JACC.shared: Leveraging HPC Metaprogramming and Performance Portability for Computations That Use Shared Memory GPUs

Pedro Valero-Lara
Advanced Computing Systems Research Section
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
valerolarap@ornl.gov

William F. Godoy
Advanced Computing Systems Research Section
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
godoywf@ornl.gov

Keita Teranishi
Advanced Computing Systems Research Section
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
teranishik@ornl.gov

Jeffrey S. Vetter
Advanced Computing Systems Research Section
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
vetter@ornl.gov

***Abstract*—In this work, we present `JACC.shared`, a new feature of Julia for ACCelerators (JACC), which is the performance-portable and metaprogramming model of the just-in-time and LLVM-based Julia language. This new feature allows JACC applications to leverage the high-performance computing (HPC) capabilities of high-bandwidth, on-chip GPU memory. Historically, exploiting high-bandwidth, shared-memory GPUs has not been a priority for high-level programming solutions. `JACC.shared` covers that gap for the first time, thereby providing a high-level, portable, and easy-to-use solution for programmers to exploit this memory and supporting all current major accelerator architectures. Well-known HPC and AI workloads, such as multi/hyperspectral imaging and AI convolutions, have been used to evaluate `JACC.shared` on two exascale GPU architectures hosted by some of the most powerful US Department of Energy supercomputers: Perlmutter (NVIDIA A100) and Frontier (AMD MI250X). The performance evaluation reports speedup of up to $3.5\times$ by adding only one line of code to the base codes, thus providing important accelerators in a simple, portable, and transparent way and elevating the programming productivity and performance-portability capabilities for Julia/JACC HPC, AI, and scientific applications.**

***Index Terms*—Julia, JACC, metaprogramming, performance portability, high-bandwidth on-chip memory**

I. Introduction

Metaprogramming-based solutions allow for generic programming, in which programmers focus on the general structure of an application while target-specific code specialization is handled by different alternative specializations, which are transparent to the programmer. This technique can be used effectively for performance portability. Julia for ACCelerators (JACC) is the first and only performance-portable, and architecture-agnostic metaprogramming model for the just-in-time (JIT) and LLVM-based Julia programming language. JACC is an open-source library that enables programmers to move past the low-level details of vendor- or target-specific programming models and the varying characteristics of the targeted hardware architectures. JACC provides multiple device-specific backends that are implemented to support the most important high-performance computing (HPC) platforms (CPUs, NVIDIA GPUs, AMD GPUs, and Intel GPUs). Although most of the current and major HPC accelerators contain software-managed (programmable), high-bandwidth, on-chip memory, historically, its exploitation has not been a priority for high-level programming models, thereby making it difficult, or even impossible in some cases, for the programmers to leverage this kind of memory. In this work, we present `JACC.shared`, a portable, transparent, and easy-to-use function as part of the JACC model. `JACC.shared` allows programmers to benefit from the use of high-bandwidth, on-chip GPU memory, thereby providing a high-level programming solution that supports all the current major HPC accelerators for the first time. The main contributions of this work are as follows:

1) The design and implementation of a novel performance-portable and transparent (programming productivity) JACC function (`JACC.shared`) allow users to easily exploit high-bandwidth and on-chip GPU memory (shared memory GPUs).
2) `JACC.shared` is supported for the three current HPC vendor accelerators: Intel, AMD, and NVIDIA.
3) Two well-known and characteristic problems for HPC (multi/hyperspectral imaging) and AI (convolutions) were used as test cases.
4) A deep performance analysis was performed for two important hardware accelerators deployed on some of the most powerful supercomputers: Perlmutter (NVIDIA A100) and Frontier (AMD MI250X).

II. Julia for ACCelerators (JACC)

Julia was created to provide a unified programming language, community, and integrated ecosystem (e.g., packaging, testing, software tools, AI) to enhance productivity while providing performance mechanisms that rely on LLVM advancements, addressing the main weaknesses of current HPC programming languages as outlined in a recent community paper [1]. Julia uses the LLVM framework for JIT compilation, enabling the same run-time speed as other compiled languages such as C. Julia is also compatible with any external library implemented in Python, Fortran, and C. Similar to Python, Julia's syntax is simple and efficient, and users interact either by passing source code files as arguments to the `julia` command or optionally via its interactive read-eval-print loop command line to easily add commands, scripts, and packages.

Julia offers several advantages:

- Julia syntax is optimized for mathematics and scientific environments similar to the formulas used by domain-specific experts.
- JIT compilation on top of LLVM enables Julia to outperform other high-level languages (e.g., Python, R, MATLAB) in terms of speed.
- Its native support for AI makes Julia a real asset for HPC-AI integration.
- Julia provides a community and integrated ecosystem motivated by performance and productivity.

The support of Julia for HPC, although not as mature as in other languages, is already significant. The Julia ecosystem supports parallel computation on CPUs by using `Base.Threads`, which is a Julia package that is implemented in `pthreads` on top of LLVM and that enables the distribution of the computation on different CPU cores by using decorators on top of loops (similar to OpenMP and OpenACC). Julia natively supports GPU accelerator programming thanks to vendor packages, such as `CUDA.jl`, `AMDGPU.jl`, and `OneAPI.jl`. Other packages, such as `Distributed.jl` and `MPI.j` [2], allow Julia codes to run on distributed-memory environments.

Julia is not different from other programming languages in facing performance-portability challenges. Currently, Julia's programming models tend to closely follow vendor layers, which could still be too low level, thereby hindering programming productivity. JACC addresses this challenge for Julia programmers and applications, providing an HPC-portable and highly productive model targeting current HPC (CPU and GPU) hardware, which could potentially be extended to other architectures (e.g., AI custom hardware, field-programmable gate arrays) and configurations (e.g., distributed memory, multidevice use).

The JACC model (Fig. 1) is divided into two main components: memory and compute. These components have different implementations, and one per backend is supported. We implemented four backends so far on top of `Base.Threads`, CUDA, AMDGPU, and OneAPI to target CPUs, NVIDIA GPUs, AMD GPUs, and Intel GPUs, respectively.

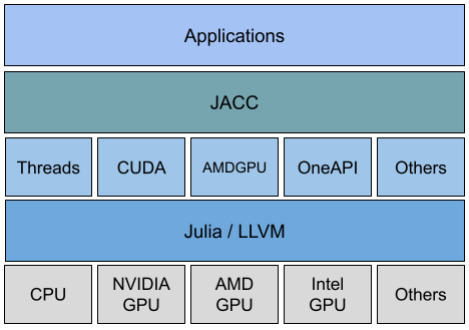

Fig. 1. JACC model illustrating its interactive and lightweight nature on top of LLVM for performance-portable code.

Owing to the dynamic and JIT nature of the Julia language, JACC differs from other existing metaprogramming solutions in how the backend is chosen [3], [4]. We use Julia's `Preferences` package, which generates the `LocalPreferences.toml` file before precompilation to store the preferences (backend) used for JACC. Additionally, JACC leverages the recently introduced package extensions in Julia version 1.9 to allow for optional package dependencies or `weakdependencies`. Therefore, vendor-specific backend implementations (e.g., CUDA, AMDGPU, OneAPI) inside JACC can coexist via function overloading and multiple dispatches without incurring additional costs when installing JACC. The default backend is Julia's `Base.Threads` implementation, which targets CPUs.

The memory management in JACC is transparent to the programmer, and we use a very similar syntax to that used in other Julia packages: `JACC.Array`. `JACC.Array` is mapped on the equivalent Julia function depending on the target backend. Notably, when using `Base.Threads` as the backend, using `JACC.Array` is not necessary.

As depicted in Fig. 2, JACC has two primary constructs: `parallel_for` and `parallel_reduce`, and this is similar to metaprogramming solutions in other languages [3], [4]. We also included two variants to be chosen based on the data layout used: unidimensional or multidimensional. These constructs comprise three main components: (1) the number of iterations of the for-loop or reduction, which is typically equal to the size of the arrays; (2) the name of the function that defines the operations to be computed in each iteration of the loop; and (3) the parameters used in the function.

As shown, JACC provides a very simple way to parallelize codes by providing a unified front end that can be deployed on top of other Julia packages and use different architectures. When using JACC, programmers do not need to burden themselves with low-level details at the hardware or software levels, and this abstraction provides a high-level and portable solution to make Julia a productive programming solution for HPC, AI, and scientific software.

```
# Unidimensional arrays
function axpy(i, alpha, x, y)
  x[i] += alpha * y[i]
end

function dot(i, x, y)
  return x[i] * y[i]
end

SIZE = 1_000_000
x = round.(rand(Float64, SIZE) * 100)
y = round.(rand(Float64, SIZE) * 100)
alpha = 2.5
dx = JACC.Array(x)
dy = JACC.Array(y)
JACC.parallel_for(SIZE, axpy, alpha, dx, dy)
res = JACC.parallel_reduce(SIZE, dot, dx, dy)

# Multidimensional arrays
function axpy(i, j, alpha, x, y)
  x[i,j] = x[i,j] + alpha * y[i,j]
end

function dot(i, j, x, y)
  return x[i,j] * y[i,j]
end

SIZE = 1_000
x = round.(rand(Float64, SIZE, SIZE) * 100)
y = round.(rand(Float64, SIZE, SIZE) * 100)
alpha = 2.5
dx = JACC.Array(x)
dy = JACC.Array(y)
JACC.parallel_for((SIZE,SIZE),axpy,alpha,dx,dy)
res = JACC.parallel_reduce((SIZE,SIZE),dot,dx,dy)
```

Fig. 2. JACC front-end example.

III. JACC.SHARED

The hierarchy of memory in GPUs, depicted in Fig. 3 (right), usually comprises a local (cache) hierarchy memory with a private L1 cache per block of threads and an L2 cache shared by all the blocks of threads or multiprocessors. Besides the local memory, one other type of in-core memory exists: a software-managed (programmable) scratchpad memory that is a so-called shared memory. Each block of threads or multiprocessor features its own space within the shared memory. Only threads of the same thread block may share data through shared memory. Accesses to shared memory are usually one or two orders of magnitude faster than accesses to global memory.

The main difference between CPU on-chip memory and GPU on-chip memory is that in the case of GPUs, the programmer is responsible for using this memory (shared memory in Fig. 3), whereas the compiler can manage on-chip memory without the intervention of the programmer in the case of CPUs. In the case of GPU accelerators, these memories provide much higher bandwidths than the off-chip global memory. However, its control and management require manually loading pieces of data from global memory. To effectively use this kind of memory, multiple threads in the same block of threads must repeatedly access the same space of memory.

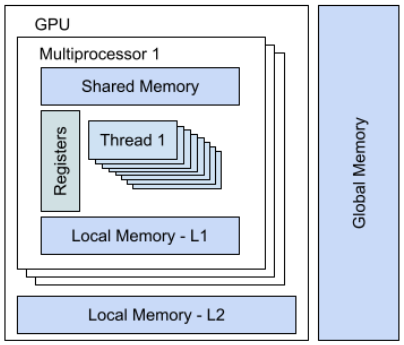

Fig. 3. On-chip memory hierarchy on CPUs and GPUs.

To simplify the control and use of on-chip memory, we implemented `JACC.shared`. `JACC.shared` is an easy-to-use and portable function implemented on top of all the JACC backends (Fig. 1). To keep this function as simple as possible, we propose the next syntax to be used in the Julia/JACC functions:

```
shared_array = JACC.shared(global_array)
```

where global_array is a `JACC.Array` (an array stored in the off-chip GPU global memory), and the `shared_array` type depends on the backend used—for instance, `cuDynamicSharedMem`, `ROCDynamicLocalArray`, `oneLocalArray`, or `Base.Array` for CUDA, AMDGPU, OneAPI, and Threads backends, respectively. The type used by the `shared_array` is transparent to the users. As a parameter, `JACC.shared` accepts any dimension and size for `global_array`; however, the output must be a unidimensional array. This is due to Julia's current limitations in dealing with shared memory in accelerators. `JACC.shared` internally deals with any necessary transformation from a bidimensional array to a 1D array if necessary. Additionally, the memory transfer from global memory to shared memory is computed in parallel, all the threads of a thread block are involved in this process. The size of shared memory in accelerators depends on the architecture to be used, so the array passed as the argument (`global_array`) for `JACC.shared` to be moved to shared memory must fit into the memory size limit of shared memory. In the case of the implementation of `JACC.shared` for CPUs, `JACC.shared` delegates to the compiler to apply any special optimization.

Unlike the other JACC features, `JACC.shared` is implemented to be used inside the functions, not outside. This does not influence other JACC capabilities, such as `JACC.Array`, `JACC.parallel_for`, or `JACC.parallel_reduce`. As an example, Fig. 5 illustrates a simple example code used for multi/hyperspectral computation with and without using `JACC.shared`. Unlike proposals from other metaprogramming solutions [5], [6] in which the use of shared memory must be defined externally, or its efficient exploitation depends on particular scenarios, `JACC.shared` allows JACC users

to define with no requirements where and when to use on-chip shared memory simply and transparently to improve performance for those applications that can leverage the capacities of on-chip GPU memory. To achieve the maximum performance when using `JACC.shared`, the threads per block to be used internally in `JACC.parallel_for` and `JACC.parallel_reduce` are recommended to be the highest possible number. Additionally, we configure the shared memory to use the maximum size possible depending on the architecture limits by using dynamic memory allocation, which requires specifying the size of this memory at kernel invocation time. All this is transparent to the programmer.

IV. Performance Evaluation

For performance evaluation, we use two well-known kernels, one for HPC (multi/hyperspectral imaging) and one for AI (convolutions), that can leverage on-chip GPU shared memory. Although both cases have similarities, we demonstrate that the benefit yielded by using `JACC.shared` can greatly differ depending on the particular characteristics or demands of the applications. One important factor to evaluate the efficiency reached by using on-chip GPU shared memory is the speedup reached depending on the number of times that a block of threads accesses shared memory—that is, the number of times that it does not access global memory. Generally, the higher this number, the better the performance.

A. Multi/Hyperspectral Imaging

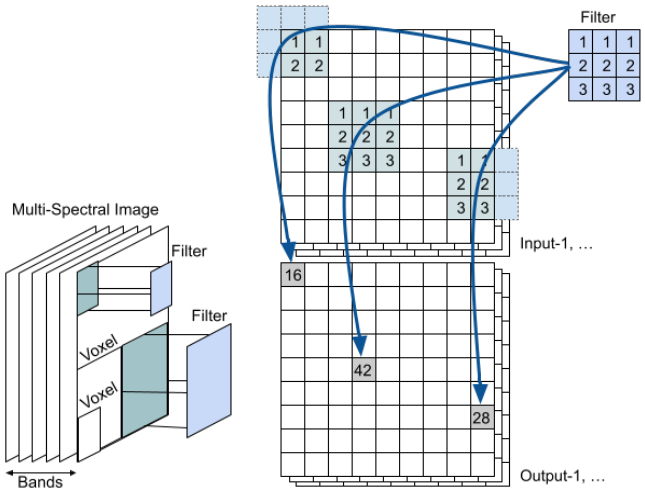

Fig. 4. Multi(hyper)spectral (left) and AI convolution (right) diagrams.

Multi/hyperspectral imaging [7] is a well-known problem applied to multiple applications, such as health care [8], space-based imaging [9], defense, remote sensing [10], farming, and environmental monitoring, among many others. Fig. 4-left illustrates a simple scheme representing the standard structure for this kind of operation. Depending on the application, we have a higher or a lower number of bands, each one representing a particular characteristic, such as color spectra, wavelengths, and spatial resolution. Every band is an image that comprises a set of voxels. The size of the image or number of voxels per image also depends on the application. Usually, a filter or a set of filters must be applied to each of the voxels that compose an image to capture a particular component, object, or characteristic of that band.

```
function spectral_shared(i, j, image, filter,
    num_bands)
 #Shared memory initialization
 filter_shared = JACC.shared(filter)
 for b in 1:bands
  @inbounds image[b, i, j] *= filter_shared[j]
 end
end

num_bands = 60
num_voxel = 10_240
size_voxel = 64*64
image = init_image(Float32,
        num_bands, num_voxel, size_voxel)
filter = init_filter(Float32, size_voxel)
jimage = JACC.Array(image)
jfilter = JACC.Array(filter)
JACC.parallel_for((num_voxel,size_voxel),
     spectral, jimage, jfilter, num_bands)
```

Fig. 5. `JACC.shared` example for multispectral computation.

To keep the analysis simple and general, we used the kernels illustrated in Fig. 5 and selected a set of representative parameters corresponding to the three main components of this kind of operation: number of bands, size of the images, and size of voxels/filter (Fig. 6). This is a simple kernel in which a filter is applied to all the voxels of each of the images. As shown in Fig. 4-left, the filter must be accessed by all the threads. To avoid using a higher number of threads than the one supported by the GPUs, every thread accesses a specific part (pixel) of the filter and applies it to the corresponding pixel of a given voxel for each of the images or bands of the multi/hyperspectral image. So, in this case, moving the filter from global memory to shared memory should provide better performance.

As shown in Fig. 6, the number of bands is the most influential factor when using shared memory, reaching a speedup on the NVIDIA GPU of about $1.4\times$ and $1.83\times$ for multispectral images of 30 and 60 bands, respectively. Generally, the more bands, the better the performance. This is, in fact, an important increment of performance given the relatively low number of memory accesses to shared memory w.r.t. the total memory accesses to global memory and the minimal modification required to reach such a speedup (just adding one line of code). On the AMD GPU, no speedup is achieved when using shared memory. This is caused by a higher software and/or hardware overhead when using this kind of memory on AMD GPUs than on NVIDIA GPUs. The support for AMD GPUs in the Julia ecosystem is still not as mature as the support for NVIDIA GPUs. Additionally, differences exist in the capabilities (bandwidth) for shared memory in both architectures, NVIDIA and AMD, thereby also affecting the performance, which may require a higher usage of shared

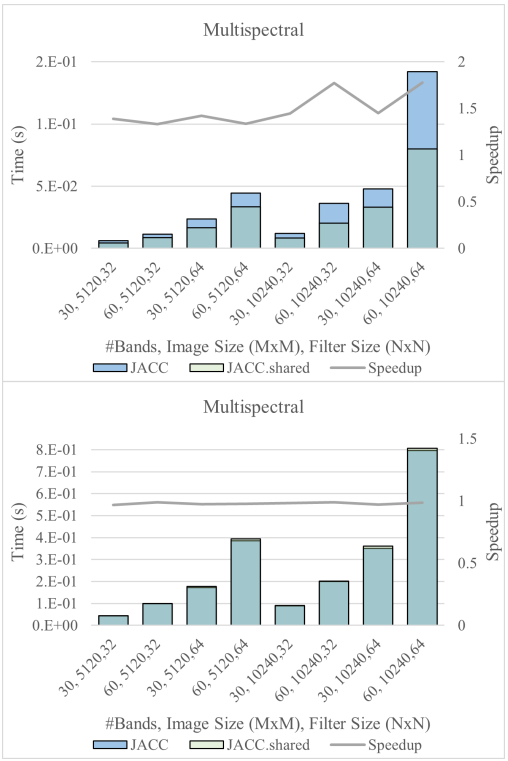

Fig. 6. Multispectral performance (NVIDIA A100 GPU *top* and AMD MI250X GPU *bottom*).

memory to obtain the desired extra performance, as will be demonstrated in the next subsection.

B. Convolution

Convolutions are the core operation of deep learning applications based on convolutional neural networks (CNNs). CNNs have become a key operation for AI. This interest comes from their impressive results in tasks such as image classification, speech recognition, and natural language processing. Two main factors were necessary to enable the success of CNNs: (1) the availability of large datasets and (2) the high performance of current computing systems. Large datasets are needed to train the deep neural network parameters until a highly accurate result is reached. In turn, such an amount of data requires the use of HPC accelerators to keep the training time of deep neural networks within reasonable limits.

CNNs are based on the use of convolutional layers, which are the result of weighted sums of inputs, like in fully connected layers. Convolutional layers use an operation called *convolution* to implement the weighted sums. Most of the execution time of a convolutional layer is spent performing convolutions. A convolution operation is a 2D discrete convolution, and it uses a 3D input and a filter. Each output element is computed as the dot product of the filter with a subvolume of the input, as depicted in Fig. 4-right.

Given the particular characteristics of this operation, the granularity used for this operation is higher than the one used in the previous case. Every thread is responsible for computing

```
function convolution_shared(i, j, input, ouput,
    filter, filter_size, num_inputs)
  #Shared memory initialization
  filter_shared = JACC.shared(filter)
  for n in 1:num_inputs
    find = 1
    conv = 0.0
    i_ind = i - filter_size
    j_ind = j - filter_size
    for fi in 1:filter_size
      for fj in 1:filter_size
        if (i_ind + fi > 0) && (j_ind + fj > 0)
          @inbounds conv += filter_shared[find] *
              input[n, i_ind + fi, j_ind + fj]
        end find += 1
      end find += 1
    end
    @inbounds ouput[n, i, j,] = conv
  end
end

SIZE_x = 1024
SIZE_y = 1024
NUM_inputs = 256
SIZE_filter = 5
input = init_input(Float32,
        NUM_inputs, SIZE_x, SIZE_y)
output = init_output(Float32,
         NUM_inputs, SIZE_x, SIZE_y)
filter = init_filter(Float32,
         SIZE_filter*SIZE_filter)
jinput = JACC.Array(input)
joutput = JACC.Array(output)
jfilter = JACC.Array(filter)
JACC.parallel_for((SIZE_x,SIZE_y),
     convolution, jinput, joutput, filter,
     SIZE_filter, NUM_inputs)
```

Fig. 7. `JACC.shared` example for convolution computation.

a dot product on a particular part of the input and the filter and for storing the result of that operation in the proper position of the output. As shown in Fig. 4-right, every thread must access the same filter, so moving the filter to shared memory should improve the performance. By using this distribution or granularity, we maximize the use of shared memory such that each thread of the same block of threads must access the same entire filter multiple times instead of just one element of the filter (as in the multi[hyper]spectral test case). For clarity and simplicity, Fig. 7 illustrates a simple JACC code that computes convolution.

Unlike in the test case running the multi(hyper)spectral imaging code, here we have a much higher number of accesses to shared memory per block of threads, which is about 3 million memory accesses for the biggest configuration (256, 1,024, 5) evaluated when the number of accesses to global memory is about the same. As a result, we observe (Fig. 8) a much better speedup when using shared memory in this case compared with the test case running the multispectral imaging code. On the NVIDIA GPU, we observe a speedup of up to 3.5× when using shared memory. Once again, this extra performance required adding only one line of code to the base code (Fig. 7). The most impactful factor for performance is the size of the input. On the AMD GPU, the performance gain reached by using shared memory, although smaller than

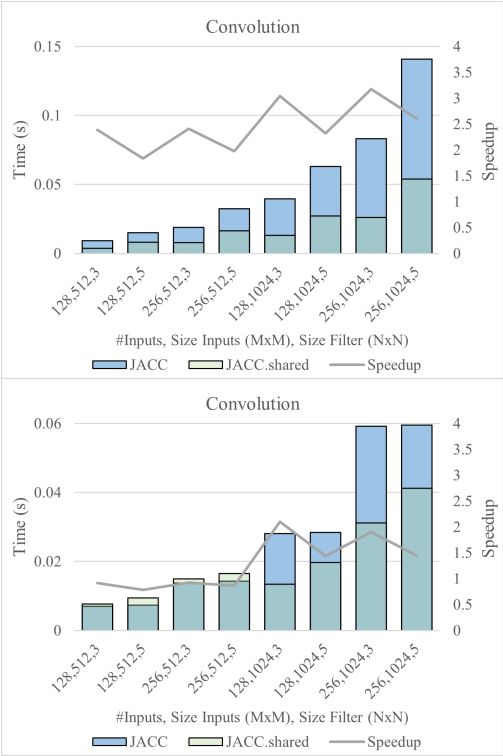

Fig. 8. Convolution performance (NVIDIA A100 GPU *top* and AMD MI250X GPU *bottom*).

on the NVIDIA GPU, is much greater than in the previous test case when running the multispectral imaging code, thus constituting a speedup of up to $2.1\times$. On the smallest test cases, with an input size equal to 512^2, we do not observe better performance when using shared memory on the AMD GPU. This is in agreement with the results presented in the previous application, strengthening the hypothesis that a higher overhead exists in the use of shared memory on the AMD GPU than on the NVIDIA one.

V. Related Work

Previously to this work, these authors evaluated the performance of Julia against other languages or models such as Kokkos, Python, OpenMP, HIP, or CUDA on multiple HPC configurations, concluding that Julia is competitive or even better in terms of performance than other C/C++ vendor-specific or open-source models [11].

The use of high-bandwidth, on-chip memory in accelerators requires low-level programming efforts, which in some cases can be complicated to implement. However, we can find multiple examples and applications in which effective use of this kind of memory is supposed to achieve important increments in performance—for instance, in image processing [12], [13], string matching (information retrieval) [14], [15], computational fluid dynamics [16]–[18], sparse [19], [20] and dense [21], [22] linear algebra, and AI [23], among others. As a result, modern and high-level programming solutions have proposed different ways to bring this capability to their specifications. For instance, `#pragma acc cached` was proposed in OpenACC as a way to use shared memory. We can find similar efforts in the C++ metaprogramming models RAJA [6] and Kokkos [5], [24] with `RAJA::LocalArray`, which is defined externally as another array to use in a RAJA application, or with `scratch_memory_space` in Kokkos. These last efforts depend on the backend or specific scenarios, such as the use of Team policy in Kokkos, for the shared memory to be used. Unfortunately, all these efforts present some limitations. For example, they are not supported in all the backends, not all the compilers provide support, and they are implemented in a way that does not leverage this kind of memory.

VI. Conclusions

This study describes the efforts for `JACC.shared`, which is a novel feature part of JACC, the metaprogramming and performance-portable model for the Julia programming language. `JACC.shared` is a transparent Julia-like function that users can easily use to define when and where to use on-chip GPU memory.

`JACC.shared` is compatible with CPUs, NVIDIA GPUs, AMD GPUs, and Intel GPUs. The performance that the users can reach by using `JACC.shared` depends on application and hardware characteristics. For the test cases evaluated, multispectral imaging and AI convolutions, we achieved a speedup of up to $1.8\times$ and $3.5\times$ on the first and second test cases, respectively. The NVIDIA A100 GPU performs better than the AMD MI250X GPU when on-chip shared memory is used.

With `JACC.shared`, we increment the performance portability and the programming productivity for Julia/JACC applications to yield greater performance. For this effort, we use the high-bandwidth, on-chip memory on a large variety of GPUs by adding only one line of code to the base code.

Acknowledgment

This research used resources of the Oak Ridge Leadership Computing Facility and the Experimental Computing Laboratory (ExCL) at the Oak Ridge National Laboratory, which is supported by the Office of Science of the US Department of Energy under Contract No. DE-AC05-00OR22725. This research was funded in part by the DOE ASCR Stewardship for Programming Systems and Tools (S4PST) project, and by Bluestone, a X-Stack project in the DOE ASCR Office. Notice: This manuscript has been authored by UT-Battelle LLC under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (https://www.energy.gov/doe-public-access-plan).

References

[1] V. Churavy, W. F. Godoy, C. Bauer, H. Ranocha, M. Schlottke-Lakemper, L. Räss, J. Blaschke, M. Giordano, E. Schnetter, S. Omlin, J. S. Vetter, and A. Edelman, "Bridging HPC Communities through the Julia Programming Language," submitted for review, 2022.

[2] S. Byrne, L. C. Wilcox, and V. Churavy, "MPI.jl: Julia bindings for the Message Passing Interface," *Proceedings of the JuliaCon Conferences*, vol. 1, no. 1, p. 68, 2021. [Online]. Available: https://doi.org/10.21105/jcon.00068

[3] D. Beckingsale, R. D. Hornung, T. Scogland, and A. Vargas, "Performance portable C++ programming with RAJA," in *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, J. K. Hollingsworth and I. Keidar, Eds. ACM, 2019, pp. 455–456. [Online]. Available: https://doi.org/10.1145/3293883.3302577

[4] C. Trott, L. Berger-Vergiat, D. Poliakoff, S. Rajamanickam, D. Lebrun-Grandié, J. Madsen, N. A. Awar, M. Gligoric, G. Shipman, and G. Womeldorff, "The kokkos ecosystem: Comprehensive performance portability for high performance computing," *Comput. Sci. Eng.*, vol. 23, no. 5, pp. 10–18, 2021. [Online]. Available: https://doi.org/10.1109/MCSE.2021.3098509

[5] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Q. Dang, N. D. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. R. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. J. Wilke, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Trans. Parallel Distributed Syst.*, vol. 33, no. 4, pp. 805–817, 2022. [Online]. Available: https://doi.org/10.1109/TPDS.2021.3097283

[6] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. Scogland, "Raja: Portable performance for large-scale scientific applications," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2019, pp. 71–81.

[7] M. E. Paoletti, J. M. Haut, X. Tao, J. Plaza, and A. Plaza, "A new GPU implementation of support vector machines for fast hyperspectral image classification," *Remote. Sens.*, vol. 12, no. 8, p. 1257, 2020. [Online]. Available: https://doi.org/10.3390/rs12081257

[8] P. Valero-Lara, J. L. Sánchez, D. Cazorla, and E. Arias, "A gpu-based implementation of the MRF algorithm in ITK package," *J. Supercomput.*, vol. 58, no. 3, pp. 403–410, 2011. [Online]. Available: https://doi.org/10.1007/s11227-011-0597-1

[9] P. Valero-Lara, "MRF satellite image classification on GPU," in *41st International Conference on Parallel Processing Workshops, ICPPW 2012, Pittsburgh, PA, USA, September 10-13, 2012*. IEEE Computer Society, 2012, pp. 149–156. [Online]. Available: https://doi.org/10.1109/ICPPW.2012.24

[10] M. Chi, A. Plaza, J. A. Benediktsson, Z. Sun, J. Shen, and Y. Zhu, "Big data for remote sensing: Challenges and opportunities," *Proc. IEEE*, vol. 104, no. 11, pp. 2207–2219, 2016. [Online]. Available: https://doi.org/10.1109/JPROC.2016.2598228

[11] W. F. Godoy, P. Valero-Lara, T. E. Dettling, C. Trefftz, I. Jorquera, T. Sheehy, R. G. Miller, M. G. Tallada, J. S. Vetter, and V. Churavy, "Evaluating performance and portability of high-level programming models: Julia, python/numba, and kokkos on exascale nodes," in *IEEE International Parallel and Distributed Processing Symposium, IPDPS 2023 - Workshops, St. Petersburg, FL, USA, May 15-19, 2023*. IEEE, 2023, pp. 373–382. [Online]. Available: https://doi.org/10.1109/IPDPSW59300.2023.00068

[12] P. Valero-Lara, "MRF satellite image classification on GPU," in *41st International Conference on Parallel Processing Workshops, ICPPW 2012, Pittsburgh, PA, USA, September 10-13, 2012*. IEEE Computer Society, 2012, pp. 149–156. [Online]. Available: https://doi.org/10.1109/ICPPW.2012.24

[13] ——, "Accelerating solid-fluid interaction based on the immersed boundary method on multicore and GPU architectures," *J. Supercomput.*, vol. 70, no. 2, pp. 799–815, 2014. [Online]. Available: https://doi.org/10.1007/s11227-014-1262-2

[14] ——, "hlcs. A hybrid GPGPU approach for solving multiple short and unbalanced LCS problems," in *Computational Science and Its Applications - ICCSA 2014 - 14th International Conference, Guimarães, Portugal, June 30 - July 3, 2014, Proceedings, Part VI*, ser. Lecture Notes in Computer Science, B. Murgante, S. Misra, A. M. A. C. Rocha, C. M. Torre, J. G. Rocha, M. I. Falcão, D. Taniar, B. O. Apduhan, and O. Gervasi, Eds., vol. 8584. Springer, 2014, pp. 102–115. [Online]. Available: https://doi.org/10.1007/978-3-319-09153-2_8

[15] R. U. Paredes, P. Valero-Lara, E. Arias, J. L. Sánchez, and D. Cazorla, "Similarity search implementations for multi-core and many-core processors," in *2011 International Conference on High Performance Computing & Simulation, HPCS 2012, Istanbul, Turkey, July 4-8, 2011*, W. W. Smari and J. P. McIntire, Eds. IEEE, 2011, pp. 656–663. [Online]. Available: https://doi.org/10.1109/HPCSim.2011.5999889

[16] P. Valero-Lara, "Accelerating solid-fluid interaction based on the immersed boundary method on multicore and GPU architectures," *J. Supercomput.*, vol. 70, no. 2, pp. 799–815, 2014. [Online]. Available: https://doi.org/10.1007/s11227-014-1262-2

[17] J. Gounley, M. Vardhan, E. W. Draeger, P. Valero-Lara, S. V. Moore, and A. Randles, "Propagation pattern for moment representation of the lattice boltzmann method," *IEEE Trans. Parallel Distributed Syst.*, vol. 33, no. 3, pp. 642–653, 2022. [Online]. Available: https://doi.org/10.1109/TPDS.2021.3098456

[18] P. Valero-Lara, J. S. Vetter, J. Gounley, and A. Randles, "Moment representation of regularized lattice boltzmann methods on NVIDIA and AMD gpus," in *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis, SC-W 2023, Denver, CO, USA, November 12-17, 2023*. ACM, 2023, pp. 1697–1704. [Online]. Available: https://doi.org/10.1145/3624062.3624250

[19] P. Valero-Lara, A. Pinelli, and M. Prieto-Matías, "Fast finite difference poisson solvers on heterogeneous architectures," *Comput. Phys. Commun.*, vol. 185, no. 4, pp. 1265–1272, 2014. [Online]. Available: https://doi.org/10.1016/j.cpc.2013.12.026

[20] Valero-Lara, Pedro, Martínez-Pérez, Ivan, Sirvent, Raül, Peña, Antonio J., Martorell, Xavier, and Labarta, Jesús, "Simulating the behavior of the human brain on gpus," *Oil Gas Sci. Technol. – Rev. IFP Energies nouvelles*, vol. 73, p. 63, 2018. [Online]. Available: https://doi.org/10.2516/ogst/2018061

[21] J. J. Dongarra, S. Hammarling, N. J. Higham, S. D. Relton, P. Valero-Lara, and M. Zounon, "The design and performance of batched BLAS on modern high-performance computing systems," in *International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland*, ser. Procedia Computer Science, P. Koumoutsakos, M. Lees, V. V. Krzhizhanovskaya, J. J. Dongarra, and P. M. A. Sloot, Eds., vol. 108. Elsevier, 2017, pp. 495–504. [Online]. Available: https://doi.org/10.1016/j.procs.2017.05.138

[22] N. R. Miniskar, M. A. H. Monil, P. Valero-Lara, F. Liu, and J. S. Vetter, "IRIS-BLAS: towards a performance portable and heterogeneous BLAS library," in *29th IEEE International Conference on High Performance Computing, Data, and Analytics, HiPC 2022, Bengaluru, India, December 18-21, 2022*. IEEE, 2022, pp. 256–261. [Online]. Available: https://doi.org/10.1109/HiPC56025.2022.00042

[23] M. Jordà, P. Valero-Lara, and A. J. Peña, "cuconv: CUDA implementation of convolution for CNN inference," *Clust. Comput.*, vol. 25, no. 2, pp. 1459–1473, 2022. [Online]. Available: https://doi.org/10.1007/s10586-021-03494-y

[24] P. Valero-Lara, S. Lee, M. G. Tallada, J. E. Denny, and J. S. Vetter, "Kokkacc: Enhancing kokkos with openacc," in *9th Workshop on Accelerator Programming Using Directives, WACCPD@SC 2022, Dallas, TX, USA, November 13-18, 2022*. IEEE, 2022, pp. 32–42. [Online]. Available: https://doi.org/10.1109/WACCPD56842.2022.00009