GPU-STREAM v2.0: Benchmarking the Achievable Memory Bandwidth of Many-Core Processors Across Diverse Parallel Programming Models

Tom Deakin(✉), James Price, Matt Martineau, and Simon McIntosh-Smith

Department of Computer Science, University of Bristol, Bristol, UK
`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.

The choice of one programming model over another should ideally not limit the performance that can be achieved on a device. 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.

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

M. Taufer et al. (Eds.): ISC High Performance Workshops 2016, LNCS 9945, pp. 489–507, 2016.
DOI: 10.1007/978-3-319-46079-6_34

bandwidth measure [10]. In this work we implemented the STREAM benchmark 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 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.

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

2 Measuring Memory Bandwidth

The STREAM Benchmark [10] 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:

1. Copy: $c[i] = a[i]$
2. Multiply: $b[i] = \alpha c[i]$
3. Add: $c[i] = a[i] + b[i]$
4. Triad: $a[i] = b[i] + \alpha c[i]$

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. Running the kernels in the order enumerated above ensures that any caches are invalidated between kernel calls; N is chosen to be large enough to require the data to be moved from main memory — see [10] for the rules of running STREAM. 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.

GPU-STREAM is a complementary benchmark to the standard CPU version of STREAM. GPU-STREAM enables the measurement of achievable memory bandwidth across a wide range of multi- and many-core devices [4]. The first version of GPU-STREAM 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. GPU-STREAM is Open Source and available on GitHub at github.com/UoB-HPC/GPU-STREAM. The webpage maintains a repository of all our results and we encourage submission of additional measurements. In this paper we expand GPU-STREAM to consider a second dimension to this reference point, namely the programming model.

2.1 Related Work

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 memory 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 [1].

The Standard Parallel Evaluation Corporation (SPEC) ACCEL benchmark suite whilst containing many memory bandwidth bound kernels does not include a STREAM kernel [16].

To the authors knowledge, the only study that has compared the same simple 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 [6,9].

3 Programming Models

A parallel programming model along with 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. This approach also helps to demonstrate the similarities and differences between these parallel programming

```
template <class T>
void triad()
{
  const T scalar = 3.0;
  for (int i = 0; i < array_size; i++)
    a[i] = b[i] + scalar * c[i];
}
```

Fig. 1. STREAM triad baseline kernel in C++

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.

The update to the GPU-STREAM 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 "host code" 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. 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.

3.1 OpenCL

OpenCL is an open standard, royalty-free API specified by Khronos [11]. 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 GPU-STREAM triad kernel in OpenCL C99 is shown in Fig. 2.

```
std::string kernels{R"CLC(
  constant TYPE scalar = 3.0;

  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();
}
```

Fig. 2. OpenCL triad kernel

3.2 CUDA

CUDA is a proprietary API from NVIDIA for targeting their GPU devices [12]. 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 upcoming Pascal GPUs, the GPU is allowed to cache data accessed from the host memory; previously it was zero copy.

The GPU-STREAM triad kernel is shown in Fig. 3. Note that CUDA requires specification of the number of threads per thread-block, therefore the size of the arrays must be divisible by 1024 in our implementation.

```
template <typename T>
__global__ void triad_kernel(T * a, const T * b, const T * c)
{
  const T scalar = 3.0;
  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();
}
```

Fig. 3. CUDA triad kernel

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 [13]. 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 shared amongst 'workers' on the device. The `kernels` pragma indicates that the region will be offloaded to the device as a series of 'kernels' and any loops encountered will 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 GPU-STREAM triad kernel is shown in Fig. 4. Note that a `wait` clause is required for the offload to be blocking as is required by our framework to ensure timing is correct. The `present` clause specifies that the memory is already available on the device and ensures a host/device copy is not initiated.

```
template <class T>
void ACCStream<T>::triad()
{
  const T scalar = 3.0;

  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];
  }
}
```

Fig. 4. OpenACC triad kernel

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 [14]. 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. 5.

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

```
template <class T>
void OMP3Stream<T>::triad()
{
  const T scalar = 3.0;
  #pragma omp parallel for
  for (int i = 0; i < array_size; i++)
  {
    a[i] = b[i] + scalar * c[i];
  }
}
```

Fig. 5. OpenMP triad kernel

```
template <class T>
void OMP45Stream<T>::triad()
{
  const T scalar = 0.3;

  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];
  }
}
```

Fig. 6. OpenMP v4 triad kernel

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 require the memory movement to the device must be written in our driver code, breaking the separation of implementation from driver code in our testing framework; OpenMP 4.5 fixes this issue.

The OpenMP 4 version of the GPU-STREAM triad kernel is shown in Fig. 6.

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 [5]. 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 GPU-STREAM triad kernel is shown in Fig. 7.

```
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 = 3.0;
  parallel_for(array_size, KOKKOS_LAMBDA (const int index)
  {
    a[index] = b[index] + scalar*c[index];
  });

  Kokkos::fence();
}
```

Fig. 7. Kokkos triad kernel

3.6 RAJA

RAJA is a recently released C++ abstraction layer developed by Lawrence Livermore National Laboratories that can target OpenMP and CUDA [7]. 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 RAJA version of the GPU-STREAM triad kernel is shown in Fig. 8.

```
template <class T>
void RAJAStream<T>::triad()
{
  T* a = d_a;
  T* b = d_b;
  T* c = d_c;
  const T scalar = 3.0;
  forall<policy>(index_set, [=] RAJA_DEVICE (int index)
  {
    a[index] = b[index] + scalar*c[index];
  });
}
```

Fig. 8. RAJA triad kernel

3.7 SYCL

SYCL is a royalty-free, cross-platform C++ abstraction layer from Khronos that builds on the OpenCL programming model (see Sect. 3.1) [8]. 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

```
template <class T>
void SYCLStream<T>::triad()
{
  const T scalar = 3.0;
  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<class triad>(nd_range<1>{array_size,
     ↪  WGSIZE}, [=](nd_item<1> item)
    {
      ka[item.get_global()] = kb[item.get_global()] + scalar *
       ↪  kc[item.get_global()];
    });
  });
  queue.wait();
}
```

Fig. 9. SYCL triad kernel

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 GPU-STREAM triad kernel is shown in Fig. 9.

4 Results

Table 1 lists the many-core devices that we used in our experiment. Given the breadth of devices and programming models we had to use a number of platforms and compilers to collect results. Intel do not formally publish peak MCDRAM bandwidth results for the Xeon Phi, so the presented figure is based on published claims that MCDRAM's peak memory bandwidth is five times that of KNL's DDR.

The HPC GPUs from NVIDIA were attached to a Cray XC40 supercomputer 'Swan' (K20X) and a Cray CS cluster 'Falcon' (K40 and K80). We used the GNU compilers (5.3 on Swan, 4.9 on Falcon) for Kokkos and RAJA results and the Cray compiler (8.5 on Swan, 8.4 on Falcon) for OpenMP and OpenACC results. The codes were built with CUDA 7.5.

The AMD GPUs were attached to an experimental cluster at the University of Bristol. We used the ComputeCpp compiler (2016.05 pre-release) from Codeplay [2] along with the AMD-APP OpenCL 1.2 (1912.5) drivers for SYCL results. We used the PGI Accelerator 16.4 for OpenACC on the AMD S9150 GPU.

The NVIDIA GTX 980 Ti is also attached to the University of Bristol experimental cluster (the "Zoo"). We used the clang-ykt fork of Clang for OpenMP[1]; note that the Clang OpenMP 4.x implementation is still under development and is not a stable release. We used PGI Accelerator 16.4 for OpenACC. We used CUDA 7.5 drivers for CUDA and OpenCL.

The Sandy Bridge CPUs are part of BlueCrystal Phase 3, part of the Advanced Computing Research Centre at the University of Bristol. Here we used the Intel 16.0 compiler for original STREAM and our C++ OpenMP implementation. RAJA and Kokkos were compiled using the GNU compilers. We used the PGI Accelerator 16.4 compiler for OpenACC and CUDA-x86. We used the Intel OpenCL Runtime 15.1 for OpenCL.

The Ivy Bridge CPUs are part of the experimental cluster at the University of Bristol. We used the GNU 4.8 compilers for RAJA and Kokkos and the Intel 16.0 compiler for original STREAM and our C++ OpenMP version. We used the ComputeCpp compiler from Codeplay along with the Intel OpenCL Runtime 15.1 for SYCL. We used the same OpenCL driver for OpenCL. We used the PGI Accelerator 16.4 compiler for OpenACC and CUDA-x86.

[1] https://github.com/clang-ykt.

The Haswell and Broadwell CPUs are part of a Cray XC40 supercomputer. We used the Cray compiler for original STREAM and our C++ OpenMP implementation. RAJA and Kokkos used the GNU compilers. We used the PGI 16.3 compiler for OpenACC for CUDA-x86.

The Intel Xeon Phi (Knights Landing) are part of the experimental cluster in Bristol. We used the Intel 2016 compiler for all results except OpenACC where we used the PGI compiler. Because the device in binary compatible with AVX2 architectures we specified Haswell as a target architecture for OpenACC.

We used the XL 13.1 compiler for all results on the Power 8.

Table 1. List of devices

Name	Class	Vendor	Peak memory BW (GB/s)
K20X	GPU	NVIDIA	250
K40	GPU	NVIDIA	288
K80 (1 GPU)	GPU	NVIDIA	240
GTX 980 Ti	GPU	NVIDIA	224
S9150	GPU	AMD	320
Fury X	GPU	AMD	512
E5-2670 (Sandy Bridge)	CPU	Intel	$2 \times 51.2 = 102.4$
E5-2697 v2 (Ivy Bridge)	CPU	Intel	$2 \times 59.7 = 119.4$
E5-2698 v3 (Haswell)	CPU	Intel	$2 \times 68 = 136$
E5-2699 v4 (Broadwell)	CPU	Intel	$2 \times 76.8 = 153.6$
Xeon Phi (Knights Landing) 7210	MIC	Intel	$\sim 5 \times 102 = 510$
Power 8	CPU	IBM	$2 \times 192 = 384$

In the next few sections we describe our experiences in porting the GPU-STREAM kernels to the seven different parallel programming models in our study, before describing the performance we were able to achieve when running these implementations on a diverse range of 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 is no longer is expressed as a loop. Table 2 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 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 OpenMP version in C++ implemented in our framework.

Table 2. Lines of code to implement class

Implementation	Lines of code in class	Difference
OpenMP 3 (baseline)	113	0
CUDA	183	+70
OpenCL	229	+116
OpenACC	138	+25
OpenMP 4.5	138	+25
Kokkos	150	+37
RAJA	144	+31
SYCL	145	+32

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, major 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.

The OpenMP 4.0 programming model does not allow for control of the data on the target device in an unstructured way. The data on the device is controlled by scoping rules in the code, and as such an OpenMP 4.0 implementation required an invasive procedure to add this to our code, breaking our abstraction of model from control and timing code. OpenMP 4.5 addresses this issue with `target data enter` and `exit` regions, however OpenMP 4.5 compilers were not available on all platforms at the time of writing so we had to use both 4.0 and 4.5 versions to collect the range of results.

We had to remove the `simd` clause from the OpenMP directives to achieve good performance with Clang, and use a static schedule with a chunk size of one (which can be specified via an environment variable). These changes render the code non-portable, however once OpenMP offload support becomes mature these will not be required.

Our experience of the disruption of OpenMP 4 is more related to availability of tools over issues with the model itself.

OpenACC using the PGI compiler targeting host CPUs, the AMD GPUs and the NVIDIA GTX 980 Ti, all required specifying the pointers as `restrict` in order for the loop to be parallelised, although this is not standard C++. Using `parallel loop independent` does parallelise the loop without specifying `restrict`. This was a 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. 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 device.

However, all compilers supporting OpenMP 4 and OpenACC would not correctly offload the kernel without re-declaring the arrays as local scope variables. 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.

The PGI compiler has support for CUDA-x86 whereby CUDA code can target host CPUs. All kernel calls are considered blocking unlike the CUDA API itself, and the `cudaSynchronizeDevice()` call is not supported; as such we had to remove this from the code, reducing the portability of the CUDA code. Additionally the compiler failed to build the code with templated classes.

In SYCL, explicitly choosing the size of a work-group was required to achieve good performance. As the programming model does not stipulate that this size must be set, this requirement is likely to disappear with future updates to the compiler. This is similar to OpenCL's ability to leave the choice of work-group size up to the run-time.

In addition to these code changes, despite trying to use a unified build system (CMake), many of the data points required specific compiler invocations.

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 in Fig. 10. We used 100 iterations with an array size of 2^{25} double precision elements (268 MB).

When writing code targeting NVIDIA GPUs, the results with all the programming models are similar. Both the high-level models, such as RAJA and Kokkos, and the directives based approaches, such as OpenMP and OpenACC, demonstrate equivalent performance to CUDA and OpenCL on these devices, which is a very encouraging result.

When targeting AMD GPUs however, we are unable to collect a full set of data points because neither RAJA nor Kokkos provide GPU implementations of their model to run on non-NVIDIA 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; note that RAJA and Kokkos use OpenMP for their CPU implementation and CUDA for the NVIDIA GPU implementation. 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 (83–86 %).

We use the original 'McCalpin' STREAM benchmark written in C and OpenMP as a baseline comparison for the CPU results. Thread binding is used

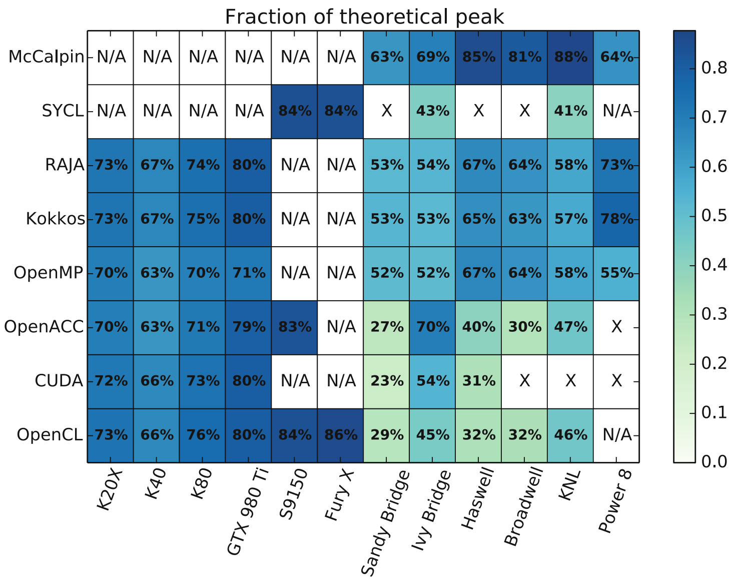


Fig. 10. Performance relative to theoretical peak of GPU-STREAM on 10 devices

via the OpenMP implementation selecting a compact affinity. We did not experiment with streaming stores. Figure 10 shows that there is some loss of performance when using C++ and OpenMP 3 for running on CPUs compared to the much simpler C language version. 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.

We used PGI's implementation of OpenACC for multi-core CPUs. On the Sandy Bridge and Ivy Bridge system we used the `numactl` tool for thread pinning and specified the `ACC_NUM_CORES` environment variable to the total number of cores in our dual-socket CPU systems. The Haswell and Broadwell CPUs are in a Cray system so were required to use the options within `aprun` to run the binary. Despite this however it does not demonstrate good peak performance on the CPUs in general. For the Xeon Phi we needed to use `MP_BLIST` to pin the OpenACC threads as `numactl` did not pin these threads correctly.

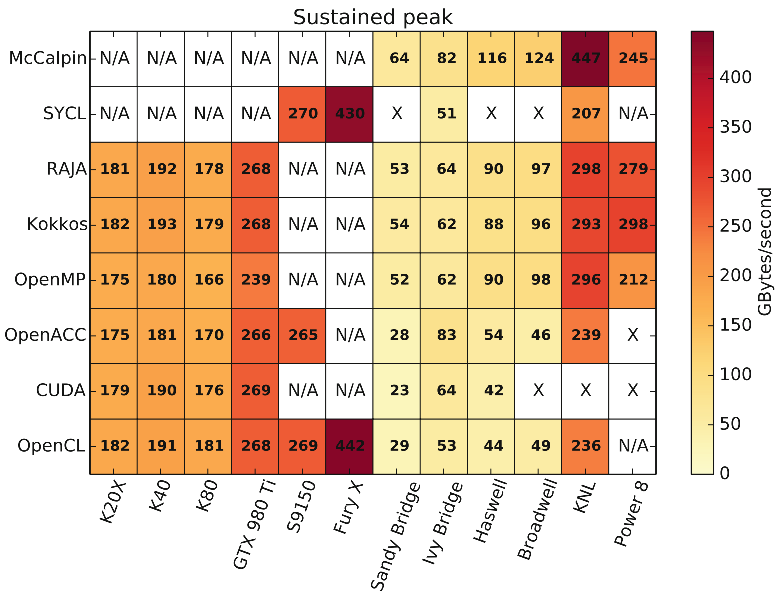
Both RAJA and Kokkos use the OpenMP programming model to implement parallel execution on CPUs. The performance results on all four CPUs tested show that RAJA and Kokkos performance matches that of hand-written OpenMP for GPU-STREAM. This result shows that both RAJA and Kokkos provide little overhead over writing OpenMP code directly, at least for GPU-STREAM. 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.

OpenCL is able to run on the CPU as well, and we tested using the Intel OpenCL runtime. This is implemented on top of Intel's Thread Building Blocks, which results in non-deterministic thread placement. In a dual-socket system the placement of threads based on memory allocations (first touch) is important in achieving good bandwidth; as such this programming model suffers in performance on Intel CPUs compared to the original STREAM code. The PGI CUDA-x86 compiler gets similar performance to OpenCL, but they are both lower than OpenMP.

Figure 11 shows the raw sustained memory bandwidth of the triad kernel in each case. 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 AMD HPC GPU from AMD, the S9150, provides an increased bandwidth over NVIDIA's HPC offerings.

The Intel Xeon Phi (KNL) had the highest achieved memory bandwidth, however this performance was not achieved in all programming models. In general we ran one thread per core and this achieved the highest performance, but Kokkos needed two threads per core to achieve comparable performance.

The Power 8 results were collected with one thread per core. The bandwidth presented is using the same problem size as all the other results; a high bandwidth is possible with a large problem. It has been previously observed that performance can decrease with smaller problems [15].



	K20X	K40	K80	GTX 980 Ti	S9150	Fury X	Sandy Bridge	Ivy Bridge	Haswell	Broadwell	KNL	Power 8
McCalpin	N/A	N/A	N/A	N/A	N/A	N/A	64	82	116	124	447	245
SYCL	N/A	N/A	N/A	N/A	270	430	X	51	X	X	207	N/A
RAJA	181	192	178	268	N/A	N/A	53	64	90	97	298	279
Kokkos	182	193	179	268	N/A	N/A	54	62	88	96	293	298
OpenMP	175	180	166	239	N/A	N/A	52	62	90	98	296	212
OpenACC	175	181	170	266	265	N/A	28	83	54	46	239	X
CUDA	179	190	176	269	N/A	N/A	23	64	42	X	X	X
OpenCL	182	191	181	268	269	442	29	53	44	49	236	N/A

Fig. 11. Sustained memory bandwidth of GPU-STREAM on 10 devices

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. 10 and 11 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', and those we were unable to obtain due to difficulties with combinations of installed libraries and platforms with an 'X'. Note that the original STREAM benchmark from McCalpin was written in C with OpenMP 3 and so cannot run on GPUs.

For SYCL, ComputeCpp generates SPIR to be consumed by an OpenCL runtime. NVIDIA's OpenCL implementation does not currently support SPIR. ComputeCpp is not currently compatible with the systems which housed the Sandy Bridge, Haswell, and Broadwell CPUs.

OpenMP and OpenACC are supported by a variety of compilers to varying degrees. It is therefore simpler to discuss the available options for devices. 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. The PGI Accelerator compiler supports AMD GPUs up to Hawaii so we used this for the S9150, but the Fury X's newer Fiji architecture is not yet supported.

The Cray compiler supports OpenMP targeting NVIDIA GPUs, with version 8.5 supporting OpenMP 4.5 and version 8.4 supporting OpenMP 4.0. We were able to use version 8.5 to collect results for the NVIDIA K20X with OpenMP 4.5 code, and had to resort to our OpenMP 4.0 code with Cray compiler version 8.4 for the K40 and K80 GPUs. The GTX 980 Ti GPU was not on a Cray system so we could not collect a result for it using the Cray compiler. However, Clang supports OpenMP 4.5 targeting NVIDIA GPUs, and so the result was obtained using this compiler.

The PGI CUDA-x86 compiler did compile the code for Broadwell and KNL but failed at runtime due to the number of available threads being unsupported. The PGI compiler was also unavailable on the Power 8 system so we were unable to collect OpenACC and CUDA results.

5 Conclusion

What is evident from Fig. 10 is that, in general, the more mature of these programming models provide better performance across the range of architectures. 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 directive based approaches of OpenMP and OpenACC look to provide a good trade off between performance and code complexity. OpenACC demonstrates good GPU performance on products from both NVIDIA and AMD, however the CPU performance is poor. This limits OpenACC's relevance to CPUs due to implementations of the model at the time of writing.

With the directive based approaches of OpenMP and OpenACC the number of lines of code to add to an existing piece of C or Fortran code is minimal. If code is already in C++, then SYCL, RAJA and Kokkos provided a similar level of minimal disruption for performance.

Acknowledgements. We would like to thank Cray Inc. for providing access to the Cray XC40 supercomputer, Swan, and the Cray CS cluster, Falcon. Our thanks to Codeplay for access to the ComputeCpp SYCL compiler and to Douglas Miles at PGI (NVIDIA) for access to the PGI compiler. We would also like to that 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. Bhat, K.: clpeak (2015). https://github.com/krrishnarraj/clpeak
2. Codeplay: ComputeCpp. https://www.codeplay.com/products/computecpp
3. Danalis, A., Marin, G., McCurdy, C., Meredith, J.S., Roth, P.C., Spafford, K., Tipparaju, V., Vetter, J.S.: The scalable heterogeneous computing (SHOC) benchmark suite. In: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, GPGPU-3, pp. 63–74. ACM, New York (2010). http://doi.acm.org/10.1145/1735688.1735702
4. Deakin, T., McIntosh-Smith, S.: GPU-STREAM: benchmarking the achievable memory bandwidth of graphics processing units (poster). In: Supercomputing, Austin, Texas (2015)
5. Edwards, H.C., Sunderland, D.: Kokkos array performance-portable manycore programming model. In: Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM 2012), pp. 1–10. ACM (2012)
6. Heroux, M., Doerfler, D., et al.: Improving performance via mini-applications. Technical report, SAND2009-5574, Sandia National Laboratories (2009)
7. Hornung, R.D., Keasler, J.A.: The RAJA Portability Layer: Overview and Status (2014)
8. Khronos OpenCL Working Group SYCL subgroup: SYCL Provisional Specification (2016)
9. Martineau, M., McIntosh-Smith, S., Boulton, M., Gaudin, W.: An evaluation of emerging many-core parallel programming models. In: Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycore, PMAM 2016, pp. 1–10. ACM, New York (2016). http://doi.acm.org/10.1145/2883404.2883420
10. McCalpin, J.D.: Memory bandwidth and machine balance in current high performance computers. IEEE Comput. Soc. Tech. Comm. Comput. Archit. (TCCA) Newslett. 19–25 (1995)
11. Munshi, A.: The OpenCL Specification, Version 1.1 (2011)
12. NVIDIA: CUDA Toolkit 7.5
13. OpenACC-Standard.org: The OpenACC Application Programming Interface - Version 2.5 (2015)

14. OpenMP Architecture Review Board: OpenMP Application Program Interface, Version 4.5 (2015)
15. Reguly, I.Z., Keita, A.K., Giles, M.B.: Benchmarking the IBM Power8 processor. In: Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering, pp. 61–69. IBM Corporation, Riverton (2015)
16. Standard Performance Evaluation Corporation: SPEC Accel (2016). https://www.spec.org/accel/