2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)

2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS) | 978-1-6654-5185-7/22/$31.00 ©2022 IEEE | DOI: 10.1109/PMBS56514.2022.00013

Benchmarking Fortran `DO CONCURRENT` on CPUs and GPUs Using BabelStream

Jeff R. Hammond
NVIDIA Helsinki Oy
Helsinki, Finland
jeff_hammond@acm.org

Tom Deakin, James Cownie and Simon McIntosh-Smith
HPC Research Group, Department of Computer Science, University of Bristol
Bristol, UK
tom.deakin@bristol.ac.uk,jcownie@acm.org,S.McIntosh-Smith@bristol.ac.uk

***Abstract*—Fortran `DO CONCURRENT` has emerged as a new way to achieve parallel execution of loops on CPUs and GPUs. This paper studies the performance portability of this construct on a range of processors and compares it with the incumbent models: OpenMP, OpenACC and CUDA. To do this study fairly, we implemented the BabelStream memory bandwidth benchmark from scratch, entirely in modern Fortran, for all of the models considered, which include Fortran `DO CONCURRENT`, as well as two variants of OpenACC, four variants of OpenMP (2 CPU and 2 GPU), CUDA Fortran, and both loop- and array-based references. BabelStream Fortran matches the C++ implementation as closely as possible, and can be used to make language-based comparisons. This paper represents one of the first detailed studies of the performance of Fortran support on heterogeneous architectures; we include results for AArch64 and x86_64 CPUs as well as AMD, Intel and NVIDIA GPU platforms.**

***Index Terms*—Fortran, parallel programming, GPUs, multi-core, memory bandwidth**

I. Introduction

Two of the biggest challenges for high-performance computing (HPC) applications are memory bandwidth and the performance portability of parallel programming models. Trends in computer architecture favor logic (compute performance) over I/O, and in most modern supercomputing platforms, the flops-per-byte metric – how much numerical computation must be done on a single element to achieve peak performance – is incredibly high. At the same time, increased on-chip parallelism in all processors, as well as the rise of GPU-based HPC platforms has led to a diversification of parallel programming models. Application programmers are faced with a large number of choices in how to evolve existing code or create new code that takes advantage of the full capability of modern HPC systems.

The BabelStream benchmark has successfully addressed the issue of measuring memory bandwidth achieved by a wide range of parallel programming models based on the C++ language, as well as a few other languages: Java, Julia, Rust and Scala. Fortran has been conspicuously absent, despite its status as the longest and probably the most widely used programming language for HPC applications. While Fortran does not have as many parallel programming models built upon it as C++, there are still enough to choose from that it is useful to benchmark all of them. The most common Fortran-based parallel models that support HPC processors are:

- OpenMP: the most established form of directives for parallelism within a node, OpenMP 5 offers a variety of constructs for parallel loops for both CPU and GPU architectures.
- OpenACC: the first standard form of parallel directives for GPUs, which is used in many Fortran applications.
- DoConcurrent: Fortran 2008 added the `DO CONCURRENT` loop construct, which can be parallelized in a variety ways, depending on the compiler.
- CUDA Fortran: the CUDA programming model is supported by the NVIDIA Fortran compiler, and offers similar features to the CUDA C/C++ language.

BabelStream supports simple and universally understood patterns for measuring memory bandwidth, which include the Copy, Add, Mul(tiplication) and Triad kernels from McCalpin's STREAM [1], [2], and the Dot kernel (inner product of vectors). We have also implemented the BabelStream version nstream, inspired by Parallel Research Kernels [3], [4], although results are not reported here.

Since Fortran 90, it has been possible to express data parallel operations in array-notation, rather than as loops over element-wise operations. Furthermore, some of the parallel programming models considered here support these constructs. We have implemented both loop-based and array-based versions, with parallelism where possible. Unfortunately, not all compilers support all possible variations in a useful way; one purpose of this paper and the associated benchmarks is to document which compilers support which models correctly in parallel.

The existence of BabelStream Fortran enables the following experiments:

- Language evaluation: Fortran can be compared against C++ and other languages.
- Parallel programming model evaluation: application programmers are often forced to make trade-offs between portability and performance – this is as true in Fortran as it is in C++.
- Compiler evaluation: Most processors are supported by at least two compilers, and it is important for application users to know which one delivers the best user experience (ease of use, performance and correctness).
- Hardware evaluation: BabelStream already facilitates this,

TABLE I
PSEUDOCODE FOR THE KERNELS IMPLEMENTED IN BABELSTREAM.

Kernel	Loop Body	Array Notation
Copy	`c(i) = a(i)`	`C = A`
Add	`c(i) = a(i) + b(i)`	`C = A + B`
Mul	`c(i) = a(i) * b(i)`	`C = A * B`
Dot	`r    = r + a(i) * b(i)`	`r = DOT_PRODUCT(A,B)`
Triad	`c(i) = a(i) + s * b(i)`	`C = A + s * B`
Nstream	`c(i) = c(i) + a(i) + s * b(i)`	`C = C + A + s * B`

but it is of course useful to be able to do it in a Fortran context.

We have performed all four sets of experiments, albeit not exhaustively. In particular, we compared Fortran compilers from the Arm Compiler for Linux (ARM), Fujitsu compilers, GNU Compiler Collection (GCC), HPE Cray Compiling Environment (CCE), Intel compilers, and NVIDIA HPC Software Development Kit (NVHPC) across a range of AArch64 and x86 CPUs, as well as GPUs from AMD, Intel, and NVIDIA. The hardware measured includes the most recent architectures from multiple vendors – AMD Milan CPU, AMD MI100 GPU, Ampere Altra Q80 CPU, AWS Graviton 2 CPU, AWS Graviton 3 CPU, Fujitsu A64FX, Intel Ice Lake server CPUs, Intel Gen12LP X^e integrated graphics (iGPU), NVIDIA Orin (ARM A78 cores) CPU, NVIDIA Ampere (A100) and Turing (TU106) GPUs – a list which includes nearly all current, relevant platforms for large-scale HPC. The most obvious omission is the AMD MI-200 series of GPUs, which is omitted due to the lack of timely access to a system.

The novel contributions of this paper include:

- Comparison of Fortran `DO CONCURRENT` to incumbent directive-based programming models (OpenMP and OpenACC) on both CPUs and GPUs.
- Evaluation of many parallel programming models for Fortran using six different compilers, across a range of hardware.
- Bandwidth measurements on the latest HPC processors, some of which may be reported here for the first time.
- A new language implementation of BabelStream, which supports the parallel programming models of interest to many HPC users.

II. BACKGROUND AND RELATED WORK

A. Background

It is common for people to think that Fortran is "a legacy language", and that it is unimportant or dying. There are a number of reasons for this, including programming language popularity surveys, such as the Tiobe index [5] which has Fortran in 19th place in August 2022, or the IEEE Spectrum assessment [6], which has Fortran in 25th place for 2021. Both of these indices are intended to help people decide which languages to learn, and people starting new projects to choose an appropriate programming language for which there will be many programmers available. They do not, therefore, measure the amount of computation being performed by code written in the given language, which is the metric that matters to hardware vendors. Nor do they consider the importance of the long lived, well validated, codes used in high-performance computing.

While it is generally hard to obtain information on how large HPC machines are used, and which languages are therefore important, such information is published for the Archer2 machine [7] (the UK National Supercomputing Service). From that data [8], and with the help of the Archer2 team, who have application to language information, we can produce summary information about machine usage over the six months March-August 2022. In Fig. 1 we can see that when we aggregate over

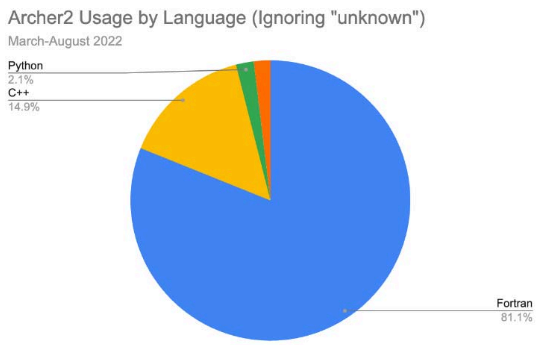


Fig. 1. Archer2 machine usage by language assuming that unknown codes use the same proportion of languages as known ones.

all codes that use more than 0.1% of the machine, applications that require Fortran consume over 80% of the node-hours. [9]. Since this machine has 5,860 dual-socket nodes, that means that Fortran accounts for the sale of over 9,400 high end CPUs, or, as the machine is believed to have cost ∼£50M ($63M at the time), the Fortran codes justified spending over £40M ($50M).

Anecdotally, there are hundreds of millions of lines of Fortran code in use across a wide range of sectors. Even the financial sector – not known for its use of Fortran, but of COBOL and Java – makes significant use of it [10]. There are at least five, and probably many more, quantum chemistry applications in use today with approximately one million lines of code each, while institutions associated with the maintenance of nuclear weapons estimate their Fortran portfolios to be many millions of lines, per lab. For obvious reasons, neither financial firms nor defense agencies allow their code to be analyzed by third parties, so it is not too surprising that Fortran usage is under-counted in modern internet surveys of open source code.

Another issue is the tendency of computer science research and education to follow the latest trends, which means that recently graduated compiler developers are far more interested in optimizing deep neural network domain specific languages than Fortran. Thus, finding compiler developers who can implement and optimize the latest Fortran standard features is challenging, and disincentivizes users from adopting these features, unless they have the means to drive compiler development.

B. Related Work

The STREAM benchmark [1], [2], created by John McCalpin, is well-established as the canonical reference for memory bandwidth measurements in HPC. However, the official STREAM benchmark contains implementations using C89 and classic Fortran, with OpenMP 3 parallelism, which means that it does not support GPUs.

BabelStream was created to allow STREAM-like measurements of the new forms of parallelism supported only in C++, such as Kokkos [11], [12], RAJA [13]–[15] and SYCL [16], [17]. BabelStream [18], [19] also supports parallel models that run on GPUs and possible other non-CPU architectures, via a wide range of programming models, including OpenMP [20], OpenACC [21], CUDA [22], HIP [23], and OpenCL [17], [24]. It has previously been used to evaluate processor and compiler performance on AArch64 and x86 CPUs [25], [26] as well as processor and parallel programming model performance on GPUs [27].

Another project that aspires to have similar coverage of programming languages and parallel programming models is the Parallel Research Kernels (PRKs) [3], [4]. The PRKs include a variant of STREAM triad (nstream, described in Section I, and show in Table I) as well as a range of other common patterns that exercise various properties of the machine hierarchy and/or forms of synchronization. It supports a similar array of languages and single-node parallel models for nstream, but also supports distributed-memory parallelism (e.g. MPI and PGAS [28]–[30]).

Some of the PRKs are themselves derived, at least in spirit, from the well-known NAS Parallel Benchmarks (NPB) [31] and the HPC Challenge (HPCC) benchmarks [32], [33], which have been used for programming language and parallel model evaluation (e.g. OpenACC [34]). HPCC includes an embarrassingly parallel version of STREAM triad, which has been used to benchmark HPC systems [35].

SHOC [36] is another set of benchmarks used to evaluate parallel programming models, although the current version is limited to MPI, CUDA, and OpenCL. RAJAPerf [37], [38] has also been developed for parallel programming model evaluation, and includes STREAM-like kernels, though it is focused exclusively on C++, by virtue of its connection to the RAJA framework.

We are not aware of widely used HPC benchmarks written in modern Fortran; although SpecHPC [39] includes 3 Fortran codes, none of them seem to use `do concurrent`; hence BabelStream Fortran may be useful to those looking for codes that do use it.

III. Implementation Details

While the GitHub source code [40] is the authoritative version of what was used for this paper, it is worth summarizing the design and kernel details here.

BabelStream Fortran is written entirely in modern Fortran and follows the C++ design with one major exception: instead of defining a class for each implementation which contains the kernels as member functions, we created a module for each implementation. Each implementation module contains the same kernel functions, as well as allocation and deallocation functions, which take the place of the C++ class constructor and destructor. All memory management is done with Fortran allocation, modified as appropriate using directives or compiler flags for GPU execution. While the code uses Fortran standard features whenever possible, there is an option to use OpenMP for timing instead of Fortran's `system_clock`, which has low resolution (~ 0.001s) with at least one compiler (Fujitsu).

The input parser is written in Fortran and behaves identically to the C++ implementation, though the option to emit results in CSV format is not yet supported[1]. The regular output format matches that of the C++ code almost exactly. While trivial, this feature should allow BabelStream users who have written scripts that consume the (non-CSV) output to continue to use them with the Fortran code.

Unfortunately, due to the lack of templates in Fortran, switching between 32- and 64-bit floating-point (`real(kind=REAL32)` and `real(kind=REAL64)`) is a compile-time option. All results in this paper use 64-bit floating-point, and we have not yet validated the 32-bit floating-point version. There is also an option to switch loop indices from 64- to 32-bit integers, since this may have a small impact on performance in some processors. All results in this paper use 64-bit integer indexing, although some GPU compilers may use 32-bit addressing internally.

The build system uses GNU Make, and selects different implementations in the Fortran source code using preprocessor logic, similar to the C++ version. Make include files support the various compilers, and users must verify that architecture-specific flags such as `-march=..` and `-gpu=ccXX` are correct for their platform. To generate binaries, one invokes "`make COMPILER=.. IMPLEMENTATION=..`". The use of unsupported or incorrect build options will produce a message to the user about the correct and supported options. Currently, AMD (AOCC and ROCM), ARM, Cray, Fujitsu, GCC, Intel (`ifort` and `ifx`), and NVHPC compilers are supported.

The key kernel patterns are shown in Listings 1 and 2 in the Appendix, where the latter relies on one of the parallel loop directives shown in Table II. For each parallel loop directive, we use an idiomatic expression that expresses all

[1]This should be fixed by the time the paper appears.

of the available parallelism but do not specify any processor- or compiler-specific options. Most of the constructs are implemented correctly in all of the compilers tested – assuming the compiler supports the model at all, of course – although we found correctness problems in the Dot kernel in a few cases. When using DoConcurrent, only the NVHPC compiler supports reductions correctly, which it does whether or not the `reduce` locality specifier is used, due to scalar reduction pattern recognition, which is also supported in CUDA Fortran. While the Cray and Intel compilers support DoConcurrent parallelism on CPUs, neither supports the `reduce` locality specifier, nor do they do pattern recognition on the pattern. It is reasonable to expect both of these compilers will support Dot correctly once they support Fortran 2023, which is when the `reduce` locality specifier is officially added to Fortran. Because of these issues and the lack of support for any locality specifier in GCC Fortran [41], the current implementation of BabelStream Fortran avoids all of these. Based on the performance results, the other relevant locality specifiers – `shared` and `local_init` – do not appear to be necessary, at least for the types of simple kernels in BabelStream.

TABLE II
SOURCE CODE OF THE PARALLEL DIRECTIVES APPLIED TO LOOPS. THE SYNTAX FOR REDUCTIONS IS `REDUCTION(+:R)` IN ALL CASES EXCEPT CUDA FORTRAN, WHERE IT IS UNNECESSARY.

Parallelism	Directive
CUDA Fortran Kernels	`!$cuf kernel do <<< * , * >>>`
OpenACC	`!$acc parallel loop`
OpenMP	`!$omp parallel do simd`

OpenMP Target	`!$omp target teams distribute`
	`!$omp&       parallel do simd`

OpenMP Target Loop	`!$omp target teams loop`

OpenMP Taskloop	`!$omp parallel`
	`!$omp master`
	`!$omp taskloop`

We do not show a template for the cases with array notation, as they are simply the composition of the definition shown in Table I with the directives shown in Table III. The implementation quality of array expressions, particularly with parallel directives, varied significantly across compilers. The only cases where Dot supports useful parallelism from the `DOT_PRODUCT` intrinsic was with the Cray and Fujitsu compilers on CPUs and with the NVHPC compiler using OpenACC `kernels` (both CPU and GPU). The Intel compiler appears to attempt to generate some parallelism with OpenMP `workshare`, but the performance and CPU utilization indicates it has very low efficiency. The ARM compiler produces incorrect results when `workshare` is applied to `DOT_PRODUCT`, which may be a bug in the upstream LLVM Flang compiler.

Finally, for CUDA Fortran, we implement two approaches. The first uses the C-style design where GPU kernels are written using explicit block and thread indices, although we omitted Dot due its inherent lack of performance portability (discussed in detail in Section IV-C). The implementation pattern is shown in Listing 3 in the Appendix. The second

TABLE III
SOURCE CODE OF THE PARALLEL DIRECTIVES APPLIED TO ARRAY STATEMENTS.

Parallelism	Directive
OpenACC	`!$acc kernels`
OpenMP	`!$omp workshare`

approach uses the ‘kernels’ directive shown in Table II, which generates very similar results to explicit CUDA Fortran kernels as well as OpenACC loops. For verification purposes, we use the CUDA kernels implementation of Dot in both CUDA implementations. One important performance tip is that assumed-shape arrays (e.g. `C(:)`) should not be used in CUDA Fortran kernels, because they cause the compiler to construct an array descriptor on every CUDA thread, whereas assumed-size arrays (e.g. `C(*)`) or known-size arrays (e.g. `C(n)`), do not.

While Fortran 2008 added a second form of parallelism, coarrays, it is not supported in BabelStream Fortran because it is not a shared-memory programming model, nor does it offer any way to support GPUs.

IV. RESULTS

We first compared Fortran implementations against their C++ counterparts, where possible, to verify that nothing was lost in translation. Obviously, language semantics and compiler quality cause differences in some code, but for the BabelStream kernels, we expect very little difference due to either.

The second set of experiments was to compare the different Fortran implementations on a range of hardware, to understand the achievable bandwidth with various programming models. While our focus was HPC server processors, we also include smaller form factors where these provide architectures not available anywhere else. For example, due to the lack of any available Intel server GPUs, we could only evaluate Intel OpenMP target support on an integrated GPU in a laptop-grade CPU. We also evaluated the NVIDIA Jetson AGX Orin development kit [42], because it is a high-performance, small form-factor AArch64 platform that officially supports Linux, and the Cortex A78 core used therein is similar to the Neoverse N1 core used in server CPUs.

One key feature of our experiments is that there are no hardware- or compiler-specific code optimizations. This may lead to less than ideal results in some cases, but a key aspect of performance portability is that application developers do not want to specialize their code – particularly when the algorithms are far more complex than BabelStream – for every processor and compiler. We also use, in most cases, generic compiler flags (`-O3`) and specify the processor targeted, although we exploit two somewhat specific flags in the case of the Intel compiler (to generate non-temporal stores) and the Fujitsu compiler, due to special characteristics of the A64FX CPU.

The goal of this paper is to establish a performance baseline for BabelStream Fortran using idiomatic, platform-agnostic code, so that tuned implementations can be explored in the future.

A. Hardware details

In Table IV, we list the various systems and process types used for experiments. A more detailed version of this table with links to technical specifications, as well as a list of operating systems, compiler versions, and compiler flags, can be found in the Appendix.

TABLE IV
HARDWARE PLATFORMS USED FOR PERFORMANCE EXPERIMENTS.

System Name	CPU	GPU
a64fx	Fujitsu A64fx	-
brewster	Ampere Altra Q80-30	A100-40G
c6g16xlarge	AWS Graviton 2	-
c7g16xlarge	AWS Graviton 3	-
gorby	AMD 7742 (Rome)	A100-80G
ice4	Intel Xeon 6338 (Ice Lake)	N/A
mi100	AMD 7502 (Rome)	MI100
nuclear	Intel i7-1165G7 (Tiger Lake)	Iris X^e Graphics GeForce RTX 2060
orin	12x Arm Cortex-A78AE	Ampere (not used)
perlmutter	AMD 7713 (Milan)	A100-40G

B. Fortran versus C++

We verified that the translation from C++ to Fortran did not compromise performance using OpenMP on an ARM CPU and using CUDA on an NVIDIA GPU, using the NVHPC 22.7 compilers, which officially support both Neoverse N1 and V1. In Table V, we see that the difference between C++ and Fortran is negligible, except in the case of DOT, where the CUDA Fortran performance is significantly better than CUDA C++ performance. This is because the CUDA C++ version uses a manual reduction, which is not tuned for the architecture in question. In contrast, the CUDA Fortran code relies on a compiler-generated reduction, which can be optimized for each architecture. When we changed the one tuning parameter in BabelStream, we were able to regain most of the performance in A100-80G GPU. For other architectures, this is not necessary, and it is likely that very little effort would be required to adapt BabelStream to support architecture-specific reduction algorithms.

In Table VI, we see that the language impact on performance can be much larger in the case of OpenMP target offload, although this is only true for some compilers. The ROCM 5.1.3 Fortran compiler is approximately 16% slower than the corresponding C++ compiler for four of the five cases, although it is substantially faster for Dot. On the other hand, the other compilers show smaller language effects. For Add and Triad, Cray, Intel and NVHPC compilers show less than 1.1% difference, whereas the differences for Copy, Mul and Dot are larger, but not more than 11%. One possible explanation for some of these differences is that AMD, Cray, and Intel GPU compilers are all based on Clang/LLVM, whereas their Fortran compilers still use proprietary code-bases. Because the effort on the Fortran side is done by each vendor on their own, it

TABLE V
COMPARISON OF FORTRAN VERSUS C++ IMPLEMENTATIONS OF BABELSTREAM, USING THE NVHPC 22.7 COMPILER. THE OPENMP RESULTS WERE OBTAINED ON *brewster*, RUNNING ON ALL 80 CPU CORES OF THE AMPERE ALTRA Q80-30. THE CUDA RESULTS WERE OBTAINED ON *gorby* USING AN A100-80GB. ALL PERFORMANCE NUMBERS REPORTED AS MB/S. THE DIFFERENCE IN BANDWIDTH IS COMPUTED AS $\Delta = B_{Fortran} - B_{C++}/B_{C++}$.

	OpenMP			CUDA		
Function	C++	Fortran	Δ	C++	Fortran	Δ
Copy	156642	157308	+0.42%	1748027	1747360	-0.04%
Mul	159294	157741	-0.98%	1745605	1745195	-0.02%
Add	165328	164461	-0.52%	1772533	1791087	+1.05%
Triad	165713	164382	-0.80%	1773863	1793031	+1.08%
Dot[1]	188901	188504	-0.21%	1555400	1764934	+13.47%
Dot[2]	-	-	-	1743625	1764934	+1.22%

[1] `DOT_NUM_BLOCKS=256`, which is the default.
[2] `DOT_NUM_BLOCKS=1024`.

has take longer for some compilers to mature. We expect these issues to diminish with time, and expect that the multi-vendor effort on the LLVM F18/Flang project [43] will make it easier for vendors to produce high-quality Fortran GPU compilers in the future.

C. GPU experiments

We compared Fortran parallel programming models using a range of GPU hardware and compilers (Figure 2). On NERSC Perlmutter, the A100-40G GPU is supported by Cray and NVHPC compilers, both of which support OpenMP and OpenACC[2]. We used the new Intel compiler (`ifx`) for Tiger Lake X^e integrated graphics, which is the newest and most capable Intel GPU hardware to which we have access. We also measured performance on the AMD MI100 server GPU and on a laptop/desktop-grade NVIDIA GPU (GeForce 2060, TU106), both using the associated vendor compilers.

On A100 systems, we observe that all the programming models deliver a similar fraction of peak performance ($83-93\%$) except in two cases, where the NVHPC compilers produce a less efficient OpenMP reduction in Dot. This issue does not appear in OpenACC or CUDA Fortran kernels, hence should be fixable.

DoConcurrent is $5-7\%$ lower than other models for Copy, Mul, Add and Triad. This can be explained by the use of managed memory in DoConcurrent, but none of the other GPU models. In order to support DoConcurrent on GPUs without language extensions, the NVHPC Fortran compiler uses CUDA managed memory in `ALLOCATE`. For all the other models, directives or data attributes cause the allocation of device memory. This apples-and-oranges comparison is deliberate, because the purpose of standard language parallelism is to avoid such extensions, and thus its idiomatic use will involve managed memory. On the other hand, while OpenMP and OpenACC support managed memory, their idiomatic use involves data directives. We have implemented options both to use device memory in DoConcurrent and to use managed

[2]In the case of OpenACC, the Cray compilers only support this for Fortran.

TABLE VI

COMPARISON OF FORTRAN VERSUS C++ FOR THE OPENMP TARGET IMPLEMENTATIONS OF BABELSTREAM, USING THE ROCM 5.1.3 COMPILERS ON MI100 (*mi100*), NVHPC 22.7 COMPILERS ON A100-80G (*gorby*), INTEL (LLVM-BASED) COMPILERS ON X^e GPU (*nuclear*), AND CRAY COMPILERS ON A100-40G (*perlmutter*). ALL PERFORMANCE NUMBERS REPORTED AS MB/S. THE DIFFERENCE IN BANDWIDTH IS COMPUTED AS $\Delta = ||B_{C++} - B_{Fortran}||/B_{C++}$.

	ROCM/MI100			NVHPC/A100			Intel/X^e iGPU			Cray/A100		
Function	C++	Fortran	Δ	C++	Fortran	Δ	C++	Fortran	Δ	C++	Fortran	Δ
Copy	891793	749290	-15.98%	1716652	1686252	-1.77%	37873	36147	-4.56%	1397392	1353748	-3.12%
Mul	885190	746944	-15.62%	1708711	1680972	-1.62%	38632	38228	-1.05%	1395025	1343306	-3.71%
Add	890688	749645	-15.84%	1770416	1752595	-1.01%	39571	39493	-0.20%	1397944	1396412	-0.11%
Triad	897473	754076	-15.98%	1773396	1754480	-1.07%	39589	39418	-0.43%	1398102	1388211	-0.71%
Dot	279909	744046	+165.82%	841262	787136	-6.43%	29056	25889	-10.90%	1352586	1339465	-0.97%

memory in the other models, to allow apples-to-apples and oranges-to-oranges comparisons, but results with those options are not included. There is value in evaluating idiomatic versus mixed-mode programming in the context of more complex algorithms and applications, as has been done in [44].

On the MI100, the only supported model is OpenMP target, which achieves $62 - 63\%$ of peak, although no model, including HIP, is able to achieve more than 80% of peak on this platform, and it is likely that a tuned implementation will improve the performance of OpenMP target relative to HIP [45]. Finally, on the Intel X^e iGPU platform, we see reasonable behavior with OpenMP Target, although support for reductions and the `target teams loop` construct are disappointing. It should be noted that this Intel X^e iGPU does not support 64-bit floating-point natively, which may impact the reduction performance. However, as BabelStream's data parallel operations are not limited by floating-point throughput, emulation of 64-bit precision is not a performance bottleneck in these cases[3].

D. CPU experiments

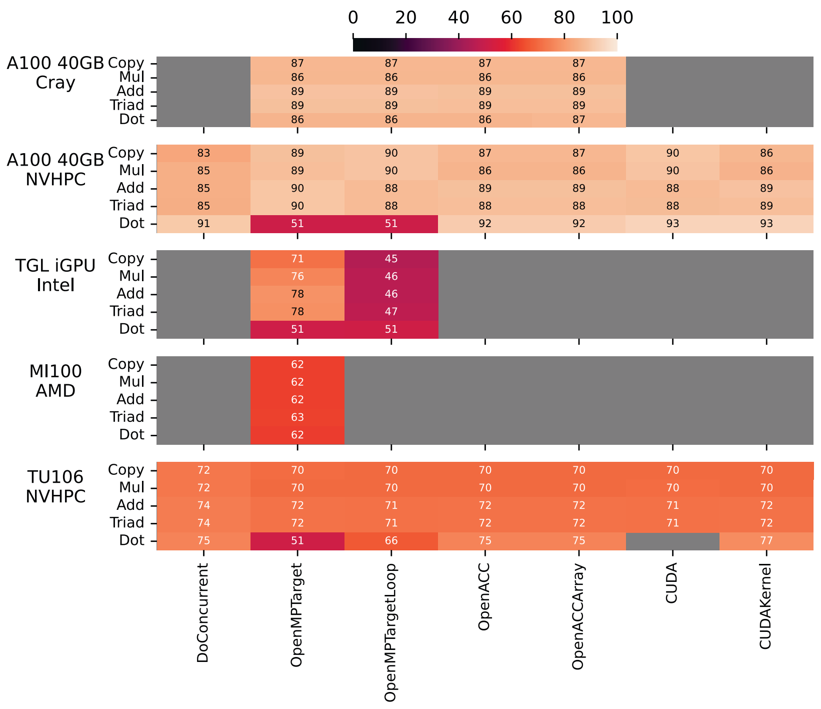
The CPU experiments considered x86 and AArch64 (ARM) processors, including the latest server models from Amazon, AMD, Ampere Computing, Fujitsu and Intel. First, we consider the latest x86 processors, Ice Lake server and Milan (Zen3), both of which have eight channels of DDR4 memory. The Graviton 2 and Graviton 3 platforms have 64 cores and support DDR4 and DDR5, respectively. The latter is the only DDR5 platform considered in this paper, but demonstrates the expected increase in bandwidth from this technology. The Ampere Altra Q80 processor is, like Graviton 2, based on the Arm Neoverse N1 core, but has more cores (80) than the Graviton 2 (64). We have also measured the Ampere Altra Max processor with 128 cores, although since more cores do not increase the achievable memory bandwidth, we do not include these results. The Fujitsu A64FX processor is well-known for its excellent memory bandwidth, and allows us to compare five different Fortran compilers (ARM, Cray, Fujitsu, GCC and NVHPC).

[3]The final version of this paper will use 32-bit floating-point for this platform, once that implementation is validated. Preliminary results suggest the impact is $3 - 6\%$.

In order to deal with NUMA and thread affinity [46], where necessary, we set `OMP_PROC_BIND=close` and `OMP_PLACES=threads`, and restricted the process affinity mask to the first socket, with only one thread per core: ``numactl -m 0 -C `seq -s "," 0 $((${OMP_NUM_THREADS}-1))` ``. On AMD Milan, we used ``numactl -m 0,1,2,3 -C `seq -s "," 0 $((${OMP_NUM_THREADS}-1))` `` to achieve a similar result. Where appropriate, we also used `ACC_NUM_CORES=${OMP_NUM_THREADS}`. All experiments used all of the cores in a single socket, even though this is not always optimal, because real applications are complicated and it is important to users to know the available bandwidth when all of the compute resources are utilized, since this is the most common way that HPC systems are used.

On AMD Milan and Intel Ice Lake server, we find that the Cray, GCC, and NVHPC compilers – and AMD AOCC or Intel (`ifort`), on the relevant platform – produced similar quality results in the cases shown (Figure 3). The percentage of peak is lower on AMD Milan for all models, which we cannot explain, particularly since it can be assumed to be configured correctly in NERSC Perlmutter. We are in the process of evaluating other AMD Milan systems to understand this issue better, and experiments with different affinity settings have not resolved the issue. As is seen everywhere, NVHPC does not generate parallelism with OpenMP `workshare` applied to `DOT_PRODUCT`, and the results are consistent with single-core bandwidth. Traditional OpenMP is the only parallel model supported well by all of the compilers: GCC does not support parallelism in DoConcurrent, NVHPC does not support Taskloop, AMD AOCC Flang does not support Taskloop and does not parallelize DoConcurrent, while Cray and Intel generate incorrect results for Dot with DoConcurrent. The issues with correctness in Cray and Intel follow from the lack of support for either the Fortran 2023 `reduce` locality specifier or pattern-recognition on the pattern, features which are supported by the NVHPC compilers. As Cray and Intel have historically had excellent support for the latest Fortran standards, we expect this issue to be short-lived.

For the ARM processors considered here, the overall trend is that compilers are capable of delivering a high degree of performance across all the programming models, although the



		DoConcurrent	OpenMPTarget	OpenMPTargetLoop	OpenACC	OpenACCArray	CUDA	CUDAKernel
A100 40GB Cray	Copy		87	87	87	87		
	Mul		86	86	86	86		
	Add		89	89	89	89		
	Triad		89	89	89	89		
	Dot		86	86	86	87		
A100 40GB NVHPC	Copy	83	89	90	87	87	90	86
	Mul	85	89	90	86	86	90	86
	Add	85	90	88	89	89	88	89
	Triad	85	90	88	88	88	88	89
	Dot	91	51	51	92	92	93	93
TGL iGPU Intel	Copy		71	45				
	Mul		76	46				
	Add		78	46				
	Triad		78	47				
	Dot		51	51				
MI100 AMD	Copy		62					
	Mul		62					
	Add		62					
	Triad		63					
	Dot		62					
TU106 NVHPC	Copy	72	70	70	70	70	70	70
	Mul	72	70	70	70	70	70	70
	Add	74	72	71	72	72	71	72
	Triad	74	72	71	72	72	71	72
	Dot	75	51	66	75	75		77

Fig. 2. Comparison of Fortran implementations of BabelStream on GPUs. The values reported are the percentage of theoretical peak for each system. Results are omitted due to lack of support or incorrect results.

Fujitsu A64FX shows some poor results in cases that are obviously not supported properly by compilers. Considering just the Neoverse-based CPUs (Graviton and Ampere Altra), we see all but two results are above 75% of peak (Figures 4 and 5). The outliers are the Mul kernel on Graviton 3, which is specific to GCC OpenMP `taskloop`, and NVHPC's OpenMP `workshare` applied to `DOT_PRODUCT`, which is known not to parallelize, in contrast to its OpenACC `kernels` counterpart.

In the case of A64FX (Figure 5), we see that Array expressions, GCC taskloop, and OpenMP `workshare` perform poorly, usually because compilers are not generating parallelism. On the other hand, DoConcurrent does extremely well with the Fujitsu compiler – better than OpenMP – and performs comparably to OpenMP when using NVHPC and CCE. Because OpenMP is known to produce excellent results in other contexts, we believe compiler optimization differences account for the difference between DoConcurrent and OpenMP with the Fujitsu compiler specific to our code, but we have not yet found the root cause. The other major issue is the noticeable gap between the best possible performance of the Fujitsu compiler and that of the other compilers. It is understood that the processor vendor has a unique incentive to optimize for their processors, although the differences seen here are larger than in mainstream server architectures, reminiscent of the situation with Intel Xeon Phi processors.

Finally, the Orin CPU results (Figure 6) show very consistent results across compilers, and where parallelism is supported by the various models, the results there are also consistent. One important note on the percentage of peak is that it is not possible for the CPU to drive all of the 204.8 GB/s LPDDR5 memory bandwidth, which is provisioned because of the integrated Ampere GPU. Unfortunately, we do not have a tighter upper bound for CPU bandwidth than LPDDR5.

V. Conclusions

This paper describes the Fortran implementation of BabelStream using loop parallelism from DoConcurrent, OpenMP (traditional, taskloop, target, target loop), OpenACC and CUDA Fortran. We have implemented kernels using array expressions (or intrinsics, in the case of Dot), in conjunction with OpenMP workshare and OpenACC kernels. The Fortran implementations were shown to be comparable to C++ in the cases of OpenMP on CPU and CUDA, which have a high degree of compiler maturity. Larger performance variation between languages was observed for OpenMP target, which can be attributed to compiler maturity. The existence of at least one compiler where the difference between C++ and Fortran is negligible suggests no fundamental limitations exist due to language semantics.

We saw the greatest degree of performance portability with NVIDIA GPUs, x86 CPUs, and mainstream Neoverse-based ARM CPUs, which is not at all surprising, given that compiler implementation quality follows from widespread usage of

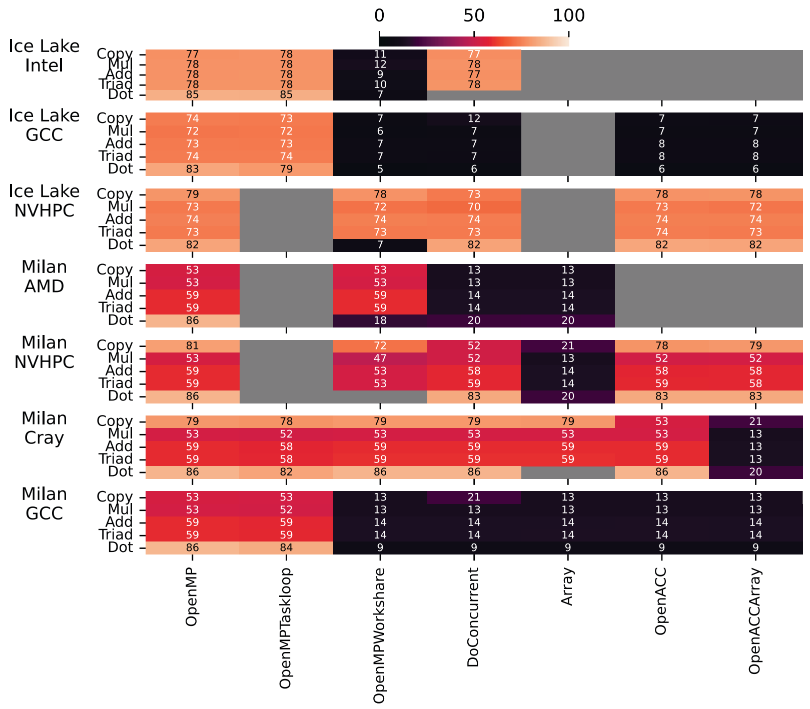


Fig. 3. Comparison of Fortran implementations of BabelStream on x86_64 CPUs. The values reported are the percentage of theoretical peak for each system. Results are omitted due to lack of support or incorrect results. We omitted Array from Ice Lake Xeon because none of the compilers support parallelism, and thus the comparison is uninteresting.

processors. While only one compiler supports all of these platforms (NVHPC), it supports OpenMP, OpenACC and DoConcurrent parallelism, which provides an existence proof that it is possible to do so across a range of architectures.

Compiler support for programming models tracks application use. Traditional OpenMP parallelism on CPUs has been around for decades and is well-supported in all compilers. Newer forms of parallelism, such as OpenMP taskloop and Fortran 2023 extensions to `DO CONCURRENT` are not yet implemented effectively in all compilers, and it is disappointing that, after 14 years, Fortran 2008 `DO CONCURRENT` fails to fulfill its purpose as a portable parallel loop construct supported by all compilers. However, when it is implemented properly with parallelism, we see that it is competitive with directive-based parallelism on all CPUs and NVIDIA GPUs. The only platforms where there is no parallel implementation of DoConcurrent are AMD and Intel GPUs, both of which are relatively new to the HPC space. Thus, there are no technical barriers to the effective implementation of CPU and GPU parallelism in `DO CONCURRENT`, only uninspired compiler developers.

While OpenMP workshare has been around for many years, implementation quality varies significantly due to it remaining a low priority for compiler developers, because of little use in applications, which itself follows from the lack of adoption of array expressions in Fortran codes, often because of poor compiler implementations. Additionally, the `DOT_PRODUCT`, which is almost trivial to implement, and is equivalent to xDOT from the BLAS, is sadly inconsistent from a performance portability perspective, surely due to lack of priority rather than difficulty. This chicken-and-egg problem is common in parallel computing. One potential use of BabelStream Fortran is to allow HPC users and operators to measure the differences between implementations, in order to expose deficiencies in compilers, so that they might be improved.

In the future, we hope to produce – or see others produce – tuned versions of the various implementations for each processor, and to explore in detail how each compiler generates parallel code for each model. At the same time, we hope compiler developers will use BabelStream Fortran as a tool to improve compilers so that such tuning efforts are not necessary, since at least for the patterns considered here, there is not a good reason for compilers to not deliver high-quality results with the idiomatic use of Fortran and directive-based parallelism. We also hope that BabelStream Fortran can be used to demonstrate the utility of DoConcurrent across a range of HPC processors, and motivate compilers currently not supporting this feature with parallelism to do so.

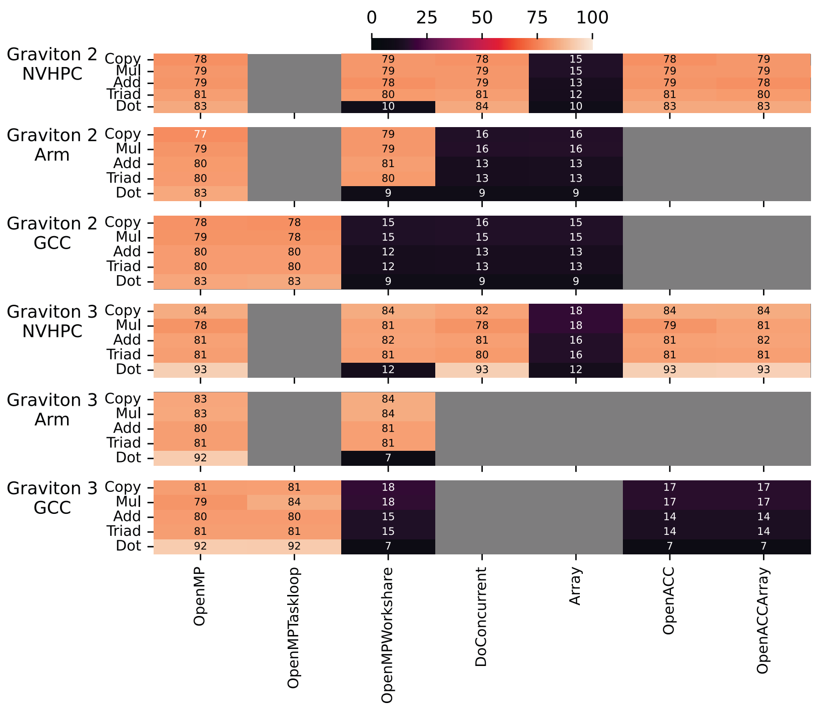


Fig. 4. Comparison of Fortran implementations of BabelStream on Graviton AArch64 CPUs. The values reported are the percentage of theoretical peak for each system. Results are omitted due to lack of support or incorrect results in the compiler.

VI. Acknowledgements

The authors thank Gonzalo Brito for encouragement, Jeff Larkin for help analyzing GPU performance issues, Brent Leback for NVHPC Fortran compiler tips, John Linford for help with AWS Graviton access, and George Markomanolis for helpful feedback on the paper.

This work used the Isambard 2 UK National Tier-2 HPC Service (http://gw4.ac.uk/isambard/) operated by GW4 and the UK Met Office, and funded by EPSRC (EP/T022078/1).

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231 using NERSC award ERCAP0021543.

Experiments in this paper used the NVHPC compiler engineering lab, generously provided and supported by Steve Scalpone and John Mudd.

References

[1] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE computer society technical committee on computer architecture (TCCA) newsletter*, vol. 2, no. 19-25, 1995.

[2] ——, "STREAM: Sustainable Memory Bandwidth in High Performance Computers." [Online]. Available: https://www.cs.virginia.edu/stream/

[3] T. G. Mattson, R. F. Van der Wijngaart, and J. R. Hammond, "Parallel Research Kernels," 2022. [Online]. Available: https://github.com/ParRes/Kernels

[4] R. F. Van der Wijngaart and T. G. Mattson, "The Parallel Research Kernels: A tool for architecture and programming system investigation," in *Proceedings of the IEEE High Performance Extreme Computing Conference*. IEEE, 2014. [Online]. Available: http://dx.doi.org/10.1109/HPEC.2014.7040972

[5] "TIOBE Index." [Online]. Available: https://www.tiobe.com/tiobe-index/

[6] "IEEE Spectrum Top Programming Languages." [Online]. Available: https://spectrum.ieee.org/top-programming-languages/

[7] "Archer2." [Online]. Available: https://www.archer2.ac.uk/

[8] "Archer2 Usage Reports." [Online]. Available: https://www.archer2.ac.uk/support-access/status.html#usage-statistics

[9] "CpuFun: Is Fortran "a dead language"?" [Online]. Available: https://cpufun.substack.com/p/is-fortran-a-dead-language

[10] J. OŚullivan, "25 million lines of Fortran," Jun. 2006. [Online]. Available: https://etrading.wordpress.com/2006/06/01/25-million-lines-of-fortran/

[11] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001257

[12] S. N. Laboratory, "Kokkos C++ performance portability programming ecosystem: The programming model – parallel execution and memory abstraction." [Online]. Available: https://github.com/Kokkos/kokkos

[13] R. D. Hornung and J. A. Keasler, "The RAJA portability layer: Overview and status," U.S. Department of Energy Office of Scientific and Technical Information, Tech. Rep., 9 2014. [Online]. Available: https://doi.org/10.2172/1169830

[14] D. Beckingsale, R. Hornung, T. Scogland, and A. Vargas, "Performance portable C++ programming with RAJA," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '19. New York, NY, USA: ACM, 2019, pp. 455–456. [Online]. Available: http://doi.acm.org/10.1145/3293883.3302577

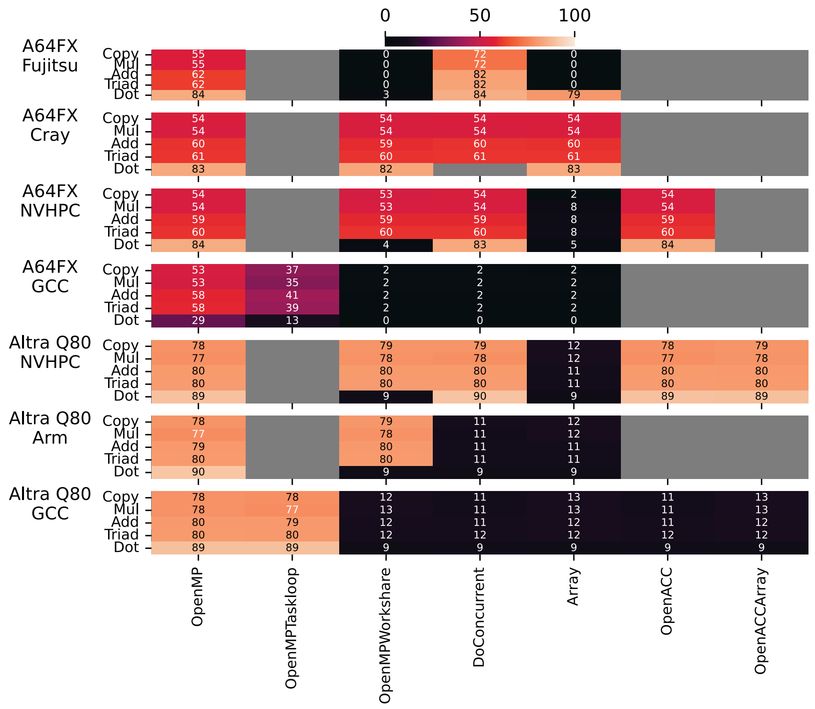


Fig. 5. Comparison of Fortran implementations of BabelStream on A64FX and Q80 AArch64 CPUs. Results are omitted due to lack of support or incorrect results.

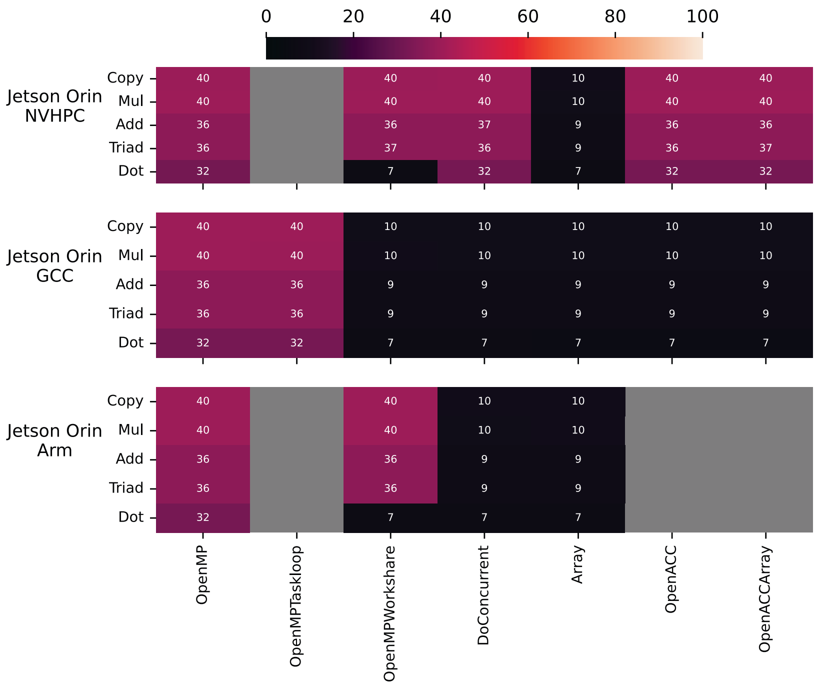


Fig. 6. Comparison of Fortran implementations of BabelStream on Jetson AGX Orin AArch64 CPU. The percentage of peak is relative to a value that is not achievable by the CPU – only the GPU – hence appears low. Results are omitted due to lack of support in compilers.

[15] Lawrence Livermore National Laboratory, "RAJA performance portability layer." [Online]. Available: https://github.com/LLNL/RAJA

[16] Khronos OpenCL Working Group – SYCL subgroup, "SYCL specification," https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf, Feb. 2019.

[17] The Khronos SYCL Working Group, "SYCL 2020 Specification," https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf, May 2022.

[18] T. J. Deakin, J. R. Price, and S. N. Mcintosh-Smith. [Online]. Available: https://github.com/UoB-HPC/BabelStream

[19] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith, "Evaluating attainable memory bandwidth of parallel programming models via babelstream," *International Journal of Computational Science and Engineering*, vol. 17, no. 3, pp. 247–262, 2018. [Online]. Available: https://doi.org/10.1504/IJCSE.2017.10011352

[20] OpenMP Architecture Review Board, "OpenMP aplication program interface – version 5.0," https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf, Nov. 2018.

[21] "The OpenACC Application Programming Interface," Nov. 2021. [Online]. Available: https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.2-final.pdf

[22] "CUDA C++ Programming Guide." [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[23] "HIP: C++ Heterogeneous-Compute Interface for Portability." [Online]. Available: https://github.com/ROCm-Developer-Tools/HIP

[24] Khronos OpenCL Working Group, "The OpenCL specification, version 1.2," Nov. 2012. [Online]. Available: https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf

[25] S. McIntosh-Smith, J. Price, A. Poenaru, and T. Deakin, "Benchmarking the first generation of production quality Arm-based supercomputers," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 20, p. e5569, 2020, e5569 cpe.5569. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5569

[26] A. Poenaru, T. Deakin, S. McIntosh-Smith, S. D. Hammond, and A. J. Younge, "An evaluation of the Fujitsu A64FX for HPC applications," in *Presentation in AHUG ISC 21 Workshop*, 2021.

[27] T. Deakin and S. McIntosh-Smith, "Evaluating the performance of HPC-style SYCL applications," in *Proceedings of the International Workshop on OpenCL*, 2020, pp. 1–11.

[28] R. F. Van der Wijngaart, A. Kayi, J. R. Hammond, G. Jost, T. St. John, S. Sridharan, T. G. Mattson, J. Abercrombie, and J. Nelson, "Comparing runtime systems with exascale ambitions using the parallel research kernels," in *High Performance Computing*, J. M. Kunkel, P. Balaji, and J. Dongarra, Eds. Cham: Springer International Publishing, 2016, pp. 321–339.

[29] A. Fanfarillo and J. Hammond, "Caf events implementation using mpi-3 capabilities," in *Proceedings of the 23rd European MPI Users' Group Meeting*, ser. EuroMPI 2016. New York, NY, USA: ACM, 2016, pp. 198–207. [Online]. Available: http://doi.acm.org/10.1145/2966884.2966916

[30] E. Kayraklioglu, W. Chang, and T. El-Ghazawi, "Comparative performance and optimization of chapel in modern manycore architectures," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2017, pp. 1105–1114.

[31] D. Bailey, T. Harris, W. Saphir, R. Van Der Wijngaart, A. Woo, and M. Yarrow, "The NAS parallel benchmarks 2.0," Technical Report NAS-95-020, NASA Ames Research Center, Tech. Rep., 1995.

[32] P. R. Luszczek, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, and D. Takahashi, "The HPC challenge (HPCC) benchmark suite," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, vol. 213, no. 10.1145, 2006, pp. 1 188 455–1 188 677.

[33] J. Dongarra and P. Luszczek, "HPC challenge: design, history, and implementation highlights," in *Contemporary High Performance Computing*. Chapman and Hall/CRC, 2017, pp. 13–30.

[34] R. Xu, X. Tian, S. Chandrasekaran, Y. Yan, and B. Chapman, "OpenACC parallelization and optimization of NAS parallel benchmarks," in *Proc. GPU Technol. Conf.*, 2014, pp. 1–27.

[35] S. Saini, R. Ciotti, B. T. Gunney, T. E. Spelce, A. Koniges, D. Dossa, P. Adamidis, R. Rabenseifner, S. R. Tiyyagura, and M. Mueller, "Performance evaluation of supercomputers using HPCC and IMB benchmarks," *Journal of Computer and System Sciences*, vol. 74, no. 6, pp. 965–982, 2008.

[36] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (SHOC) benchmark suite," in *Proceedings of the 3rd workshop on general-purpose computation on graphics processing units*, 2010, pp. 63–74.

[37] Lawrence Livermore National Laboratory, "RAJA performance portability layer." [Online]. Available: https://github.com/LLNL/RAJAPerf

[38] R. D. Hornung and H. E. Hones, "Raja performance suite," U.S. Department of Energy Office of Scientific and Technical Information, Tech. Rep., 9 2017. [Online]. Available: https://www.osti.gov//servlets/purl/1394927

[39] Standard Performance Evaluation Consortium, "Spechpc." [Online]. Available: https://www.spec.org/hpg/hpc2021/

[40] J. R. Hammond. [Online]. Available: https://github.com/jeffhammond/BabelStream/tree/fortran-ports

[41] J. Hammond, "Bug 101602 - [F2018] local and local_init are not supported in DO CONCURRENT," Jul. 2021. [Online]. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=101602

[42] L. S. Karumbunathan, "NVIDIA Jetson AGX Orin Series," Mar. 2022. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/gtcf21/jetson-orin/nvidia-jetson-agx-orin-technical-brief.pdf

[43] "Welcome to Flang's documentation." [Online]. Available: https://flang.llvm.org/docs/

[44] M. M. Stulajter, R. M. Caplan, and J. A. Linker, "Can Fortran's 'do concurrent' replace directives for accelerated computing?" in *International Workshop on Accelerator Programming Using Directives*. Springer, 2022, pp. 3–21. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-97759-7_1

[45] G. S. Markomanolis, A. Alpay, J. Young, M. Klemm, N. Malaya, A. Esposito, J. Heikonen, S. Bastrakov, A. Debus, T. Kluge *et al.*, "Evaluating GPU programming models for the LUMI supercomputer," in *Asian Conference on Supercomputing Frontiers*. Springer, Cham, 2022, pp. 79–101.

[46] G. Lockwood, Oct. 2019. [Online]. Available: https://www.glennklockwood.com/hpc-howtos/process-affinity.html

[47] "Intel Core i7-1165G7 Processor." [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/208921/intel-core-i71165g7-processor-12m-cache-up-to-4-70-ghz-with-ipu.html

[48] "Phantom Canyon." [Online]. Available: https://simplynuc.com/phantom-canyon/

[49] NERSC, "Perlmutter." [Online]. Available: https://docs.nersc.gov/systems/perlmutter/system_details/

[50] "NVIDIA DGX A100 System Architecture," Jul. 2020. [Online]. Available: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/dgx-a100/dgxa100-system-architecture-white-paper.pdf

[51] "NVIDIA Arm HPC Developer Kit." [Online]. Available: https://developer.nvidia.com/arm-hpc-devkit

[52] "AWS Graviton Processor." [Online]. Available: https://aws.amazon.com/ec2/graviton/

[53] Arm Limited, "Arm Neoverse V1 reference design," 2019. [Online]. Available: https://documentation-service.arm.com/static/5f05e93dcafe527e86f61acd

[54] ——, "Arm Neoverse V1 reference design," 2021. [Online]. Available: https://documentation-service.arm.com/static/608815225e70d934bc69f25a

[55] "AMD Instinct MI100 Accelerator." [Online]. Available: https://www.amd.com/en/products/server-accelerators/instinct-mi100

[56] "Intel Xeon Gold 6338 Processor." [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/212285/intel-xeon-gold-6338-processor-48m-cache-2-00-ghz/specifications.html

[57] "FUJITSU Supercomputer PRIMEHPC FX700," Jul. 2021. [Online]. Available: https://www.fujitsu.com/downloads/SUPER/primehpc-fx700-datasheet-en-202107.pdf

VII. Appendix: Experimental Details

The following contains details of the hardware and software used for experiments, source code patterns, and the raw data used to generate the figures.

A. Platform details

TABLE VII
Hardware platforms used for performance experiments.

System Name	CPU	GPU	Links
orin	12x Arm Cortex-A78AE	integrated (Ampere, cc87)	[42]
nuclear	Intel Core i7-1165G7 (Tiger Lake)	Iris Xe Graphics (device=0x9a49) GeForce RTX 2060 (Turing, cc75)	[47], [48]
perlmutter	AMD 7713 (Milan)	4x A100-40G	[49]
gorby	AMD 7742 (Rome)	4x A100-80G	[50]
brewster	Ampere Altra Q80-30	2x A100-40G	[51]
c6g16xlarge	AWS Graviton 2	-	[52], [53]
c7g16xlarge	AWS Graviton 3	-	[52], [54]
mi100	AMD 7502 (Rome)	MI100	[55]
ice4	Intel Xeon 6338 (Ice Lake)	N/A	[56]
a64fx	Fujitsu A64fx	-	[57]

TABLE VIII
Operating system details for performance experiments.

System Name	Operating System	Linux kernel
orin	Ubuntu 20.04.4 LTS	5.10.65-tegra
nuclear	Ubuntu 22.04.1 LTS	5.15.0-46-generic
perlmutter	Cray Linux (based on SUSE Linux Enterprise Server 15 SP3)	5.3.18-150300.59.43_11.0.51-cray_shasta_c
gorby	Ubuntu 20.04.4 LTS	5.4.0-122-generic
brewster	Ubuntu 20.04.4 LTS	5.4.0-122-generic
c6g16xlarge	Amazon Linux 2	5.10.112-108.499.amzn2.aarch64
c7g16xlarge	Amazon Linux 2	5.10.112-108.499.amzn2.aarch64
mi100	Ubuntu 20.04.4 LTS	5.13.0-35-generic
ice4	Rocky Linux 8.5	4.18.0-348.20.1.el8_5.x86_64
a64fx	Red Hat Enterprise Linux 8.2	4.18.0-193.el8.aarch64

TABLE IX
COMPILERS, GENERIC FLAGS AND ARCHITECTURE-SPECIFIC COMPILER FLAGS FOR EACH PLATFORM.

System Name	Compilers	Compiler Invocation
orin	ARM 22.0.2	`armflang -std=f2018 -O3 -mcpu=cortex-a78`
	GCC 12.2.0	`gfortran -std=f2018 -O3 -mcpu=cortex-a78ae`
	NVHPC 22.7	`nvfortran -O3 -tp=neoverse-n1 -gpu=ccn87,cc86`
nuclear	Intel 2021.6	`ifx -std18 -Ofast -xHOST -fopenmp-targets=spir64`
	NVHPC 22.7	`nvfortran -O3 -tp=host -gpu=cc75`
perlmutter	AMD AOCC 3.2.0	`flang -std=f2018 -O3 -march=znver3`
	Cray 14.0.1	`ftn -e F -O3`
	GCC	`gfortran -std=f2018 -O3 -march=znver3`
	NVHPC 22.7	`nvfortran -O3 -tp=zen3 -gpu=cc80`
gorby	NVHPC 22.7	`nvfortran -O3 -tp=host -gpu=cc80`
brewster	NVHPC 22.7	`nvfortran -O3 -tp=neoverse-n1`
c6g16xlarge	ARM 22.0.2	`armflang -std=f2018 -O3 -mcpu=neoverse-n1`
	GCC 12.2.0	`gfortran -std=f2018 -O3 -mcpu=neoverse-n1`
	NVHPC 22.7	`nvfortran -O3 -tp=neoverse-n1`
c7g16xlarge	ARM 22.0.2	`armflang -std=f2018 -O3 -mcpu=neoverse-v1`
	GCC 12.1.0	`gfortran -std=f2018 -O3 -mcpu=neoverse-v1`
	NVHPC 22.7	`nvfortran -O3 -tp=neoverse-v1`
mi100	AMD ROCM 5.1.3	`flang -std=f2018 -O3 -fopenmp-targets=amdgcn-amd-amdhsa -Xopenmp-target=amdgcn-amd-amdhsa -march=gfx908`
ice4	GCC 12.2.0	`gfortran -std=f2018 -O3 -march=icelake-server`
	Intel 2021.6	`ifx -std18 -Ofast -xHOST -qopt-zmm-usage=low -qopt-streaming-stores=always`
	NVHPC 22.7	`nvfortran -O3 -tp=host`
a64fx	Cray 10.0.3	`ftn -e F -O3`
	Fujitsu 4.3.1	`frt -X08 -Kfast -KA64FX -KSVE -KARMV8_3_A -Kzfill=100 -Kprefetch_sequential=soft -Kprefetch_sequential=soft -Kprefetch_line=8 -Kprefetch_line_L2=16 -Koptmsg=2 -Keval`
	GCC 11.1.0	`gfortran -std=f2018 -O3`
	NVHPC 22.7	`nvfortran -O3 -tp=host`

TABLE X
COMPILER FLAGS FOR THE DIFFERENT PROGRAMMING MODELS. A DASH INDICATES THE MODEL IS SUPPORTED BUT NO SPECIAL FLAG EXISTS, OFTEN BECAUSE THE COMPILER DOES NOT GENERATE PARALLELISM FOR THIS FEATURE. THE NOTATION N/A IMPLIES THE FEATURE IS NOT SUPPORTED.

Compiler	DoConcurrent	Array	OpenMP	OpenACC	CUDA
ARM	-	-	`-fopenmp`	N/A	N/A
AMD	-	-	`-fopenmp`	N/A	N/A
Cray CPU	`-h thread_do_concurrent`	`-h autothread`	`-h omp`	`-h acc -h omp`	N/A
Cray GPU	N/A	N/A	`-h omp`	`-h acc`	N/A
Fujitsu	`-Kparallel,reduction`	`-Kparallel,reduction`	`-fopenmp`	N/A	N/A
GCC	-	-	`-fopenmp`	`-fopenacc`	N/A
Intel ifort (CPU)	`-parallel`	`-parallel`	`-qopenmp`	N/A	N/A
Intel ifx (GPU)	-	-	`-fopenmp`	N/A	N/A
NVHPC CPU	`-stdpar=multicore`	-	`-mp=multicore`	`-acc=multicore`	N/A
NVHPC GPU	`-stdpar=gpu`	-	`-mp=gpu`	`-acc=gpu`	`-cuda`

B. Absolute performance data

TABLE XI
Comparison of Fortran implementations of BabelStream on GPUs. All measurements are reported as MBytes/sec. Target and TargetLoop refer to the OpenMP implementations of those constructs. Results are omitted due to lack of support, incorrect results, or poor parallel performance - see text for details.

	perlmutter (A100-40GB GPU, Cray 14.0.1), size=2^{29}						
Function	DoConcurrent	OpenMPTarget	TargetLoop	OpenACC	OpenACCArray	CUDA	CUDAKernel
Copy	-	1354213	1353948	1353999	1352954	-	-
Mul	-	1343710	1343582	1343792	1342505	-	-
Add	-	1388589	1384966	1384893	1384464	-	-
Triad	-	1381507	1379137	1378184	1377497	-	-
Dot	-	1341748	1341620	1341352	1353724	-	-
	perlmutter (A100-40GB GPU, NVHPC 22.7) size=2^{29}						
Function	DoConcurrent	Target	TargetLoop	OpenACC	OpenACCArray	CUDA	CUDAKernel
Copy	1284207	1379466	1395603	1349346	1349558	1397669	1344509
Mul	1319316	1376791	1392910	1339665	1339457	1395829	1333654
Add	1321800	1397374	1363770	1382945	1382634	1365519	1387712
Triad	1317192	1397950	1364045	1373642	1373218	1366968	1376298
Dot	1409804	788516	788863	1422881	1425006	-	1453261
	mi100 (MI-100 GPU, ROCM 5.1.3), size=2^{29}						
Function	DoConcurrent	Target	TargetLoop	OpenACC	OpenACCArray	CUDA	CUDAKernel
Copy	-	749290	-	-	-	-	-
Mul	-	746944	-	-	-	-	-
Add	-	749645	-	-	-	-	-
Triad	-	754076	-	-	-	-	-
Dot	-	744046	-	-	-	-	-
	nuclear (TU106 GPU, NVHPC 22.7), size=2^{26}						
Function	DoConcurrent	Target	TargetLoop	OpenACC	OpenACCArray	CUDA	CUDAKernel
Copy	243429	235423	234549	234084	234089	234503	234135
Mul	243093	234441	234247	234493	233737	235016	234646
Add	247943	240465	239892	240397	240390	239964	240465
Triad	247977	240354	239928	240501	240465	240028	240501
Dot	252123	172242	223282	252236	252645	-	258428
	nuclear (Xe, ifx 2022.1.0), size=2^{26}						
Function	DoConcurrent	Target	TargetLoop	OpenACC	OpenACCArray	CUDA	CUDAKernel
Copy	-	36522	22910	-	-	-	-
Mul	-	38671	23618	-	-	-	-
Add	-	39833	23758	-	-	-	-
Triad	-	39794	24106	-	-	-	-
Dot	-	26149	26329	-	-	-	-

TABLE XII
Comparison of Fortran implementations of BabelStream on Ice Lake Xeon. All measurements are reported as MBytes/sec. Taskloop and Workshare refer to the OpenMP implementations of those constructs. Results are omitted due to lack of support, incorrect results, or poor parallel performance - see text for details.

	ice4 (Ice Lake Xeon, ifort 2021.6.0), size=2^{30}					
Function	DoConcurrent	OpenMP	Taskloop	Workshare	OpenACC	OpenACCArray
Copy	158726	158584	158816	22277	-	-
Mul	160015	160154	160085	24920	-	-
Add	160050	159792	159901	18509	-	-
Triad	160041	160170	160008	20388	-	-
Dot	-	172085	171830	14981	-	-
	ice4 (Ice Lake Xeon, NVHPC 22.7), size=2^{30}					
Function	DoConcurrent	OpenMP	Taskloop	Workshare	OpenACC	OpenACCArray
Copy	147077	159344	-	158937	157916	156304
Mul	145289	148722	-	148858	148444	148552
Add	150576	151819	-	151514	150535	150188
Triad	148657	150302	-	149340	149458	148855
Dot	167362	170664	-	-	167305	167220
	ice4 (Ice Lake Xeon, GCC 12.2.0), size=2^{30}					
Function	DoConcurrent	OpenMP	Taskloop	Workshare	OpenACC	OpenACCArray
Copy	23747	148693	148730	13869	14874	15206
Mul	13912	148556	148292	13166	14055	14372
Add	15143	150834	150836	14404	15366	15897
Triad	15213	150866	150846	14455	15427	15932
Dot	11399	171174	164161	9922	11397	11437

TABLE XIII
COMPARISON OF FORTRAN IMPLEMENTATIONS OF BABELSTREAM ON AMD MILAN. ALL MEASUREMENTS ARE REPORTED AS MBYTES/SEC. TASKLOOP AND WORKSHARE REFER TO THE OPENMP IMPLEMENTATIONS OF THOSE CONSTRUCTS. RESULTS ARE OMITTED DUE TO LACK OF SUPPORT, INCORRECT RESULTS, OR POOR PARALLEL PERFORMANCE - SEE TEXT FOR DETAILS.

	perlmutter (AMD 7713 CPU, AOCC 3.2.0), size=2^{30}						
Function	DoConcurrent	Array	OpenMP	Taskloop	Workshare	OpenACC	OpenACCArray
Copy	25993	25673	108665	-	108843	-	-
Mul	25869	25866	107814	-	107818	-	-
Add	28605	28515	121244	-	121249	-	-
Triad	28716	28656	121021	-	121063	-	-
Dot	40374	40819	176325	-	36958	-	-
	perlmutter (AMD 7713 CPU, Cray 14.0.1), size=2^{30}						
Function	DoConcurrent	Array	OpenMP	Taskloop	Workshare	OpenACC	OpenACCArray
Copy	162340	162465	162481	160820	162417	108382	43242
Mul	107960	107845	107851	106562	107877	107892	25677
Add	121357	121329	121353	119482	121264	121634	27366
Triad	121799	121635	121772	119608	121670	122005	27605
Dot	-	176671	176682	170642	176156	176473	40439
	perlmutter (AMD 7713 CPU, NVHPC 22.7), size=2^{30}						
Function	DoConcurrent	Array	OpenMP	Taskloop	Workshare	OpenACC	OpenACCArray
Copy	103284	43300	148345	-	146799	158267	149429
Mul	102715	26168	97463	-	96568	105672	104267
Add	115718	28475	108872	-	107633	118349	114034
Triad	115648	28546	108763	-	107543	118492	110279
Dot	167126	40238	157793	-	-	171439	156921
	perlmutter (AMD 7713 CPU, GCC), size=2^{30}						
Function	DoConcurrent	Array	OpenMP	Taskloop	Workshare	OpenACC	OpenACCArray
Copy	43181	26270	108603	107670	26238	26009	26228
Mul	26084	26132	107720	107005	26072	25982	26118
Add	28584	28690	121292	119977	28514	28858	28569
Triad	28730	28814	121408	120285	28652	28894	28726
Dot	18154	18196	176821	172544	18238	18134	18261

TABLE XIV
COMPARISON OF FORTRAN IMPLEMENTATIONS OF BABELSTREAM ON GRAVITON 2. ALL MEASUREMENTS ARE REPORTED AS MBYTES/SEC. TASKLOOP AND WORKSHARE REFER TO THE OPENMP IMPLEMENTATIONS OF THOSE CONSTRUCTS.

	c6g16xlarge (Graviton 2 CPU, GCC 12.1), size=2^{30}					
Function	DoConcurrent	OpenMP	Taskloop	Workshare	OpenACC	OpenACCArray
Copy	-	159917	159325	-	-	-
Mul	-	160698	159946	-	-	-
Add	-	163456	163218	-	-	-
Triad	-	164101	163564	-	-	-
Dot	-	168776	168798	-	-	-
	c6g16xlarge (Graviton 2 CPU, NVHPC 22.7), size=2^{30}					
Function	DoConcurrent	OpenMP	Taskloop	Workshare	OpenACC	OpenACCArray
Copy	158661	158312	-	161241	158208	161078
Mul	161381	161357	-	161597	161275	161385
Add	160916	160785	-	158432	160748	158314
Triad	164427	164253	-	162806	164287	162624
Dot	170501	170248	-	-	170148	170104
	c6g16xlarge (Graviton 2 CPU, ARM 22.0.2), size=2^{30}					
Function	DoConcurrent	OpenMP	Taskloop	Workshare	OpenACC	OpenACCArray
Copy	-	156060	-	161025	-	-
Mul	-	160290	-	161537	-	-
Add	-	163088	-	164277	-	-
Triad	-	163709	-	162436	-	-
Dot	-	169757	-	-	-	-

TABLE XV

COMPARISON OF FORTRAN IMPLEMENTATIONS OF BABELSTREAM ON GRAVITON 3. ALL MEASUREMENTS ARE REPORTED AS MBYTES/SEC. TASKLOOP AND WORKSHARE REFER TO THE OPENMP IMPLEMENTATIONS OF THOSE CONSTRUCTS. RESULTS ARE OMITTED DUE TO LACK OF SUPPORT, INCORRECT RESULTS, OR POOR PARALLEL PERFORMANCE - SEE TEXT FOR DETAILS.

	c7g16xlarge (Graviton 3 CPU, GCC 12.1), size=2^{30}					
Function	DoConcurrent	OpenMP	Taskloop	Workshare	OpenACC	OpenACCArray
Copy	-	241906	241489	-	-	-
Mul	-	235122	201549	-	-	-
Add	-	240041	243435	-	-	-
Triad	-	241035	241832	-	-	-
Dot	-	276443	274416	-	-	-
	c7g16xlarge (Graviton 3 CPU, NVHPC 22.7), size=2^{30}					
Function	DoConcurrent	OpenMP	Taskloop	Workshare	OpenACC	OpenACCArray
Copy	245121	252112	-	251472	252159	251510
Mul	234001	234208	-	244085	234372	244042
Add	242690	244289	-	245778	244319	245679
Triad	239632	242101	-	243518	242152	243490
Dot	277772	277852	-	35429	277865	279471
	c7g16xlarge (Graviton 3 CPU, ARM 22.0.2), size=2^{30}					
Function	DoConcurrent	OpenMP	Taskloop	Workshare	OpenACC	OpenACCArray
Copy	-	248365	-	252018	-	-
Mul	-	248926	-	251205	-	-
Add	-	244356	-	245434	-	-
Triad	-	244664	-	245310	-	-
Dot	-	275831	-	-	-	-

TABLE XVI

COMPARISON OF FORTRAN IMPLEMENTATIONS OF BABELSTREAM ON OTHER AARCH64 CPUS. ALL MEASUREMENTS ARE REPORTED AS MBYTES/SEC. TASKLOOP AND WORKSHARE REFER TO THE OPENMP IMPLEMENTATIONS OF THOSE CONSTRUCTS. RESULTS ARE OMITTED DUE TO LACK OF SUPPORT, INCORRECT RESULTS, OR POOR PARALLEL PERFORMANCE - SEE TEXT FOR DETAILS.

	brewster (Ampere Altra Q80 CPU, NVHPC 22.7), size=2^{30}					
Function	DoConcurrent	OpenMP	Taskloop	Workshare	OpenACC	OpenACCArray
Copy	158014	157270	-	158531	157132	158767
Mul	157890	157862	-	159744	157479	160330
Add	164272	164130	-	163794	164210	164085
Triad	164258	164076	-	163880	164121	163939
Dot	188742	188229	-	-	188168	188595
	orin (A78, NVHPC 22.7), size=2^{27}					
Function	DoConcurrent	OpenMP	OpenMPTaskloop	OpenMPWorkshare	OpenACC	OpenACCArray
Copy	81762	81510	-	81535	81591	81477
Mul	81778	81771	-	81756	81799	81691
Add	74778	74667	-	74629	74678	74690
Triad	74726	74711	-	74758	74742	74756
Dot	64878	64785	-	13696	64802	64836
	orin (A78, GCC 12.2.0), size=2^{27}					
Function	DoConcurrent	OpenMP	OpenMPTaskloop	OpenMPWorkshare	OpenACC	OpenACCArray
Copy	20668	81836	81756	20788	20609	20670
Mul	20567	81734	81584	20835	20565	20631
Add	17594	74681	74559	17667	17559	17527
Triad	17518	74573	74554	17666	17505	17506
Dot	13602	64838	64926	13682	13597	13592
	orin (A78, ARM 22.0.2), size=2^{27}					
Function	DoConcurrent	OpenMP	OpenMPTaskloop	OpenMPWorkshare	OpenACC	OpenACCArray
Copy	20842	81822	-	81902	-	-
Mul	20783	81716	-	81809	-	-
Add	17703	74486	-	74441	-	-
Triad	17663	74465	-	74498	-	-
Dot	13700	64963	-	13615	-	-

C. Source code listings

Listing 1. Fortran `DO CONCURRENT` implementation. KERNEL is the loop body shown in Table I.

```
do concurrent (i=1:N)
    <Kernel>
end do
```

TABLE XVII

COMPARISON OF FORTRAN IMPLEMENTATIONS OF BABELSTREAM ON A64FX. ALL MEASUREMENTS ARE REPORTED AS MBYTES/SEC. TASKLOOP AND WORKSHARE REFER TO THE OPENMP IMPLEMENTATIONS OF THOSE CONSTRUCTS. RESULTS ARE OMITTED DUE TO LACK OF SUPPORT, INCORRECT RESULTS, OR POOR PARALLEL PERFORMANCE - SEE TEXT FOR DETAILS.

	isambard (A64FX, NVHPC 22.7), size=2^{27}					
Function	DoConcurrent	OpenMP	OpenMPTaskloop	OpenMPWorkshare	OpenACC	OpenACCArray
Copy	553761	549566	-	541255	551174	
Mul	550355	551174	-	547548	550976	
Add	602571	605266	-	603000	605266	
Triad	614011	616809	-	613777	617093	
Dot	851500	858170	-	39772	855571	
	isambard (A64FX, Cray 10.0.1), size=2^{27}					
Function	DoConcurrent	OpenMP	OpenMPTaskloop	OpenMPWorkshare	OpenACC	OpenACCArray
Copy	557122	556950	-	553512		
Mul	554900	550073	-	552280		
Add	612309	612689	-	606180		
Triad	621490	622084	-	618272		
Dot	-	849565	-	844300		
	isambard (A64FX, Fujitsu), size=2^{27}					
Function	DoConcurrent	OpenMP	OpenMPTaskloop	OpenMPWorkshare	OpenACC	OpenACCArray
Copy	734442	566918	-	1071	-	
Mul	734442	561266	-	1159	-	
Add	835754	632114	-	1718	-	
Triad	838347	633358	-	1717	-	
Dot	864580	859794	-	33996	-	
	isambard (A64FX, GCC), size=2^{27}					
Function	DoConcurrent	OpenMP	OpenMPTaskloop	OpenMPWorkshare	OpenACC	OpenACCArray
Copy	16213	547124	380854	19276		
Mul	19260	543230	357814	18330		
Add	23231	593454	415643	22207		
Triad	23236	596978	399952	22244		
Dot	3130	294985	129675	3104		

Listing 2. Template for parallel loop implementations. `<Kernel>` is the loop body shown in Table I. `<Parallel Directive>` is one of the options listed in Table II.

```
<Parallel Directive>
do i=1,N
    <Kernel>
end do
```

Listing 3. Representative implementation of the CUDA Fortran implementations with explicit kernels (as opposed to the `kernels` directive). The source code has been modified in trivial ways for display purposes.

```
attributes(global) &
subroutine do_nstream(n,s,A,B,C)
 implicit none
 integer, intent(in), value :: n
 real(kind=REAL64), intent(in), value :: s
 real(kind=REAL64), intent(inout) :: A(n)
 real(kind=REAL64), intent(in) :: B(n)
 real(kind=REAL64), intent(in) :: C(n)
 integer :: i
 i = blockDim%x * (blockIdx%x-1) &
   + threadIdx%x
 if (i <= N) then
   A(i) = A(i) + B(i) + s * C(i)
 endif
end subroutine do_nstream

subroutine nstream(ss)
 use cudafor, only: cudaDeviceSynchronize
 implicit none
 real(kind=REAL64), intent(in) :: ss
```

```
  real(kind=REAL64) :: s
  integer :: err
  s = ss
  call do_nstream<<<gs,bs>>>(n,s,A,B,C)
  err = cudaDeviceSynchronize()
  ! error handling
end subroutine nstream
```