Deakin, T., Price, J., Martineau, M., & McIntosh-Smith, S. (2018). Evaluating attainable memory bandwidth of parallel programming models via BabelStream. *International Journal of Computational Science and Engineering*, *17*(3), 247-262. https://doi.org/10.1504/IJCSE.2017.10011352

Evaluating attainable memory bandwidth of parallel programming models via BabelStream

Tom Deakin
James Price
Matt Martineau
Simon McIntosh-Smith

Department of Computer Science,
University of Bristol,
Bristol, UK
E-mail: tom.deakin@bristol.ac.uk

Abstract: Many scientific codes consist of memory bandwidth bound kernels — the dominating factor of the runtime is the speed at which data can be loaded from memory into the Arithmetic Logic Units, before results are written back to memory. One major advantage of many-core devices such as General Purpose Graphics Processing Units (GPGPUs) and the Intel Xeon Phi is their focus on providing increased memory bandwidth over traditional CPU architectures. However, as with CPUs, this peak memory bandwidth is usually unachievable in practice and so benchmarks are required to measure a practical upper bound on expected performance.

We augment the standard set of STREAM kernels with a dot product kernel to investigate the performance of simple reduction operations on large arrays. Such kernels are usually present in scientific codes and are still memory bandwidth bound.

The choice of one programming model over another should ideally not limit the performance that can be achieved on a device. BabelStream (formally GPU-STREAM) has been updated to incorporate a wide variety of the latest parallel programming models, all implementing the same parallel scheme. As such this tool can be used as a kind of *Rosetta Stone* which provides both a cross-platform and cross-programming model array of results of achievable memory bandwidth.

Keywords: Performance portability; Many-core; Parallel programming models; Memory bandwidth benchmark.

Biographical notes: Tom Deakin is studying for a PhD at the University of Bristol sponsored by AWE. He is focusing on applications of High Performance Computing, in particular using accelerated many-core devices. In 2012 Tom graduated valedictorian from the University of Bristol with first class honours with a MSci in Mathematics and Computer Science, winning the prize for Best graduating student in Computer Science and Mathematics.

James Price is working towards a PhD in Computer Science in the High Performance Computing group at the University of Bristol. His work involves investigating ways to improve the programmability of modern, many-core computer processors, with a particular emphasis on achieving performance portability across wide ranges of CPU and accelerator architectures. James is an active contributor to the OpenCL parallel programming standard.

Matt Martineau is currently undertaking a PhD in Computer Science at the University of Bristol, supported by the Atomic Weapons Establishment. At present he is focusing on cutting edge technologies and practices for High Performance Computing, with an aim to understand how optimisations proven in micro-benchmarks can be scaled up to real-world scientific codes.

Simon McIntosh-Smith is a Professor of High Performance Computing at the University of Bristol in the UK. His background includes 15 years in industry designing microprocessors at companies including Inmos, STMicroelectronics, Pixelfusion and ClearSpeed. He joined the University of Bristol in 2009 where his research focuses on performance portability and fault tolerant software techniques to reach Exascale. McIntosh-Smith actively contributes to the OpenMP and OpenCL parallel programming standards.

1 Introduction

The number of programming models for parallel programming has grown rapidly in recent years. Given that they in general aim to both achieve high performance and run across a range of hardware (i.e. are portable), the programmer may hope they are abstract enough that they enable some degree of *performance portability*. In principle therefore, one might expect that, when writing or porting a new code, the choice of parallel programming language should largely be a matter of preference. In reality there are often significant differences between the results delivered by different parallel programming models, and thus benchmarks play an important role in objectively comparing across not just different hardware, but also the programming models. This study aims to explore this space and highlight these differences.

Many scientific codes are memory bandwidth bound, and thus are commonly compared against the STREAM benchmark, itself a simple achievable memory bandwidth measure [14]. The simplicity of the STREAM kernels allow us to investigate if memory bandwidth limits are achievable irrespective of the programming model chosen. The focus therefore is to evaluate realistic performance expectations of the portable programming models, for if STREAM does not perform well it is unlikely that a large scientific code written in this way will perform well. STREAM itself was written with similar goals in mind, namely to represent the achievable performance of a scientific code by modelling typical computation kernels and not as a highly optimised micro-benchmark to measure optimal performance [16]. As with STREAM, we also focus only on single-node performance and do not investigate multi-node memory bandwidth characteristics which may form the subject of a future paper. In this work we implemented the STREAM kernels in a wide variety of parallel programming models and across a diverse range of CPU and GPU devices, comparing the percentage of theoretical peak that was achieved.

Specifically, we make the following contributions:

1. We port the STREAM memory bandwidth benchmark to seven parallel programming models, all of which support many-core processors: Kokkos, RAJA, OpenMP 4.x, OpenACC, SYCL, OpenCL and CUDA.

2. We present performance portability results for these seven parallel programming models on a variety of GPUs from two vendors and on several generations of Intel CPU along with IBM's Power 8 and Intel's Xeon Phi (Knights Landing).

3. We update the BabelStream (formally GPU-STREAM) benchmark to provide a 'Rosetta Stone', a simple example code which can assist in understanding how to program in the different programming models. This will also enable testing of future programming models in a simple way.

4. We also include a dot product kernel, in all seven programming models, to additionally examine the available memory bandwidth for reductions.

The paper is structured as follows: in Sec. 2 we introduce the STREAM benchmark and explain the basic structure. In Sec. 3 we describe the key features of the programming models we use in this paper, before presenting performance results in Sec. 4. Finally we conclude in Sec. 5.

2 Measuring Memory Bandwidth

The STREAM Benchmark [14] measures the time taken for each of four simple operators (kernels) applied to three large arrays (a, b and c), where α is a scalar constant. As part of the STREAM2 Benchmark [15], a reduction operation is introduced in the form of a dot product of two of the large arrays. The five kernels in our benchmark, made up from the original four STREAM kernels plus the dot product from STREAM2 are:

1. Copy: $c[i] = a[i]$
2. Multiply: $b[i] = \alpha c[i]$
3. Add: $c[i] = a[i] + b[i]$
4. Triad: $a[i] = b[i] + \alpha c[i]$
5. Dot: $\text{sum} = \text{sum} + a[i] * b[i]$

These kernels have been demonstrated to be *memory bandwidth bound*. The number of bytes read from and written to memory can be modelled by visual inspection of the source code. We let β be the size in bytes of an element — for double precision floating point $\beta = 8$. For an array containing N elements, the copy and multiply kernels read $N\beta$ bytes and write $N\beta$ bytes, totalling $2N\beta$ bytes. The add and triad kernels both read $2N\beta$ bytes and write $N\beta$ bytes, totalling $3N\beta$ bytes. The dot kernel reads $2N\beta$ bytes but does not contain a write to main memory; the result of the reduction is single scalar variable. Running the kernels in the order enumerated above ensures that any caches are invalidated between kernel calls; N is chosen to be large enough so that the arrays are larger than last level cache (on cache-based architectures) to require the data to be moved from main memory — see [14] for the rules of running STREAM which we adopt here. The achieved sustained memory bandwidth can be found as the ratio of bytes moved and the execution time of the kernel. A typical modern CPU can achieve a STREAM result equivalent to 80% or more of its peak memory bandwidth.

BabelStream is a complementary benchmark to the standard CPU version of STREAM. BabelStream enables the measurement of achievable memory

bandwidth across a wide range of multi- and many-core devices [4]. The first version of BabelStream implemented the four STREAM kernels in OpenCL and CUDA, allowing the benchmark to be used across a diverse set of hardware from a wide range of vendors. As a tool it allows an application developer to know how well a memory bandwidth bound kernel is performing. BabelStream is Open Source and available on GitHub at `http://uob-hpc.github.io/BabelStream/`. The webpage maintains a repository of all our results and we encourage submission of additional measurements. In this paper we expand BabelStream to consider a second dimension to this reference point, namely the programming model.

Additionally we include a new kernel which represents a different balance of read and write operations: the dot product. The dot product bandwidth model used is identical to that used in STREAM2, which counts the movement of data to and from main memory alone, ignoring any cache effects. This is equivalent on the GPU to ignoring the implementation details of using shared/local memory as a form of cache. As such, the expected performance of a reduction over a large data set can be realised.

2.1 Related Work

The STREAM2 benchmark [15] is a serial benchmark designed to investigate both the performance of the cache hierarchy and any disparity in the performance of read and write operations. Due to the serial nature of this code along with the complex iteration monitoring, it has not been possible to provide results using this benchmark directly. We have therefore re-implemented the dot product kernel into the original STREAM benchmark [14] and use this as our baseline for this kernel.

The *deviceMemory* benchmark from the Scalable HeterOgeneous Computing (SHOC) Benchmark Suite is an implementation of the triad STREAM kernel [3]. However, this also includes the PCIe transfer time in the bandwidth measurement. Including this factor hides the bandwidth to device memory itself. In a large scale application consisting of many kernels the transfer of data to the GPU would be performed upfront and data would not be transferred at each kernel execution. As such comparing performance "relative to STREAM" is not possible with the SHOC benchmark.

The *clpeak* benchmark, whilst measuring device memory bandwidth implements a reduction so is not a direct comparison to STREAM [2]. It is also not comparable to the dot kernel here as the *clpeak* reduction uses the OpenCL vector types which result in a non-contiguous memory access pattern on the devices tested in this study.

The Standard Parallel Evaluation Corporation (SPEC) ACCEL benchmark suite whilst containing many memory bandwidth bound kernels does not include a STREAM kernel [26].

Performance portability studies have previously only focussed on a single programming model. Sawadsitang et al. focus on the performance of OpenACC compared to CUDA and OpenCL on GPUs and Intel Xeon Phi (Knights Corner) for numerical kernels, however their discussion of performance portability does not take into account the relative performance profiles of the difference devices [25]. Sabne et al. use automatic performance tuning to evaluate the performance portability of OpenACC, and as such does not allow for the baseline performance of the model to be demonstrated [24].

To the authors' knowledge, the only study that has compared the same benchmark in all the programming models of interest across a wide range of devices is one they themselves performed, where the TeaLeaf heat diffusion mini-app from the Mantevo benchmark suite was used in a similar manner to measure performance portability [13, 7].

3 Programming Models

A parallel programming model along with a runtime or an implementation of that model provides programmers a way to write code to run on multiple physical execution units. A common way of providing this functionality is via an Application Programming Interface (API) which may be through function calls, compiler directives or an extension to a programming language.

We briefly introduce each of the programming models used in this paper. Due to the simplicity of the STREAM kernels, we also include the triad kernel in each model to enable the reader to make a look-and-feel comparison. A similar approach was taken with the TeaLeaf mini-app in [13]. This approach also helps to demonstrate the similarities and differences between these parallel programming models, exposing how intrusive or otherwise the models may be for existing code. We take the standard STREAM triad kernel written in a baseline of C++ running on a CPU in serial, as shown in Fig. 1.

We have also included the dot kernel for each model. Some of the models provide a way of describing the reduction from within the API or through the use of a directive, whereas other models require the programmer to write the reduction by hand. The baseline serial dot kernel is shown in Fig. 2. As can be seen the method returns the final result of the reduction and so we require that the reduction is complete with the final value on the host for this method to complete.

The update to the BabelStream benchmark [4] presented in this paper has been designed in a plug-and-play fashion; each programming model plugs into a common framework by providing an implementation of an abstract C++ class. This means that the "driver code" providing the timing routines, etc. is identical between different models. Note that an independent binary is built per parallel programming model, avoiding any possibility of interference between them. Further

programming models are simple to add using this approach.

In considering the memory bandwidth of kernels alone, the transfer of memory between the host and device is not included as in our previous work; the one exception is obtaining the final result of the dot product. Therefore timings are of the kernel execution time and measure the movement of memory on the device alone. The framework developed ensures that all data transfer between host and device is completed before the timing of the kernels are recorded. This therefore requires that each kernel call is *blocking* so that the host may measure the total execution time of the kernels in turn. This is consistent with the approach in the original STREAM benchmark.

Additionally our framework has memory movement routines to ensure that data is valid on the device a priori to the kernel execution. In order to cater for architectures that exhibit NUMA characteristics, the initialisation of the data arrays is performed with a model-specific parallel routine to allow systems with a first-touch allocation policy to allocate memory regions near the cores or compute units that will use them. This was observed to be particularly important for achieving high bandwidth on CPU devices.

3.1 OpenCL

OpenCL is an open standard, royalty-free API specified by Khronos [18]. The model is structured such that a host program co-ordinates one or more attached accelerator devices; this is a fairly explicit approach as the API gives control over selecting devices from a variety of vendors within a single host program. Because OpenCL is designed to offload to generic devices, vendor support is widespread from manufactures of CPUs, GPUs, FPGAs and DSPs.

Each OpenCL device has its own memory address space, which must be explicitly controlled by the programmer; memory is not shared between the host and device. OpenCL 2.0 introduced a Shared Virtual Memory concept which allows the host and device to share an address space, although explicit synchronisation for discrete devices is still required via the host to ensure memory consistency.

Kernels are typically stored as plain text and are compiled at run time. The kernels are then run on the device by issuing them to a command queue. Data movement between host and device is also coordinated via a command queue.

The host API is provided via C function calls, and a standard C++ interface is also provided. Kernels are written in a subset of C99; OpenCL 2.2 provisionally allows kernels to be written in C++. The BabelStream triad kernel in OpenCL C99 is shown in Fig. 3.

OpenCL 1.x does not include any built-in reduction constructs and so we must implement one by hand; we use a two-stage commutative reduction found commonly in vendor software development kits and online (e.g. [1] where this shows the best performance). Local memory is utilised to share partial reduction values between work-items within the work-group. We run four work-groups per compute unit, with the size of each block set to the device maximum work-group size. OpenCL provides API calls to query the device for these specific values at run time. The partial sum from each work-group is written to global memory, and these are copied back to the host for the final summation in serial. The BabelStream dot kernel in OpenCL C99 along with the routine to launch the kernel and compute the final summation is shown in Fig. 4.

3.2 CUDA

CUDA is a proprietary API from NVIDIA for targeting their GPU devices [19]. CUDA kernels are written in a subset of C++ and are included as function calls in the host source files. They are compiled offline.

The API is simplified so that no explicit code is required to acquire a GPU device; additional routines are provided to allow greater control if required by the programmer.

In the more recent versions of CUDA the memory address space is shared between the host and the GPU so that pointers are valid on both. Synchronisation of memory access is still left to the programmer. CUDA also introduces Managed memory which allows a more automatic sharing of memory between host and device. With CUDA 8 and the latest Pascal GPUs, the GPU is allowed to cache data accessed from the host memory; previously it was zero copy.

The BabelStream triad kernel is shown in Fig. 5. Note that CUDA *requires* specification of the number of threads per thread-block, and as such any choice may be non-optimal across devices for a single source kernel. A previous study showed that 1024 threads per thread-block achieves reasonable performance and so we use this value in our study [23]. Note therefore the size of the arrays must be divisible by 1024 in our implementation.

CUDA does not provide any built-in reduction implementations. We implement a two-stage commutative reduction by hand that matches the one we used for OpenCL. Shared memory is used to share partial reduction values between threads within the thread block, with the final values written to device memory. The partial values are copied back to the host and the final summation is completed in serial. We launch the kernel with 256 thread blocks, each block containing 1024 threads. The BabelStream dot kernel is shown in Fig. 6.

3.3 OpenACC

The OpenACC Committee, consisting of members including NVIDIA/PGI, Cray and AMD, partitioned from the OpenMP standard to provide a directive-based solution for offloading to accelerators [20]. The accelerator is programmed by adding compiler directives

(pragmas or sentinels) to standard CPU source code. A few API calls are also provided to query the runtime and offer some basic device control and selection.

There are two different options for specifying the parallelism in offloaded code. The OpenACC `parallel` construct starts parallel execution on the device, redundantly if no other clauses are present. The `loop` construct is applied to the loop to describe that the loop iterations are to be distributed amongst threads on the device (a single vector lane of a single worker of a single gang). The `kernels` pragma indicates that the region will be offloaded to the device as a series of 'kernels' and any loops encountered may be executed as a kernel in parallel. The `kernels` construct allows the compiler to make decisions about the parallelism, whereas the `parallel` construct gives the programmer control to define the parallelism. The parts of the code which are run on the accelerator are compiled offline, and can be tuned for particular accelerators via compiler flags.

Current implementations of OpenACC can target devices including AMD and NVIDIA GPUs, IBM Power CPUs and x86 multi-core CPUs. Current OpenACC compilers that are available include GCC 6.1, Cray and PGI (NVIDIA).

The BabelStream triad kernel is shown in Fig. 7. Note that a `wait` clause is required for the offload to be blocking as is required by our framework to ensure timing is correct (see Sec. 3). The `present` clause specifies that the memory is already available on the device and ensures a host/device copy is not initiated.

Through the use of the `kernel` directive the OpenACC implementation can determine at compile time if a reduction operation is required, and as such we can use the same compiler directive as for the other kernels. The programmer does not need to specify that a reduction operation occurs. OpenACC does provide a `reduction` clause for use with the `parallel loop` directive. The BabelStream dot kernel is shown in Fig. 8.

3.4 OpenMP

The OpenMP specification from the OpenMP Architecture Review Board has traditionally allowed thread based parallelism in the fork-join model on CPUs [21]. The parallelism is described using a directive approach (with pragmas or sentinels) defining regions of code to operate (redundantly) in parallel on multiple threads. Work-sharing constructs allow loops in a parallel region to be split across the threads. The shared memory model allows data to be accessed by all threads. An OpenMP 3 version of the triad kernel, suitable for running only on CPUs is shown in Fig. 9.

The OpenMP 4.0 specification introduced the ability to offload regions of code to a target device. The approach has later been improved in the OpenMP 4.5 specification. Structured blocks of code marked with a `target` directive are executed on the accelerator, whilst by default the host waits for completion of the offloaded region before continuing. The usual work-sharing constructs allow loops in the `target` region and further directives allow finer grained control of work distribution.

Memory management in general (shallow copies) is automatically handled by the implementation; the host memory is copied to the device on entry to the offloaded region by natural extensions to the familiar implicit scoping rules in the OpenMP model. Finer grained control of memory movement between the host and device is controlled via `target data` regions and memory movement clauses; in particular arrays must be mapped explicitly.

The unstructured `target data` regions in OpenMP 4.5 allow simple integration with our framework. The scoping rules of OpenMP 4.0 would require that the memory movement to the device to have been written in our driver code, breaking the separation of implementation from driver code in our testing framework; OpenMP 4.5 fixes this issue present in the original 4.0 specification.

The OpenMP 4 version of the BabelStream triad kernel is shown in Fig. 11.

The same directives shown previously can be used with the addition of a `reduction` clause for the dot kernel, as shown for BabelStream in Figs. 10 and 12. The clause requires the programmer to specify the reduction operator (a sum `+` in this case) and the reduction variable.

Whilst the directives are different for CPU and GPU implementations, the loop bodies are identical. For BabelStream we use a preprocessor switch to determine which variant of the pragmas to use at compile time.

3.5 Kokkos

Kokkos is an open source C++ abstraction layer developed by Sandia National Laboratories that allows users to target multiple architectures using OpenMP, Pthreads, and CUDA [6]. The programming model requires developers to wrap up application data structures in abstract data types called Views in order to distinguish between host and device memory spaces. Developers have two options when writing Kokkos kernels: (1) the functor approach, where a templated C++ class is written that has an overloaded function operator containing the kernel logic; and (2) the lambda approach, where a simple parallel dispatch function such as `parallel_for` is combined with an anonymous function containing the kernel logic. It is also possible to nest the parallel dispatch functions and achieve nested parallelism, which can be used to express multiple levels of parallelism within a kernel.

The Kokkos version of the BabelStream triad kernel is shown in Fig. 13.

Kokkos provides the `parallel_reduce` dispatch function to describe a reduction, and is shown for BabelStream in Fig. 14. The loop body must be changed to use a temporary for the reduction variable which is

declared as an argument to the lambda function; in the baseline code in Fig. 2 the `sum` variable is used but we must the new `tmp` lambda argument in the loop body instead. The dispatch function is also passed the original reduction variable so that it can be set.

3.6 RAJA

RAJA is a recently released C++ abstraction layer developed by Lawrence Livermore National Laboratories that can target OpenMP and CUDA [8]. RAJA adopts a novel approach of precomputing the iteration space for each kernel, abstracting them into some number of Segments, which are aggregated into a container called an IndexSet. By decoupling the kernel logic and iteration space it is possible to optimise data access patterns, easily adjust domain decompositions and perform tiling. The developer is required to write a lambda function containing each kernel's logic that will be called by some parallel dispatch function, such as `forall`. The dispatch functions are driven by execution policies, which describe how the iteration space will be executed on a particular target architecture, for instance executing the elements of each Segment in parallel on a GPU.

The policy is typically set using a `typedef` in a global header file and so only need setting at this high level. The index set is again usually set in a globally accessible instance and so only needs setting once for each unique loop bound.

The RAJA version of the BabelStream triad kernel is shown in Fig. 15.

RAJA provides reduction classes which wrap the reduction variables. For our summation operation, the result is instead declared as a `RAJA::ReduceSum` instead of a simple native data type. The loop body (expressed as a lambda) therefore requires no change. The BabelStream dot kernel is shown in Fig. 16.

3.7 SYCL

SYCL is a royalty-free, cross-platform C++ abstraction layer from Khronos that builds on the OpenCL programming model (see Sec. 3.1) [12]. It is designed to be programmed as single-source C++, where code offloaded to the device is expressed as a lambda function or functor; template functions are supported.

SYCL aims to be as close to standard C++14 as possible, in so far as a standard C++14 compiler can compile the SYCL source code and run on a CPU via a header-only implementation. A SYCL device compiler has to be used to offload the kernels onto an accelerator, typically via OpenCL. The approach taken in SYCL 1.2 compilers available today is to generate SPIR, a portable intermediate representation for OpenCL kernels. The provisional SYCL 2.2 specification will require OpenCL 2.2 compatibility.

The SYCL version of the BabelStream triad kernel is shown in Fig. 17.

The two-stage commutative reduction used for OpenCL and CUDA is again implemented for our SYCL version of the dot kernel as SYCL does not have any built in reduction APIs. The BabelStream dot kernel is shown in Fig. 18.

4 Results

Table 1 lists the many-core devices that we used in our experiment. The GPU micro-architecture code names are shown in parenthesis. The CPUs are all dual-socket, except for the Intel Xeon Phi. Given the breadth of devices and programming models we had to use a number of platforms and compilers to collect results. Intel does not formally publish the peak MCDRAM bandwidth for the Xeon Phi (Knights Landing), so the presented figure is based on published claims that MCDRAM's peak memory bandwidth is five times that of it's DDR.

The HPC GPUs from NVIDIA were attached to a Cray XC40 supercomputer 'Swan' (K20X), a Cray CS cluster 'Falcon' (K40 and K80), and a Cray XC50/XC40 supercomputer 'Piz Daint' (P100). The NVIDIA GTX 980 Ti and Titan X are attached to an experimental cluster at the University of Bristol (the "Zoo"). The AMD GPUs were also attached to the experimental "Zoo" cluster in Bristol. The Sandy Bridge CPUs are part of BlueCrystal Phase 3, part of the Advanced Computing Research Centre at the University of Bristol. The Ivy Bridge CPUs are part of the Cray XC30 supercomputer 'Edison' at the National Energy Research Scientific Computing Center (NERSC). The Haswell and Broadwell CPUs are part of the Cray XC40 supercomputer 'Swan'. The Intel Xeon Phi (Knights Landing) is part of the "Zoo" experimental cluster in Bristol. The IBM Power 8 CPUs are part of an Advanced Systems Technology Test Bed at Sandia National Laboratories, and the system includes the off-chip Centaur L4 caches.

Tables 2 and 3 show the compiler and driver versions used for each combination of device and programming model. The clang-ykt fork of Clang for OpenMP used on the "Zoo" is available at `https://github.com/clang-ykt`; note that the Clang OpenMP 4.x implementation is still under development and is not a stable release. The ComputeCpp compiler from Codeplay generates SPIR, but as SPIR support is not available in the proprietary OpenCL drivers from NVIDIA we used pocl [9] with an experimental NVIDIA backend developed at the University of Bristol for these SYCL results. For OpenACC on Broadwell and KNL, we targeted the Haswell architecture as Broadwell/KNL are not yet available options in the PGI compiler.

In the next few sections we describe our experiences in porting the BabelStream kernels to the seven different parallel programming models in our study, and describe the performance we were able to achieve when running

these implementations on this diverse range of multi- and many-core devices.

4.1 Code changes and experiences

The C++ solutions of SYCL, RAJA and Kokkos all provide a similar syntax for describing the parallel work. A for-loop is replaced by an equivalent statement with the loop body expressed as a lambda function. The directive based approaches of OpenMP and OpenACC both annotate for-loops with compiler directives which describe the parallelism of the loop. OpenCL and CUDA require the loop body to be written in a separate function which is then instantiated on the device with an API call which defines the number of iterations; the iteration no longer is expressed as a loop. Table 4 gives an idea of how much code was required to implement this benchmark in each of the programming models. The number of lines of code in the specific implementation in each of the programming models of our virtual class (code plus header file) was counted and is shown in the first column. For each version we also include the change in the number of lines of code compared to our baseline serial version in C++ implemented in our framework.

Whilst the authors found that writing this simple benchmark in each of the programming models was a simple task, getting them to build on a variety of platforms for a variety of devices was a significant challenge in many cases. Additionally, some changes to the code were required in order for specific platform and compiler combinations to be performant, or in some cases to work at all.

OpenACC using the PGI compiler targeting host CPUs, the AMD GPUs and the NVIDIA consumer GPUs, all required specifying the pointers as `restrict` in order for the loop to be parallelised, although this is not standard C++. However, this causes the Cray OpenACC compiler to serialise the offloaded region; the loop does execute on the device but in serial and so has poor performance. When collecting the results we removed `restrict` for the Cray results and inserted `restrict` for the PGI results as we believe this to be an issue in the compiler itself rather than the model. Using `parallel loop independent` does parallelise the loop without specifying `restrict`. However, PGI advocate the use of the `kernels` directive over the `parallel` directive [17]. This would be relatively simple change, in a simple benchmark case, but there may be larger codes where the reason the automatic parallelism fails may not be evident. Additionally this would result in the programmer changing the *way* they express the same parallelism when using a particular programming model by altering the code for a different architecture or compiler, the code itself is no longer performance portable — you require one implementation per compiler even if they target the same device.

Table 4 shows that OpenMP involved more lines of additional code than OpenACC. This is due to including the pragmas for the CPU and GPU architectures in the same source, with the preprocessor definition to switch between them as previously discussed in Sec. 3.4.

However, all compilers supporting OpenMP 4 and OpenACC would not correctly offload the kernel without re-declaring the arrays as local scope variables; note that this is down to the implementation of the models rather than the memory models themselves. These variables are declared inside the class, but the compilers were unable to recognise them in the directives (Cray), or else crash at runtime (PGI targeting GPUs). The authors have found this is also the case with using structure members in C. It is the opinion of the authors that these local variables should not be required for correct behaviour as a valid pointer value is passed to the directive; however we note an additional level of dereferencing would be required by the model implementation to access class members.

In addition to these code changes, the data points required specific compiler invocations for each combination of compiler, programming model, device and host system.

4.2 Performance

We display the fraction of peak memory bandwidth we were able to achieve for a variety of devices against each programming model for the triad kernel in Fig. 19 and for the dot kernel in Fig. 20. The figures show the sustained peak memory bandwidth obtained, the percentage of theoretical peak obtained (using the peak numbers from Table. 1), and the percentage of the best achieved memory bandwidth across all models for that device. The percentage of theoretical peak allows us to see how close compared to vendor quoted memory bandwidth is achievable in each case, whilst the percentage of best achieved highlights any difference between the performance of the programming models for each device. We used 100 iterations with an array size of 2^{25} double precision elements (268 MB).

We will analyse these results in appropriate groups of devices from each vendor, and compare the performance achieved for the triad and dot kernels in the programming models, wherein the figures should be read in vertical columns, bottom to top. In Sec. 4.2.6 we will compare the performance of each of the programming models across the range of devices tested, wherein the figures should be read in horizontal rows, left to right. Some of the combinations of hardware and programming model were not supported, and details of these cases are presented in Sec. 4.3. We used the OpenMP backend of RAJA and Kokkos to target the CPUs and the CUDA backend to target the NVIDIA GPUs.

4.2.1 NVIDIA GPUs

The NVIDIA GPUs used in this study consist of the HPC K20X, K40, K80 and P100, along with the consumer GTX 980 Ti and Titan X. The P100 with its HBM2 memory demonstrates the highest achievable

memory bandwidth of all the devices tested, as seen by obtaining the highest value in Fig. 19a; this is also true of the dot kernel in Fig. 20a.

The achievable performance for codes targeting NVIDIA GPUs written in any of the models are consistent, across the different generations of GPU. Figure 19b shows that all the models can achieve 60–80% of theoretical peak across all devices. More impressively the models all achieve over 90% of relative peak performance (with the exception of RAJA on a K80 which achieves 88% relative). As such both the high-level C++-style models and directive based approaches all demonstrate equivalent performance to each other and to the lower level models of CUDA and OpenCL, the stalwart of programming these devices; this is a very encouraging result.

The dot kernel on the P100 demonstrates a higher available bandwidth than the triad kernel, achieving 75% of peak for triad and 80% of peak for dot. This increase is less pronounced for the earlier generations of NVIDIA GPU, in particular the Kepler architecture. The Titan X is also a Pascal architecture, with GDDR5X memory instead of HBM2, and shows a small improvement in bandwidth from triad to dot, however the increase is smaller compared to the P100 result.

Figure 20c shows that the dot kernel in OpenACC does not obtain good memory bandwidth with the Cray compiler on the K40 and K80 GPUs, indicating a performance issue in the code generated for the reduction kernels. The PGI compiler was not available on these systems, however showed good performance on the other NVIDIA GPUs where this was available. The Cray OpenMP implementation shows very good performance relative to CUDA, with the immature clang-ykt providing good bandwidth for triad, but with lower bandwidths for some devices for the dot kernel. It is expected that these are performance bugs and will eventually be fixed in the respective compilers.

4.2.2 AMD GPUs

All the graphs in Figs. 19 and 20 show that we were unable to collect a full set of data points; of all the vendors, general support for most of the programming models is lacking at the time of writing. A fuller explanation of the reasons for the missing points is listed in Sec. 4.3. This should be viewed as a weakness of the implementations of the models, rather than any issues with the programming models applicability to AMD hardware.

It should be noted that the data points that we were able to collect for AMD's GPUs achieved the highest fractions of peak for all GPUs, 84–86% for the triad kernel. The OpenACC dot kernel performance relatively poorly, as shown in Fig. 20c, but is on a par with the percentage of theoretical peak performance on OpenACC obtained on NVIDIA GPUs.

4.2.3 Intel CPUs

We use the original 'McCalpin' STREAM benchmark written in C and OpenMP as a baseline comparison for the CPU results. Thread binding is used via setting the OpenMP environment variable `OMP_PROC_BIND` to `true`, and the number of threads is specified via `OMP_NUM_THREADS` to equal the number of cores. We found that running one thread per core gave the best memory bandwidth and so hyperthreads we not used here. The OpenMP specification states that threads are not allowed to move between places (which are implementation defined by default). Therefore the OpenMP implementation is free to choose the best way to pin the threads to cores. We did not experiment with turning on streaming stores, however the Intel compiler was free to generate these instructions.

The available memory bandwidth has improved through the generations of Intel CPUs, from an achievable 66 GB/s to 128 GB/s for triad. However, this bandwidth is generally unavailable using all of programming models. Fig. 19b shows that there is some loss of performance when using C++ and OpenMP 3 for running on CPUs compared to the original STREAM benchmark written in C, marked as 'McCalpin' on the figures. For example, on Broadwell CPUs the C++ version achieves 64% of peak memory bandwidth, compared to 83% when using the C version. Both these codes use the standard OpenMP 3 programming model, however the original STREAM benchmark has the problem size known at compile time and so the compiler is able to make additional optimisations including on the peel and remainder loops based on the trip count. In more realistic (non-benchmark) kernels, this information would likely not be available at compile time. The dot kernel shows a similar drop in performance compared to the McCalpin version; indeed Figs. 19c and 20c show that the McCalpin versions are the only results achieving 100%. The McCalpin benchmark shows that the dot kernel gets an improved bandwidth over the triad kernel.

Both RAJA and Kokkos use the OpenMP programming model to implement parallel execution on CPUs. The performance results on all four CPUs tested show that RAJA and Kokkos performance matches that of hand-written OpenMP for BabelStream for both the triad and dot kernels. This result shows that both RAJA and Kokkos provide little overhead over writing OpenMP code directly, at least for BabelStream. As such they may provide a viable alternative to OpenMP for writing code in a parallel C++-style programming model compared to the directive based approach in OpenMP. However as noted above, C++ compiler implementations of OpenMP may suffer from a performance loss compared to a C with OpenMP implementation.

We used PGI's implementation of OpenACC for multi-core CPUs, requesting thread pinning using `MP_BIND=yes` and specifying the number of threads with `ACC_NUM_CORES`. OpenACC does not demonstrate good

peak performance on the CPUs in general, as can be seen in Figs. 19b and 20b, which we attribute to poor NUMA behaviour. Although we use a parallel loop to initialise the data in each array, we conclude that the directive used to create the device allocations is undesirably allocating data on a single socket. If we instead use the `acc_malloc` routine to allocate the device arrays we observe much higher performance on CPUs, however this approach currently does not function correctly on GPUs, and we assert that our original directive approach *should* exhibit the correct first-touch behaviour.

OpenCL is able to run on the CPU as well, and we tested using the Intel OpenCL runtime. For the triad kernel OpenCL gets close to the C++ OpenMP performance in Sandy Bridge and Haswell, but is lower on Ivy Bridge and Broadwell, as seen in Fig. 19c. The dot kernel struggles to match the C++ OpenMP performance, however achieves similar to OpenACC performance in Fig. 20c. We can achieve higher reduction performance on CPUs in OpenCL by using a more cache-friendly access pattern, but this approach results in performance degradation on the GPU devices. This performance portability issue arises due to the explicit nature of OpenCL, which forces the programmer to describe the memory access pattern.

Figure 19c shows that SYCL has little overhead for the triad kernel over OpenCL. However Fig. 20c shows that the dot kernel is around 20-30% slower than OpenCL, which could indicate some overheads in the code generated for local memory use and work-item synchronisation.

4.2.4 Intel Xeon Phi (Knights Landing)

The Intel Xeon Phi (KNL) has MCDRAM memory which offers an increased memory bandwidth over the DDR found in the the other CPUs. It is clear in Fig. 19a that this device provides a significant improvement over all the CPUs for the triad kernels. It also demonstrated more bandwidth than the GDDR GPUs, however the P100 with HBM2 does offer a further improvement. We ran one thread per core, using `numactl` to ensure allocation in MCDRAM. Again, we found that running one thread per core gave the best memory bandwidth and so hyperthreads we not used here. We used the same environment variables for thread binding and core counts with OpenMP and OpenACC as used for the Intel CPUs in Sec. 4.2.3.

As with the Intel CPUs, the highest bandwidth was demonstrated with the original McCalpin benchmark, and the C++ OpenMP versions show a drop in performance; see Fig. 19c. Kokkos again matches the OpenMP implementation, however the performance of RAJA is lower than expected as in a previous study the performance of RAJA matched that of Kokkos on this architecture [5]. This may be due to the recent updates to the implementation of the RAJA library. The dot kernel in particular obtains low performance.

OpenACC again shows a further loss in performance. We were able to collect a result for OpenCL on KNL by setting an environment variable: `VOLCANO_CPU_ARCH=core-avx2`; however the performance is lacking. This approach restricts the OpenCL compiler to generate AVX2 instructions, which only deliver half the vector width of the KNL's AVX512 ALUs. Additionally, the KNL has two vector floating point ALUs, but only one is capable of running legacy (AVX2 and lower) vector instructions [10], which further limits the device utilisation.

Unlike with Intel CPUs, the dot kernel does not achieve a higher memory bandwidth than the triad kernel; triad 448 GB/s compared to dot 340 GB/s. The triad kernel has two read operations and one write operation for each kernel invocation, whereas the dot kernel does not have the write operation. A previous study found that because MCDRAM has separate read and write channels, streaming writes can be performed at the same time as the reads [11]. As such they appear as extra memory bandwidth without increasing runtime and so the measure of achieved memory bandwidth is increased. With reads alone the maximum memory bandwidth is not possible to obtain as the write channel remains unused.

4.2.5 Power 8

The Power 8 results were collected with one thread per core. The bandwidth presented in Fig. 19b of 78% of theoretical peak is using the same problem size as all the other results; a higher bandwidth is possible with a large problem as has been previously observed that performance can decrease with smaller problems [22].

The Power 8 CPU obtains a higher peak bandwidth (nearly 299 GB/s) than the Intel CPUs tested, over double that of the Broadwell (Fig. 19a). The C++ OpenMP implementation of BabelStream almost matches the performance of the original McCalpin STREAM benchmark; the drop in performance seen with this on the Intel CPUs is not present here. Again, RAJA and Kokkos show little overhead over the directive based OpenMP programming model. Figure 19c shows that the available programming models all provide good performance on this platform.

The performance of the dot kernel shows more variability. As with the Intel Xeon Phi (Knights Landing) the dot kernel achieves lower performance than the triad kernel; in this case about 20 GB/s lower. Although each Power 8 socket has a memory bandwidth of 192 GB/s, this is split into 128 GB/s of read bandwidth and 64 GB/s write bandwidth [22]. As the dot kernel does not contain any writes, we are therefore not maximising the use of the memory channels and so would expect a drop in achievable memory bandwidth.

As with the triad kernel, the C++ OpenMP version of dot gives almost identical performance to the McCalpin STREAM benchmark. However performance is lost with the Kokkos implementation of this kernel, reducing the

bandwidth to 67% of the best achieved (Fig. 20c). The RAJA performance for this kernel is very low, only 6 GB/s and so there is clearly an issue with the implementation of reductions in the RAJA framework for large thread counts.

4.2.6 Performance portability of the programming models

This section comments on the percentage of achievable memory bandwidth obtained by each programming model across the different devices. The results are for the triad kernel unless specified otherwise. An important observation is that none of the programming models presented here run across all of the devices we investigated. Where performance is lacking it is generally down to the implementation of the model itself, rather than an inherent property of the model which are all (theoretically) portable.

OpenCL and SYCL provide the greatest coverage of devices, running on NVIDIA and AMD GPUs, Intel CPUs and Xeon Phi (KNL), only being unsupported on Power 8. These programming models provided good performance across all the GPUs, greater than 95% of the best achievable, however only greater than 50% of the best on the Intel CPUs and KNL. For the reduction kernel the performance on CPUs is currently lacking, however the performance on GPUs from both vendors is good.

OpenMP was able to run on the NVIDIA GPUs, Intel CPUs and KNL and Power 8, with only the AMD GPUs currently unsupported. The performance was also good, with over 91% of best achievable performance on NVIDIA GPUs and over 74% on Intel CPUs (over 68% on KNL). This slightly lower performance on the CPUs is likely down to the issues with optimisation in the compiler as discussed in Sec. 4.2.3. We had to program two different directives for the CPU and GPU architectures in order to achieve this portability, as the model differentiates between an offload and non-offload model using different directives. For BabelStream this was simple as we could use a compiler definition to choose the directives at compile time, however we had to duplicate the directives on each kernel. The performance of the reduction is generally good across all devices, with a small performance issue on the GTX 980 Ti; although this result uses an immature compiler.

Whilst OpenACC demonstrated over 94% of best achievable performance on the supported GPUs, the performance on CPUs was somewhat lacking averaging 41% of best achievable. The performance with the Cray compiler on the NVIDIA Kepler GPUs is also lacking for the reduction kernel.

The CUDA programming model achieves good portability across the different generations of NVIDIA GPU for both kernels, obtaining over 97% of best across the architectures. However the model is only supported on these GPUs and so is not portable beyond this single vendor.

The RAJA portability layer demonstrates 90% of best achievable bandwidth on NVIDIA GPUs and 73% on CPUs. Unfortunately performance on KNL is lacking at the time of writing. Code changes were required to change the policy when changing between CPUs and GPUs, however this is a global change and therefore simple to mange even in a very large code base. However, RAJA requires the programmer to manage memory themselves, and so it required use of the CUDA API to allocate and copy memory between the host and the GPU; as such all memory allocations in the code need to be done differently between architectures. The reduction kernel suffers on the KNL and on the Power 8; both of these are highly threaded devices utilising the RAJA OpenMP back end.

Kokkos provides similar portability to RAJA, but shows marginally improved results on NVIDIA GPUs with over 96% of best achievable bandwidth, and over 73% on the Intel CPUs. Kokkos required a single line of code to be changed to switch between the CPU and GPU, that specifies the device to allocate the View (the memory) on. This feature solves the issue with RAJA about having to change the data allocations between devices. The reduction kernel also performs well on the KNL with 96% of best achievable but only 67% on the Power 8.

OpenCL and SYCL provide the greatest functional portability of the devices tested. The other models, apart from CUDA, show equal coverage of the other devices. OpenACC does additionally support the AMD Hawaii GPUs but does not support their latest generation. OpenACC performance is also not as consistent as RAJA, Kokkos and OpenMP across the devices tested. RAJA and Kokkos demonstrate little overhead over the directive based approach of OpenMP.

4.3 Missing data points

All of the parallel programming models explored in this paper are designed to be portable; at the very least this should enable running on a variety of devices across a variety of vendors. However, as can be seen in Figs. 19 and 20 there are a surprising number of results that are not possible to obtain. We mark those that are impossible to collect due to missing implementations from the vendors as 'N/A'. Note that the original STREAM benchmark from McCalpin was written in C with OpenMP 3 and so cannot run on GPUs.

There are currently no implementations of OpenCL for Power 8, and so a result for this and SYCL was not possible to obtain. Additionally the PGI Accelerator compiler does not support Power 8 as a target and so it was not possible to compile multi-core OpenACC code.

Neither Kokkos and RAJA fully support the AMD GPUs used in this paper. The main development effort of GPU support in RAJA and Kokkos has been written using CUDA, and as such they cannot currently support AMD GPUs. This is currently a weakness of the RAJA and Kokkos implementations, which could be addressed

when proper OpenMP 4.5 support becomes more widely available as both RAJA and Kokkos use OpenMP for their CPU implementation.

For AMD GPUs, GCC 6.1 introduces OpenMP 4.5 support, but only for integrated graphics in the APU devices, not for the discrete GPUs used in this paper; therefore we cannot present OpenMP results for the AMD GPUs. The PGI Accelerator compiler supports OpenACC on AMD GPUs up to Hawaii so we used this for the S9150, but the Fury X's newer Fiji architecture is not supported.

There is no CUDA compiler directly targeting AMD GPUs.

The PGI compiler no longer has support for CUDA-x86, whereby CUDA code could target host CPUs, and therefore no results are presented.

5 Conclusion

The simple triad and dot product kernels have been presented in a variety of parallel programming models including those in a C++-style and those which are directive based. The performance of both of these kernels has been tested across a variety of CPUs and many-core devices including GPUs and Xeon Phi.

None of the programming models is currently available to run a single code across all devices that we tested. Whatever definition of 'performance portability' one might wish, a performance portable code must also at least be functionally portable across different devices.

The results in Fig. 19 and 20 show that most of the models are able to provide good performance for these kernels across the range of different architectures. The other kernels (Copy, Mul and Add) have a similar performance profile to Triad. The OpenACC compilers demonstrate good GPU performance on products from both NVIDIA and AMD, however the CPU performance is currently lacking primarily because of the first-touch issues discussed in Sec. 4.2.3. This limits OpenACC's relevance to CPUs due to implementations of the model at the time of writing.

The directive based approaches of OpenMP and OpenACC look to provide a good trade off between performance and code complexity. The number of lines to add to an existing piece of code is minimal; in particular for a C or Fortran style code. If code is already written using C++ idioms, then SYCL, RAJA and Kokkos provided a similar level of minimal disruption for performance.

Many-core devices such as GPUs and Xeon Phi offer an increased memory bandwidth over CPUs, although the latest CPU offerings are competitive with GDDR memory GPUs. The advantages of the high bandwidth memory in the forms of MCDRAM, HBM1 and HBM2 are clear, and for bandwidth bound kernels the benefits are very pronounced. It is a positive result that the programming models here are able to successfully take advantage of this technology.

The code is Open Source and available on GitHub at `http://uob-hpc.github.io/BabelStream/`. By presenting the same simple kernels in multiple programming models, the benchmark is also able to act as a guide for programmers when writing programs in these models for the first time, or performing ports between them.

Acknowledgements

We would like to thank Cray Inc. for providing access to the Cray XC40 supercomputer, Swan, and the Cray CS cluster, Falcon. We also wish to thank Alice Koniges at NERSC for access to Edison and Maria Grazia Giuffreda at CSCS Swiss National Supercomputing Center for access to Piz Daint. The authors extend their thanks to Si Hammond and Sandia National Labs; the results used are run on the Sandia ASC Architecture Test Beds program. Our thanks to Codeplay for early access to the ComputeCpp SYCL compiler and to Douglas Miles at PGI (NVIDIA) for access to the PGI compiler. We would also like to thank the University of Bristol Intel Parallel Computing Center (IPCC). This work was carried out using the computational facilities of the Advanced Computing Research Centre, University of Bristol - `http://www.bris.ac.uk/acrc/`. Thanks also go to the University of Oxford for access to the Power 8 system.

References

[1] AMD. OpenCL Optimization Case Study - Simple Reductions. http://tinyurl.com/m6rtpw3.

[2] Krishnaraj Bhat. clpeak, 2015.

[3] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S Vetter. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-3, pages 63–74, New York, NY, USA, 2010. ACM.

[4] Tom Deakin and Simon McIntosh-Smith. GPU-STREAM: Benchmarking the Achievable Memory Bandwidth of Graphics Processing Units (poster). In *Supercomputing*, Austin, Texas, 2015.

[5] Tom Deakin, James Price, Matt Martineau, and Simon McIntosh-Smith. *GPU-STREAM v2.0: Benchmarking the Achievable Memory Bandwidth of Many-Core Processors Across Diverse Parallel Programming Models*, pages 489–507. Springer International Publishing, Cham, 2016.

[6] H. C. Edwards and D. Sunderland. Kokkos Array Performance-portable Manycore Programming

Model. In *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM'12)*, pages 1–10. ACM, 2012.

[7] M.A. Heroux, D.W. Doerfler, et al. Improving Performance via Mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, 2009.

[8] R D Hornung and J A Keasler. The RAJA Portability Layer : Overview and Status. *Lawrence Livermote National Laboratory Technical Report*, 2014.

[9] Pekka Jääskeläinen, Carlos Sánchez de La Lama, Erik Schnetter, Kalle Raiskila, Jarmo Takala, and Heikki Berg. pocl: A Performance-Portable OpenCL Implementation. *International Journal of Parallel Programming*, 43(5):752–785, 2015.

[10] Jim Jeffers, James Reinders, and Avinash Sodani. *Intel Xeon Phi Processor High Performance Programming*. Morgan Kaugmann, Cambridge, MA, Knights Landing edition, 2016.

[11] Jim Jeffers, James Reinders, and Avinash Sodani. Quantum chromodynamics. In *Intel Xeon Phi Processor High Performance Programming*, pages 581–598. Elsevier, 2016.

[12] Khronos OpenCL Working Group SYCL subgroup. SYCL Provisional Specification, 2016.

[13] Matt Martineau, Simon McIntosh-Smith, Mike Boulton, and Wayne Gaudin. An Evaluation of Emerging Many-Core Parallel Programming Models. In *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM'16, pages 1–10, New York, NY, USA, 2016. ACM.

[14] John D McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, Dec 1995.

[15] John D McCalpin. STREAM2. https://www.cs.virginia.edu/stream/stream2/, 1999.

[16] John D McCalpin. Memory Bandwidth and System Balance in HPC Systems. SC'16 Invited Talk, 2016.

[17] Doug Miles. When will OpenACC and OpenMP merge. Supercomputing, 2016.

[18] Aaftab Munshi. The OpenCL Specification, Version 1.1, 2011.

[19] NVIDIA. CUDA Toolkit 7.5.

[20] OpenACC-Standard.org. The OpenACC Application Programming Interface - Version 2.5, 2015.

[21] OpenMP Architecture Review Board. OpenMP Application Program Interface, Version 4.5, 2015.

[22] István Z Reguly, Abdoul-Kader Keita, and Michael B Giles. Benchmarking the IBM Power8 processor. In *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*, pages 61–69, Riverton, NJ, USA, 2015. IBM Corp.

[23] Karl Rupp. GPU Memory Bandwidth vs. Thread Blocks (CUDA) / Workgroups (OpenCL), 2016.

[24] Amit Sabne, Putt Sakhnagool, Seyong Lee, and Jeffrey S Vetter. Evaluating Performance Portability of OpenACC. *27th International Workshop on Languages and Compiler for Parallel Computing (LCPC)*, pages 1–15, 2014.

[25] Suttinee Sawadsitang, James Lin, Simon See, Francois Bodin, and Satoshi Matsuoka. Understanding Performance Portability of OpenACC for Supercomputers. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 699–707. IEEE, may 2015.

[26] Standard Performance Evaluation Corporation. SPEC Accel. https://www.spec.org/accel/, 2016.

```
template <class T>
void triad()
{
  const T scalar = 0.4;
  for (int i = 0; i < array_size; i++)
    a[i] = b[i] + scalar * c[i];
}
```

Figure 1: STREAM triad baseline kernel in C++

```
template <class T>
T dot()
{
  T sum = 0.0;
  for (int i = 0; i < array_size; i++)
    sum += a[i] * b[i];
}
```

Figure 2: STREAM dot baseline kernel in C++

```
std::string kernels{R"CLC(
  constant TYPE scalar = 0.4;

  kernel void triad(
    global TYPE * restrict a,
    global const TYPE * restrict b,
    global const TYPE * restrict c)
  {
    const size_t i = get_global_id(0);
    a[i] = b[i] + scalar * c[i];
  }
)CLC"};

template <class T>
void OCLStream<T>::triad()
{
  (*triad_kernel)(
    cl::EnqueueArgs(queue, cl::NDRange(array_size)),
    d_a, d_b, d_c
  );
  queue.finish();
}
```

Figure 3: OpenCL triad kernel

```
std::string kernels{R"CLC(
  kernel void stream_dot(
    global const TYPE * restrict a,
    global const TYPE * restrict b,
    global TYPE * restrict sum,
    local TYPE * restrict wg_sum,
    int array_size)
  {
    size_t i = get_global_id(0);
    const size_t local_i = get_local_id(0);
    wg_sum[local_i] = 0.0;
    for (; i < array_size; i += get_global_size(0))
      wg_sum[local_i] += a[i] * b[i];

    for (int offset = get_local_size(0) / 2; offset > 0; offset /= 2)
    {
      barrier(CLK_LOCAL_MEM_FENCE);
      if (local_i < offset)
      {
        wg_sum[local_i] += wg_sum[local_i+offset];
      }
    }

    if (local_i == 0)
      sum[get_group_id(0)] = wg_sum[local_i];
  }
)CLC"};

template <class T>
T OCLStream<T>::dot()
{
  (*dot_kernel)(
    cl::EnqueueArgs(queue,
                    cl::NDRange(dot_num_groups*dot_wgsize),
                    cl::NDRange(dot_wgsize)),
    d_a, d_b, d_sum, cl::Local(sizeof(T) * dot_wgsize), array_size
  );
  cl::copy(queue, d_sum, sums.begin(), sums.end());

  T sum = 0.0;
  for (T val : sums)
    sum += val;

  return sum;
}
```

Figure 4: OpenCL dot kernel

```
template <typename T>
__global__ void triad_kernel(T * a, const T * b, const T * c)
{
  const T scalar = 0.4;
  const int i = blockDim.x * blockIdx.x + threadIdx.x;
  a[i] = b[i] + scalar * c[i];
}

template <class T>
void CUDAStream<T>::triad()
{
  triad_kernel<<<array_size/1024, 1024>>>(d_a, d_b, d_c);
  cudaDeviceSynchronize();
}
```

Figure 5: CUDA triad kernel

Table 1 List of devices

Name	Architecture	Class	Vendor	Theoretical Peak Memory BW (GB/s)
K20X	Kepler	GPU	NVIDIA	250
K40	Kepler	GPU	NVIDIA	288
K80 (1 GPU)	Kepler	GPU	NVIDIA	240
GTX 980 Ti	Maxwell	GPU	NVIDIA	224
Titan X	Pascal	GPU	NVIDIA	480
P100	Pascal	GPU	NVIDIA	732
S9150	Hawaii	GPU	AMD	320
Fury X	Fiji	GPU	AMD	512
E5-2670	Sandy Bridge	CPU	Intel	2×51.2=102.4
E5-2697 v2	Ivy Bridge	CPU	Intel	2×59.7=119.4
E5-2698 v3	Haswell	CPU	Intel	2×68=136
E5-2699 v4	Broadwell	CPU	Intel	2×76.8=153.6
Xeon Phi 7210	Knights Landing	MIC	Intel	∼5×102 = 510
Power 8 @ 3.69 GHz, 8 core	-	CPU	IBM	2×192=384

Table 2 Compiler configurations for GPUs

Model	K20X	K40	K80	GTX 980 Ti	Titan X	P100	S9150	Fury X
Driver	352.68	361.93.02	361.93.02	370.28	370.28	375.20	1912.5	1912.5
RAJA	GNU 5.3	GNU 4.9	GNU 4.9	GNU 4.9	GNU 4.9	GNU 5.3	n/a	n/a
Kokkos	GNU 5.3	GNU 4.9	GNU 4.9	GNU 4.9	GNU 4.9	GNU 5.3	n/a	n/a
OpenMP	CCE 8.5.5	CCE 8.5.5	CCE 8.5.5	clang-ykt	clang-ykt	CCE 8.5.5	n/a	n/a
OpenACC	PGI 16.10	CCE 8.5.5	CCE 8.5.5	PGI 16.7	PGI 16.7	PGI 16.9	PGI 16.7	n/a
CUDA	7.5	7.5	7.5	7.5	8.0	8.0	n/a	n/a
OpenCL	-	-	-	-	-	-	-	-
SYCL	ComputeCpp CE0.1.2							

Table 3 Compiler configurations for CPUs

Model	Sandy Bridge	Ivy Bridge	Haswell	Broadwell	KNL	Power 8
McCalpin	Intel 16.0	Intel 16.0	Intel 16.0	Intel 16.0	Intel 17.0	XL 13.1.4
RAJA	Intel 16.0	Intel 16.0	Intel 16.0	Intel 16.0	Intel 17.0	XL 13.1.4
Kokkos	Intel 16.0	Intel 16.0	Intel 16.0	Intel 16.0	Intel 17.0	XL 13.1.4
OpenMP	Intel 16.0	Intel 16.0	Intel 16.0	Intel 16.0	Intel 17.0	XL 13.1.4
OpenACC	PGI 16.10	PGI 16.10	PGI 16.10	PGI 16.10	PGI 16.10	XL 13.1.4
OpenCL	Intel 16.1	Intel 15.1	Intel 15.1	Intel 15.1	Intel 16.1.1	n/a
SYCL	ComputeCpp CE0.1.2					n/a

```
template <class T>
__global__ void dot_kernel(const T * a, const T * b, T * sum,
                           unsigned int array_size)
{

  extern __shared__ __align__(sizeof(T)) unsigned char smem[];
  T *tb_sum = reinterpret_cast<T*>(smem);

  int i = blockDim.x * blockIdx.x + threadIdx.x;
  const size_t local_i = threadIdx.x;

  tb_sum[local_i] = 0.0;
  for (; i < array_size; i += blockDim.x*gridDim.x)
    tb_sum[local_i] += a[i] * b[i];

  for (int offset = blockDim.x / 2; offset > 0; offset /= 2)
  {
    __syncthreads();
    if (local_i < offset)
    {
      tb_sum[local_i] += tb_sum[local_i+offset];
    }
  }

  if (local_i == 0)
    sum[blockIdx.x] = tb_sum[local_i];
}

template <class T>
T CUDAStream<T>::dot()
{
  dot_kernel<<<256, 1024, sizeof(T)*1024>>>(d_a, d_b, d_sum, array_size);
  check_error();

  cudaMemcpy(sums, d_sum, 256*sizeof(T), cudaMemcpyDeviceToHost);
  check_error();

  T sum = 0.0;
  for (int i = 0; i < 256; i++)
    sum += sums[i];

  return sum;
}
```

Figure 6: CUDA dot kernel

Table 4 Lines of code to implement class

Implementation	Lines of Code in Class	Difference
Serial (baseline)	119	0
CUDA	240	+121
OpenCL	319	+200
OpenACC	159	+40
OpenMP 4.5	209	+90
Kokkos	162	+43
RAJA	163	+44
SYCL	294	+175

```
template <class T>
void ACCStream<T>::triad()
{
  const T scalar = 0.4;

  unsigned int array_size = this->array_size;
  T * restrict a = this->a;
  T * restrict b = this->b;
  T * restrict c = this->c;
  #pragma acc kernels present(a[0:array_size], \
                              b[0:array_size], \
                              c[0:array_size]) wait
  for (int i = 0; i < array_size; i++)
  {
    a[i] = b[i] + scalar * c[i];
  }
}
```

Figure 7: OpenACC triad kernel

```
template <class T>
T ACCStream<T>::dot()
{
  T sum = 0.0;

  unsigned int array_size = this->array_size;
  T * restrict a = this->a;
  T * restrict b = this->b;
  #pragma acc kernels present(a[0:array_size], \
                              b[0:array_size]) wait
  for (int i = 0; i < array_size; i++)
  {
    sum += a[i] * b[i];
  }

  return sum;
}
```

Figure 8: OpenACC dot kernel

```
template <class T>
void OMPStream<T>::triad()
{
  const T scalar = 0.4;
  #pragma omp parallel for
  for (int i = 0; i < array_size; i++)
  {
    a[i] = b[i] + scalar * c[i];
  }
}
```

Figure 9: OpenMP triad kernel

```
template <class T>
T dot()
{
  T sum = 0.0;
  #pragma omp parallel for reduction(+:sum)
  for (int i = 0; i < array_size; i++)
    sum += a[i] * b[i];
}
```

Figure 10: OpenMP dot kernel

```
template <class T>
void OMPStream<T>::triad()
{
  const T scalar = 0.4;

  unsigned int array_size = this->array_size;
  T *a = this->a;
  T *b = this->b;
  T *c = this->c;
  #pragma omp target teams distribute parallel for simd \
    map(to: a[0:array_size], b[0:array_size], c[0:array_size])
  for (int i = 0; i < array_size; i++)
  {
    a[i] = b[i] + scalar * c[i];
  }
}
```

Figure 11: OpenMP 4 triad kernel

```
template <class T>
T dot()
{
  T sum = 0.0;

  unsigned int array_size = this->array_size;
  T *a = this->a;
  T *b = this->b;
  #pragma omp target teams distribute parallel for simd \
   map(to: a[0:array_size], b[0:array_size], tofrom: sum) \
   reduction(+:sum)
  for (int i = 0; i < array_size; i++)
    sum += a[i] * b[i];
}
```

Figure 12: OpenMP 4 dot kernel

```
template <class T>
void KOKKOSStream<T>::triad()
{
  View<double*, Kokkos::Cuda> a(*d_a);
  View<double*, Kokkos::Cuda> b(*d_b);
  View<double*, Kokkos::Cuda> c(*d_c);

  const T scalar = 0.4;
  parallel_for(array_size, KOKKOS_LAMBDA (const int index)
  {
    a[index] = b[index] + scalar*c[index];
  });
  Kokkos::fence();
}
```

Figure 13: Kokkos triad kernel

```
template <class T>
T KOKKOSStream<T>::dot()
{
  View<double *, DEVICE> a(*d_a);
  View<double *, DEVICE> b(*d_b);

  T sum = 0.0;

  parallel_reduce(array_size,
                  KOKKOS_LAMBDA (const int index, double &tmp)
  {
    tmp += a[index] * b[index];
  }, sum);

  return sum;

}
```

Figure 14: Kokkos dot kernel

```
template <class T>
void RAJAStream<T>::triad()
{
  T* a = d_a;
  T* b = d_b;
  T* c = d_c;
  const T scalar = 0.4;

  forall<policy>(index_set [=] RAJA_DEVICE (int index)
  {
    a[index] = b[index] + scalar*c[index];
  });
}
```

Figure 15: RAJA triad kernel

```
template <class T>
T RAJAStream<T>::dot()
{
  T* a = d_a;
  T* b = d_b;

  RAJA::ReduceSum<reduce_policy, T> sum(0.0);

  forall<policy>(index_set, [=] RAJA_DEVICE (int index)
  {
    sum += a[index] * b[index];
  });

  return T(sum);
}
```

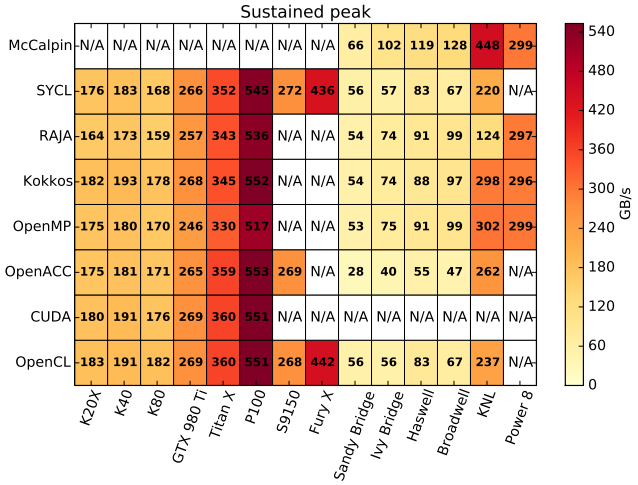
Figure 16: RAJA dot kernel

```
template <class T>
void SYCLStream<T>::triad()
{
  const T scalar = 0.4;
  queue->submit([&](handler &cgh)
  {
    auto ka = d_a->template get_access<access::mode::write>(cgh);
    auto kb = d_b->template get_access<access::mode::read>(cgh);
    auto kc = d_c->template get_access<access::mode::read>(cgh);
    cgh.parallel_for<triad_kernel>(p->get_kernel<triad_kernel>(),
      range<1>{array_size}, [=](item<1> item)
    {
      auto id = item.get()[0];
      ka[id] = kb[id] + scalar * kc[id];
    });
  });
  queue->wait();
}
```

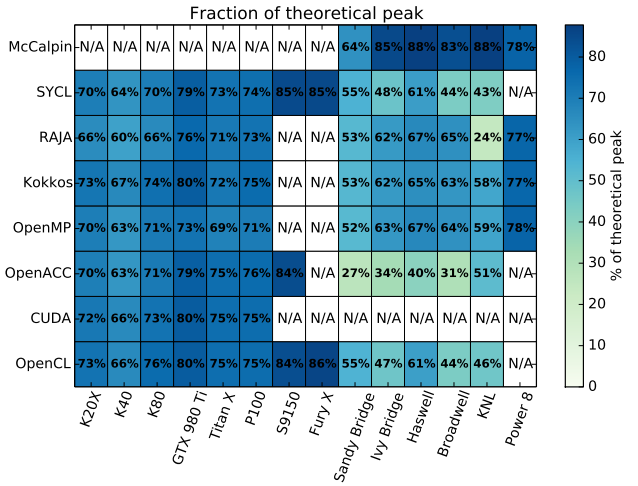
Figure 17: SYCL triad kernel

```
template <class T>
T SYCLStream<T>::dot()
{
  queue->submit([&](handler &cgh)
  {
    auto ka   = d_a->template get_access<access::mode::read>(cgh);
    auto kb   = d_b->template get_access<access::mode::read>(cgh);
    auto ksum = d_sum->template get_access<access::mode::write>(cgh);

    auto wg_sum =
      accessor<T, 1,
               access::mode::read_write,
               access::target::local>
       (range<1>(dot_wgsize), cgh);

    size_t N = array_size;

    cgh.parallel_for<dot_kernel>(p->get_kernel<dot_kernel>(),
      nd_range<1>(dot_num_groups*dot_wgsize, dot_wgsize), [=](nd_item<1> item)
    {
      size_t i = item.get_global(0);
      size_t li = item.get_local(0);
      size_t global_size = item.get_global_range()[0];

      wg_sum[li] = 0.0;
      for (; i < N; i += global_size)
        wg_sum[li] += ka[i] * kb[i];

      size_t local_size = item.get_local_range()[0];
      for (int offset = local_size / 2; offset > 0; offset /= 2)
      {
        item.barrier(cl::sycl::access::fence_space::local_space);
        if (li < offset)
          wg_sum[li] += wg_sum[li + offset];
      }

      if (li == 0)
        ksum[item.get_group(0)] = wg_sum[0];
    });
  });

  T sum = 0.0;
  auto h_sum = d_sum->template get_access<access::mode::read,
                                          access::target::host_buffer>();
  for (int i = 0; i < dot_num_groups; i++)
  {
    sum += h_sum[i];
  }

  return sum;
}
```

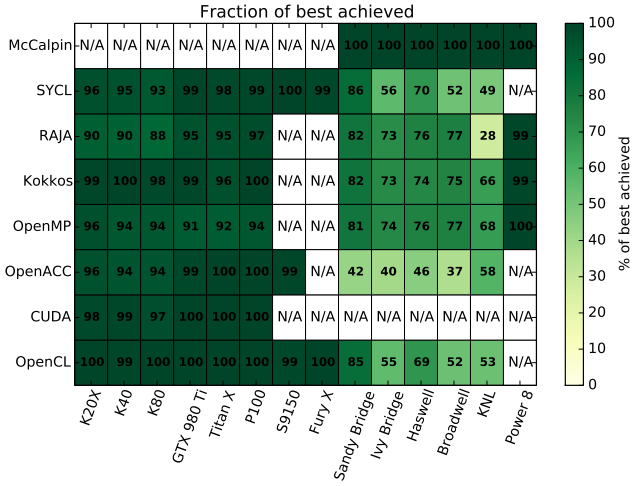
Figure 18: SYCL dot kernel



(a) Sustained peak

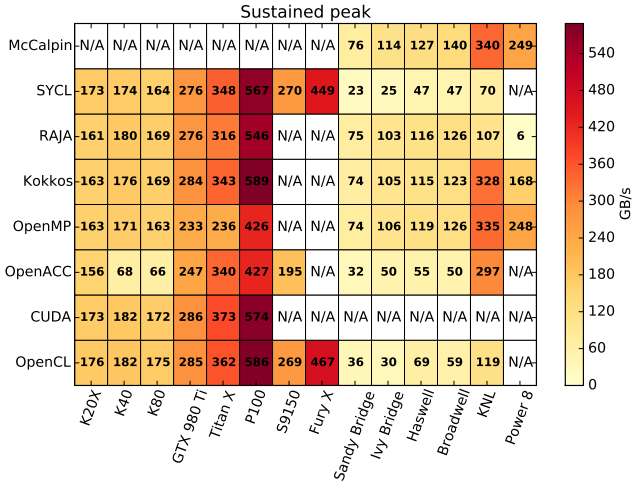


(b) Fraction of theoretical peak

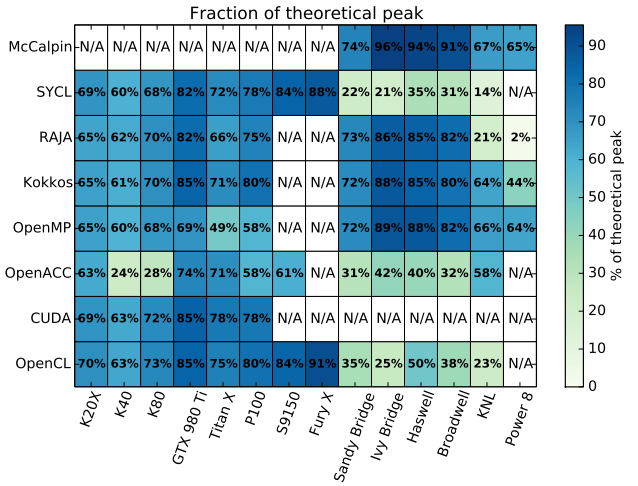


(c) Fraction of best achieved

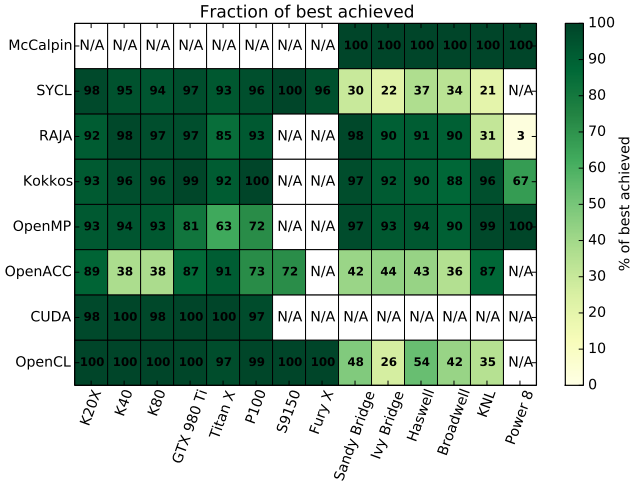
Figure 19: Performance of triad relative to theoretical peak memory bandwidth of BabelStream on 10 devices



(a) Sustained peak



(b) Fraction of theoretical peak



(c) Fraction of best achieved

Figure 20: Performance of dot relative to theoretical peak memory bandwidth of BabelStream on 10 devices