a, Xptr, m, Yptr, n, &b, Zptr);</pre>	<pre># In Python using JAX Z = 2 * (X @ Y.T) # Expression is optimised at # execution time via JIT compiler</pre>
(a) Too cumbersome: low-level provides speed, but code is unwieldy and error-prone.	(b) Too much overhead: high-level Python is easy to read but requires run-time overhead.
<pre># In Julia using CUDA.jl Z = 2 .* (X * Y')</pre>	<pre>// In C++ using Bandicoot fmat Z = 2 * (X * Y.t());</pre>
(c) Too much overhead: requires the entire Julia execution environment to run.	(d) Just right: Bandicoot optimises at <i>compile-time</i> and requires no run-time overhead.

Figure 1. Implementations of the simple expression $Z \leftarrow 2XY^T$ using various toolkits.

cuBLAS can be used, with each vendor typically supplying its own device-family-specific offering (e.g. HIP/ROCm, OneAPI, etc). However, as vendors repeatedly demonstrate little interest in interoperability with competitors’ products, the use of these tools ties the implementation to a particular vendor’s GPUs. Additionally, most practitioners are not comfortable working at such a low level, which results in significantly increased development effort that often produces cumbersome code.

On the other end of the spectrum are high-level tools that aim to entirely abstract the GPU away, allowing practitioners to focus entirely on their computational tasks. Tools like MATLAB and Julia [5] aim to provide simple APIs so that practitioners can write visually simple code that reflects their mathematical intentions. In the field of machine learning, commonly-used machine learning toolkits like PyTorch [3] and TensorFlow [1] are essentially GPU-based linear algebra engines. Unfortunately, such high-level languages typically require environments that add considerable overhead at run-time, including increased execution time and resource usage. Furthermore, sometimes add-ons like `torch.compile` [3] or JAX [11] are required to analyse linear algebra expressions and select or generate optimised GPU kernels via a just-in-time (JIT) compilation approach.

However, it is possible to obtain high speed and low overhead while still providing an intuitive, high-level API that does not burden practitioners with device-specific or vendor-specific miscellany. Inspired by the widely-used C++ Armadillo CPU linear algebra library [17], which uses template metaprogramming to optimise expressions at *compile-time*, we introduce and describe the Bandicoot GPU linear algebra library which uses template metaprogramming to generate specialised and optimised GPU kernels *entirely* at compile-time. This approach is in contrast to previous C++ based approaches that rely on *run-time* expression analysis and/or elaborate JIT compilation infrastructure [6,13]. Due to the low overhead and the ability to generate highly specialised kernels, expressions written in Bandicoot can outperform other toolkits, sometimes by considerable margins.

Bandicoot aims to facilitate easy adaptation of Armadillo based code, enabling existing CPU-focused computational pipelines to take advantage of GPUs. Bandicoot supports multiple GPU backends, including CUDA and OpenCL, with further backends like Vulkan, HIP/ROCm, and Apple Metal under active development. As the library is not focused simply on numerical linear algebra for machine learning, support is provided for various decompositions and factorisations, such as SVD, eigen, Cholesky, etc.

2 Intuitive and Simple User Interface

Bandicoot provides a high-level interface for linear algebra without sacrificing performance. To maximise the reusability of existing codebases, the Bandicoot API is modelled to match the API of the widely-used Armadillo C++ linear algebra toolkit [17]. Armadillo provides an API that closely approximates the API of MATLAB, which lessens the work involved for a practitioner when making a transition from slow, high-overhead tools like MATLAB or Mathematica to fast C++ implementations. The simplicity of Armadillo’s API, along with its speed [16], is directly responsible for its use in numerous well-known downstream packages, such as the *mlpack* machine learning library [9], the *ensmallen* numerical optimisation library [7], and the popular *RcppArmadillo* project [10] that provides efficient numerical linear algebra to the R language.

Using Bandicoot does not require deep GPU knowledge; explicit setup or management of the device is not necessary unless specifically desired. An example program is shown in Figure 2. All operations are dispatched to the GPU asynchronously, allowing the GPU to compute the results while the main thread of the C++ program continues. GPU synchronisation is only performed when necessary (e.g. within the `x.print()` function), or when explicitly specified via the `coot_synchronise()` function.

In general, it is possible to take an existing Armadillo program and replace uses of Armadillo matrices and operations (e.g. `arma::fmat`) directly with Bandicoot matrices and operations (e.g. `coot::fmat`), resulting in the computations running on the GPU instead of the CPU. Although not all Armadillo operations are yet implemented in Bandicoot, work is ongoing for that, including porting factorisations from MAGMA [2].

However, not all Armadillo code will immediately run efficiently on a GPU. This is due to the different nature of GPU computations. For example, individual element access (e.g. `x(r,c) = val`) on a CPU has very low overhead. In contrast, accessing an element of a Bandicoot matrix incurs a high-latency copy from GPU memory to CPU memory. Therefore, Bandicoot code is generally much more efficient when

written as a series of matrix-level operations instead of loops over individual elements. This is similar to the recommendations made by other GPU focused toolkits.

3 Internal Design and Expression Optimisation

As Bandicoot is intended to be vendor-independent, it has been designed so that low-level device implementation is fully separated from the user interface code and code related to optimising linear algebra expressions. Figure 3 outlines the internal design.

```
#include <bandicoot>
using namespace coot;

int main()
{
    fvec A(1000, fill::randu); // column vectors with 32-bit
    fvec B(1000, fill::randu); // floating point elements
    B += log(3 * (A + 3)); // operation fused into a single kernel

    fmat X(1000, 1000, fill::randu); // matrices with 32-bit
    fmat Y(1000, 1000, fill::randu); // floating-point elements

    fmat Z = X.t() * Y; // combined transpose and matrix multiplication
    Z %= pow(2 * X + Y.t(), 2); // operation fused into a single kernel

    fmat L, U, P; lu(L, U, P, Z); // LU decomposition
    fvec C = solve(Z, A); // solve system of linear equations

    X.print("X:"); // pretty print matrix to std::cout stream
}
```

Figure 2. An example C++ program using Bandicoot to execute linear algebra operations on a GPU. The corresponding Armadillo-based program, which uses the CPU instead of the GPU, can be obtained by replacing the code on line 1 with `#include <armadillo>` and on line 2 with `using namespace arma;`.

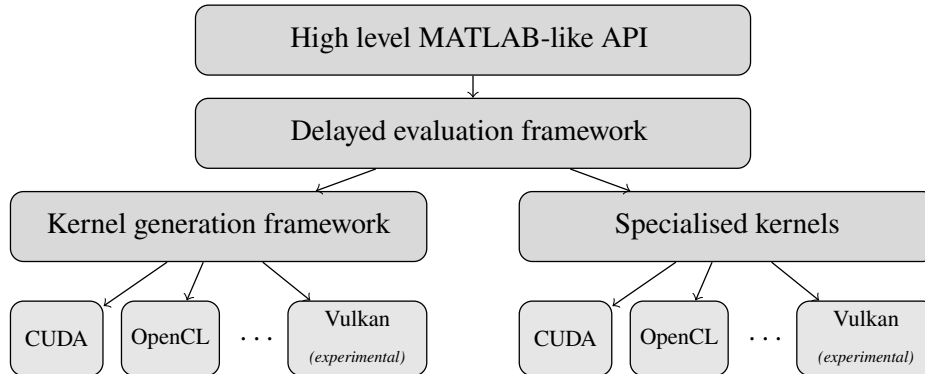


Figure 3. Internal architecture and design of Bandicoot. High-level user code is converted to efficient, fused GPU kernels at program compilation time, not at run-time. Vulkan backend support is currently experimental; more backends (e.g. HIP/ROCm) are planned.

```

__global__ void (COOT_KERNEL_FUNC)(COOT_OBJECT_0(out), COOT_OBJECT_1(in))
{
    const UWORD row = blockIdx.x * blockDim.x + threadIdx.x;
    const UWORD col = blockIdx.y * blockDim.y + threadIdx.y;
    if (COOT_OBJECT_0_BOUNDS_CHECK(out, row, col))
        COOT_OBJECT_0_AT(out, row, col) = COOT_OBJECT_1_AT(in, row, col);
}

```

Figure 4. A simplified version of Bandicoot’s copy skeleton kernel for the CUDA backend. The definition of each `COOT_()` macro is generated at compile-time.

Underneath the user-facing functions (e.g. `operator+()`, `accu()`, `trans()`, etc) is a template metaprogramming infrastructure that provides significant compile-time optimisations of mathematical expressions. This is accomplished by automatically collecting (at compile-time) the structure of a linear algebra expression as an elaborate custom type, and then using techniques such as template specialisation [22] to generate efficient code to evaluate the expression.

As an example, the expression `x.t() + 3` is represented by the C++ type `eOp<Op<fmat, op_htrans>, op_scalar_add>`, where `op_htrans` and `op_scalar_add` are placeholder types that represent the operations to be performed. In essence, the C++ type of an expression is the Abstract Syntax Tree (AST) [12] of that expression, and the use of template specialisations allows the compiler to optimise that AST. Furthermore, the entire expression scaffolding is optimised out of the program at compile-time, and thus no traces of it remain in the compiled code. More details on the infrastructure can be found in accompanying literature [8,17,18].

Unique to Bandicoot among linear algebra toolkits is its compile-time generation of fused GPU kernel sources, supported by all available GPU backends. Every GPU backend supports on-the-fly compilation of GPU kernel source, so that user code can freely be distributed between systems with various GPUs. Most linear algebra toolkits make use of this, and ship with either a set of predefined kernel sources or infrastructure to build these kernels when they are needed. For specialised operations such as matrix decompositions and pseudo-random number generation, Bandicoot has predefined kernel sources that are compiled in exactly this way.

However, because Bandicoot has the AST for each expression available at compile-time, it can generate fused GPU kernels directly at compile-time. These generated GPU kernel sources can then be embedded directly into the compiled program. This reduces the size of the compiled program, as only those kernels that are needed are included, and no run-time infrastructure for generating kernels is necessary.

The approach relies on the use of *skeleton kernels* and C-like macros, which are supported in some form by every GPU vendor’s low-level toolkits. Each skeleton kernel is a deliberately straightforward recipe for a kernel that uses placeholders to represent its inputs and outputs. Figure 4 shows a simplified example of Bandicoot’s copy skeleton kernel, which copies an input Bandicoot expression to an output.

When a user writes a linear algebra expression, Bandicoot constructs the AST of that expression as a compile-time type. Each macro can be constructed by recursing on the AST and applying individual rules to nodes of the AST via partial template specialisation. In the following text we briefly demonstrate this process applied to the example expression `x.t() + 3`, which yields the type `eOp<Op<fmat, op_htrans>, op_scalar_add>`.

Let us consider the element access macro in the skeleton copy kernel: `COOT_OBJECT_1_AT(out, row, col)`. For the simple base case type of `fmat` (a floating-point matrix), we can use the following trivial definition, where `in_n_rows` represents the number of rows in the matrix:

$$\text{COOT_OBJECT_1_AT}(\text{in}, \text{row}, \text{col}) = (\text{in}[\text{row} + \text{col} * \text{in_n_rows}])$$

At the next level up, the type is `Op<fmat, op_htrans>` (i.e., a transposed matrix). We can generate the element access macro for this as the transposed version of the inner argument of the `Op`’s element access macro, where the `row` and `col` variables are swapped on the right hand side:

```
COOT_OBJECT_1_AT(in, row, col) = (in[col + row * in_n_rows])
```

Finally, the outermost eOp is considered. For this element access macro, we need to add the value of the scalar to the element access macro of the inner argument of the eOp:

```
COOT_OBJECT_1_AT(in, row, col) = (in[col + row * in_n_rows] + aux)
```

where aux is the variable representing the scalar value (in our example, the 3 in $X.t() + 3$).

This general strategy is used to traverse the AST of any expression and generate a list of kernel arguments (represented via `COOT_OBJECT_x()`), bounds checks (`COOT_OBJECT_x_BOUNDS_CHECK()`), element access (`COOT_OBJECT_x_AT()`), a unique generated kernel name (`COOT_KERNEL_FUNC`), and a few other macros that are used by various skeleton kernels.

Importantly, this means that each skeleton kernel can be used for virtually arbitrary expressions. For instance, the copy skeleton kernel is used for almost every Bandicoot expression of the form $Y = \text{<rhs>}$ for some right-hand-side expression. This reuse of skeleton kernels significantly reduces the number of GPU kernels that must be implemented and maintained in Bandicoot, and compares favourably to other GPU toolkits that maintain hand-implemented kernels for a variety of situations.

The AST traversal example given above is a simplified version of the actual implementation in Bandicoot. The exact details of the macro generation framework code can be found in the `include/bandicoot_bits/kernel_gen/` subdirectory, present in Bandicoot 4.x versions. The framework is built around the utility `char_array<N>` struct, which is a compile-time string of length N that supports compile-time concatenation with other compile-time strings.

4 Empirical Speed Comparison

To demonstrate the efficacy of Bandicoot’s template metaprogramming approach, we compare with other toolkits on a variety of simple and complex linear algebra expressions. We use the following expressions, written here using the Bandicoot API, with capitalised variables representing matrices and lowercased variables representing scalars. The `%` operation represents the Schur product (element-wise multiplication), which is distinct from the `*` operation that represents matrix multiplication. For submatrix expressions, we use the center of the matrix with half the rows and columns.

1. Basic operations and submatrices:

```
add2:    Z = X + Y;
add4:    Z = A + B + C + D;
chain:   Z = A * B * C * D;  with decreasing sizes for each matrix.
addsub2: same as add2, but using submatrices of A and B.
addsub4: same as add4, but using submatrices of A, B, C, and D.
```

2. Compound expressions:

```
expr1:   Z = 2 * (X.t() + Y) + 2 * (X + Y.t());
expr2:   Z = a * A + (B + C).t() + log(pow(D, 2));
expr3:   Z = 1 / (X % conv_to<fmat>::from(Y) + log(log(X + 2) % W));
diagsum: (X.diag(-1) + X.diag(1)) % (Y.diag(-1) + Y.diag(1))
```

3. Neural network activations:

```
relu:    Z = X % (X > 0);
sigmoid: Z = 1 / (1 + exp(-X));
swish:   Z = X / (1 + exp(-beta * X));
gelu:    Z = (X / 2) % (1 + tanh(sqrt(2 / pi) * (X + alpha * pow(X, 3))));
```

We ran each of these expressions 50 times, after two ‘warmup’ runs to handle any JIT compilation and/or run-time costs, and computed the mean of each expression’s run-time. In all cases, the standard deviation of the run-times was negligible. Our test setup used an NVIDIA RTX 4090 GPU, with the

CUDA backend used for each toolkit. Except for the chain example, we used square matrices with 10k rows/columns and 32-bit floating point numbers, totalling ~382 MB per matrix. Results are shown in Figure 5.

Bandicoot is the fastest or on par with the fastest toolkit for the vast majority of the expressions. It must be noted that Figure 5 focuses only on the time it took to run a kernel after warmup runs, and as such the run-times do not include extra overheads such as JIT compilation. Bandicoot has no such overhead, beyond the GPU driver’s initial compilation of the generated kernel. Even ArrayFire, which is used directly from C++, suffers from some overhead as it has an internal JIT compiler.

Next, to demonstrate the ability of Bandicoot’s template metaprogramming framework to scale to an arbitrary number of matrices at compile-time, we adapted the add2 example to add an increasing number of matrices and computed the throughput achieved (in GB/s) by each toolkit. This is a memory-bound operation, and thus we can easily compute the peak possible bandwidth. For the RTX 4090, with a 384-bit GDDR6X memory bus clocked at 1313 MHz, this is theoretically 1008 GB/s. Peak speeds are generally unachievable in practice; using NVIDIA’s nvbandwidth tool, we computed a maximum achievable bandwidth of 868 GB/s.

We again used square matrices with 10k rows/columns, averaging the run-time over 50 trials. Figure 6 plots observed throughput as a function of the number of added matrices, ranging from 2 to 16. Only Bandicoot is able to maintain maximum achievable bandwidth across all tested cases. The achievable bandwidth for most other toolkits quickly drops beyond 2 matrices, and beyond 8 matrices it plateaus at about half the maximum bandwidth. While the JIT framework in ArrayFire is able to maintain high bandwidth up to 8 matrices, the throughput severely degrades past that point, eventually falling well below the other toolkits.

5 Concluding Remarks

We have described Bandicoot, a GPU linear algebra library implemented in C++. Its API matches the Armadillo library, and thus existing user code can be easily ported to make use of GPUs. Internally,

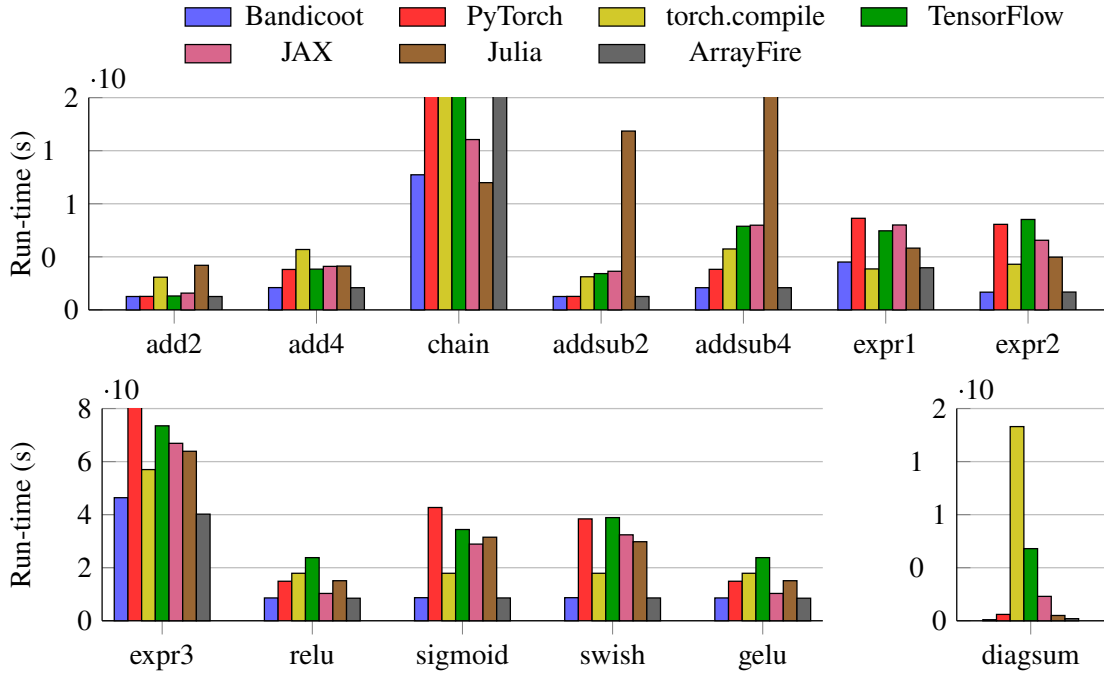


Figure 5. Average runtime (in seconds) for 50 trials of each expression, excluding warm-up costs. Bandicoot is able to fuse operations into a single kernel at compile-time, resulting in execution speed that is the fastest or on par with the fastest toolkit in the vast majority of cases.

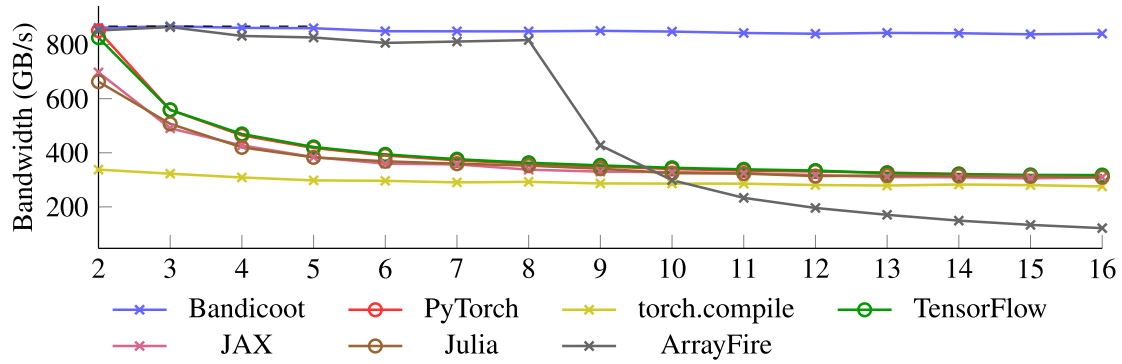


Figure 6. Effective throughput of matrix addition as a function of the number of matrices to be added. Dashed line indicates peak possible memory bandwidth.

Bandicoot uses an elaborate template metaprogramming framework that allows the fusion of operations within a compound mathematical expression into a single GPU kernel, meaning that the expression is computed in a single pass over all given input matrices. This is achieved at compile-time and hence does not require any JIT infrastructure or other overheads at run-time. Comparative evaluations demonstrate competitive performance and the ability of the generated kernels to saturate memory bandwidth.

The software is licensed under the permissive Apache License, version 2.0, and can be found at <https://coot.sourceforge.io>. Future work includes adding more backends, including Apple Metal and AMD HIP/ROCm, support for low-precision floating point types, and eventual implementation of all Armadillo functionality.

Acknowledgements. Ryan Curtin’s contributions are based on work supported by the National Aeronautics and Space Administration (NASA) under the ROSES-23 HPOSS program, grant no. 80NSSC24K1524. Ryan Curtin’s and Marcus Edel’s contributions have also been commissioned and supported by the Sovereign Tech Fund.

References

1. Abadi, M., Barham, P., et al.: TensorFlow: A System for Large-Scale Machine Learning. In: *USENIX Symposium on Operating Systems Design and Implementation*. pp. 265–283 (2016)
2. Abdelfattah, A., Beams, N., Carson, R., et al.: MAGMA: Enabling exascale performance with accelerated BLAS and LAPACK for diverse GPU architectures. *International Journal of High Performance Computing Applications* **38**(5), 468–490 (2024)
3. Ansel, J., Yang, E., He, H., et al.: PyTorch 2: Faster machine learning through dynamic Python bytecode transformation and graph compilation. In: *ACM Int. Conf. Architectural Support for Programming Languages and Operating Systems*. vol. 2, pp. 929–947 (2024)
4. Bard, D., Bellis, M., Allen, M.T., et al.: Cosmological calculations on the GPU. *Astronomy and Computing* **1**, 17–22 (2013)
5. Bezanson, J., Edelman, A., Karpinski, S., Shah, V.B.: Julia: A fresh approach to numerical computing. *SIAM Review* **59**(1), 65–98 (2017)
6. Chen, J., Joo, B., Watson III, W., Edwards, R.: Automatic offloading C++ expression templates to CUDA enabled GPUs. In: *International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. pp. 2359–2368 (2012)
7. Curtin, R.R., Edel, M., Prabhu, R.G., et al.: The ensmallen library for flexible numerical optimization. *Journal of Machine Learning Research* **22**(1), 7552–7557 (2021)
8. Curtin, R.R., Edel, M., Sanderson, C.: Bandicoot: A templated C++ library for GPU linear algebra. *Lecture Notes in Computer Science (LNCS)* **16465** (2026)
9. Curtin, R.R., Edel, M., Shrit, O., et al.: mlpack 4: a fast, header-only C++ machine learning library. *Journal of Open Source Software* **8**(82), 5026 (2023)
10. Eddelbuettel, D., Sanderson, C.: RcppArmadillo: Accelerating R with high-performance C++ linear algebra. *Computational Statistics & Data Analysis* **71**, 1054–1063 (2014)

11. Frostig, R., Johnson, M.J., Leary, C.: Compiling machine learning programs via high-level tracing. In: Conference on Systems and Machine Learning (SysML) (2018)
12. Harper, R.: Practical Foundations for Programming Languages. Cambridge University Press, 2nd edn. (2016)
13. Malcolm, J., Yalamanchili, P., McClanahan, C., et al.: ArrayFire: a GPU acceleration platform. In: Modeling and Simulation for Defense Systems and Applications VII. vol. 8403 (2012)
14. Michalakes, J., Vachharajani, M.: GPU acceleration of numerical weather prediction. In: IEEE International Symposium on Parallel and Distributed Processing. pp. 1–7 (2008)
15. Niemeyer, K.E., Sung, C.J.: Recent progress and challenges in exploiting graphics processors in computational fluid dynamics. *The Journal of Supercomputing* **67**(2), 528–564 (2014)
16. Psarras, C., et al.: The linear algebra mapping problem: Current state of linear algebra languages and libraries. *ACM Transactions on Mathematical Software* **48**(3) (2022)
17. Sanderson, C., Curtin, R.: Armadillo: An efficient framework for numerical linear algebra. In: International Conference on Computer and Automation Engineering. pp. 303–307 (2025)
18. Sanderson, C., Curtin, R.R.: Armadillo: C++ template metaprogramming for compile-time optimization of linear algebra. In: PASC Conference (2017)
19. Song, Y., et al.: PowerInfer: Fast large language model serving with a consumer-grade GPU. In: ACM SIGOPS Symposium on Operating Systems Principles. pp. 590–606 (2024)
20. Sze, V., Chen, Y.H., Yang, T.J., Emer, J.S.: Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE* **105**(12), 2295–2329 (2017)
21. Vamathevan, J., Clark, D., Czodrowski, P., et al.: Applications of machine learning in drug discovery and development. *Nature Reviews Drug Discovery* **18**(6), 463–477 (2019)
22. Vandevor, D., Josuttis, N., Gregor, D.: C++ Templates: The Complete Guide. Addison-Wesley, 2nd edn. (2017)