Effective Host-GPU Memory Management through Code Generation

Effective Host-GPU Memory Management Through Code Generation

Hans-Nikolai Vießmann
Heriot-Watt University
Edinburgh, UK
Radboud University
Nijmegen, Netherlands
h.viessmann@ru.nl

Sven-Bodo Scholz
Heriot-Watt University
Edinburgh, UK
Radboud University
Nijmegen, Netherlands
svenbodo.scholz@ru.nl

ABSTRACT

NVIDIA's CUDA provides several options to orchestrate the management of host and device memory as well as the communication between them. In this paper we look at these choices, identify the program changes required when switching between them, and we observe their effect on application performance.

We present code generation schemes that translate resource-agnostic program specifications, *i.e.*, programs without any explicit notion of memory or GPU kernels, into five CUDA versions that differ in the use of the memory and communication API of CUDA only. An implementation of these code generators within the compiler of the functional programming language Single-Assignment C (SaC) shows performance differences between the variants by up to a factor of 3.

Performance analyses reveal that the preferred choices depend on a combination of several factors, including the actual hardware being used, and several aspects of the application itself. A clear choice, therefore, cannot be made *a priori*. Instead, it seems essential that different variants can be generated from a single source for achieving performance portability across GPU devices.

CCS CONCEPTS

• **Computing methodologies** → *Parallel programming languages*; • **Theory of computation** → Semantics and reasoning; • **Software and its engineering** → **Compilers**.

KEYWORDS

code generation, GPU, CUDA, SaC, communication models, transfer bandwidth, memory management

ACM Reference Format:
Hans-Nikolai Vießmann and Sven-Bodo Scholz. 2020. Effective Host-GPU Memory Management Through Code Generation. In *IFL 2020: Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages (IFL '20), September 2–4, 2020, Canterbury, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3462172.3462199

IFL '20, September 2–4, 2020, Canterbury, United Kingdom

ACM ISBN 978-1-4503-8963-1/20/09.
https://doi.org/10.1145/3462172.3462199

1 INTRODUCTION

NVIDIA's CUDA framework and CUDA-compatible GPUs are a *de facto* standard for most GPU-based computations. The favourable performance to price ratio of GPUs combined with their suitability for many data intensive applications has led to very rapid evolution of new GPU hardware. Besides improvements in the GPU designs themselves, particular effort has been spent on improving the management of data transfers between hosts and devices. These, in turn, have led to extensions in the CUDA programming model. While such extensions typically allow for better utilisation of new hardware features, they pose challenges to code portability and code maintenance. Some of the newer features are not supported for older hardware, others introduce a vast overhead. Even if code is specifically constructed to be used on one particular hardware device, figuring out which part of the CUDA API is most suitable for a given task is not easy[1].

This need for architecture-specific tuning and its quickly evolving, volatile nature suggests that generative programming can provide the desired application stability while reducing the burden of rewrites for performance portability to a minimum. Indeed, several approaches exist [12, 14, 16, 23, 26] that demonstrate how systematic code rewrites can substantially improve GPU performance using either annotations, heuristics, or machine learning to guide the rewriting process. While the pre-existing work mainly focuses on kernel construction and interplay, this paper is concerned with the memory management on host and device as well as the orchestration of the communication between them after the kernels have been decided upon. CUDA 10.1 offers several mechanisms to manage memory and to orchestrate data transfers between different contexts:

- Memory on host and device can be allocated separately or in a unified fashion;
- Host memory allocations can be done through the operating system or CUDA itself;
- Transfers can be made synchronously or asynchronously;
- Depending on the choices above, transfers are explicit or can be triggered implicitly; synchronisations are implicit or may need to be inserted explicitly into the code.

The choices between these options depend on the capabilities of the executing architecture as well as the characteristics of a given application.

[1] NVIDIA's documentation at https://docs.nvidia.com/cuda/ provides different tuning guides for the latest five different NVIDIA architectures.

In this paper, we present our investigation of how best to leverage these options when compiling a functional language into GPU kernels. Our main contributions are:

- An overview of the memory allocation and communication mechanisms currently available in CUDA and how they need to be orchestrated to avoid race conditions.
- A bandwidth comparison between CUDA's memory transfer methods for different GPUs. Our comparison shows that there are differences of up to a factor of 2 in bandwidth between transfer methods. The comparison also shows that the bandwidth depends on the hardware that is being used.
- A code generation scheme that enables the generation of five different CUDA code variants from a single source program. This includes provisions for safely overlapping GPU activity on the device and CPU code execution on the host in the presence of asynchronous communication between host and device and asynchronous kernel launches.
- An extension of the memory allocation optimisation named Extended Memory Reuse (*EMR*) [27] which proves vital for some of the transfer schemes.
- A performance analysis of the different versions based on a full-fledged implementation in the context of the Single-Assignment C (SaC) compiler.

The paper is organised as follows, Section 2 describes CUDA and the different communication models it supports. Section 3 presents an analysis of a bandwidth comparison between these communication models. Section 4 introduces the SaC compiler and programming language and describes how we generate CUDA code. Section 5 describes our code transformations for generating code for the different CUDA transfer methods. Section 6 describes the extensions of *EMR* needed for the performance analysis in Section 7. Finally, Sections 8 and 9 discuss related works and concluding remarks.

2 CUDA AND ITS MEMORY MANAGEMENT

NVIDIA's CUDA is a software framework and driver to implement and run applications on NVIDIA's GPU devices. Like other many-core architectures, GPUs build on the following design: the GPU device has its own memory, data needs to be transferred between the CPU-based host system and the device and back. Computations on the GPU are captured in so-called *kernels*. CUDA provides various API's to interact with the GPU device. Here, we only introduce those variants relevant to the present work. For a full account, we refer the reader to the most recent CUDA manual [22].

CUDA kernels are always launched from the host and are then executed asynchronously on the GPU [22]. For that purpose, the GPU has its own scheduler that non-preemptively executes the kernel [15].

In practice, there are three communication models provided by CUDA, synchronous communication, asynchronous communication, and managed communication. A simple example serves as canonical example for most GPU code to explain the differences between these communication models. Listing 1 presents our canonical example using synchronous communication.

In lines 1–4 we define a kernel function `increment_kernel` which increments an argument array `d_va` in an element-wise fashion. The

```
1  __global__ void increment_kernel(int *d_va) {
2    int i = blockIdx.x * blockDim.x + threadIdx.x;
3    d_va[i] = d_va[i] + 1;
4  }
5
6  int main() {
7    int *va, *d_va;
8    va = (int *)malloc(1024*sizeof(int));
9    cudaMalloc(&d_va, 1024*sizeof(int));
10   ...
11   cudaMemcpy(d_va, va, 1024*sizeof(int),
12                        cudaMemcpyHostToDevice);
13   ...
14   increment_kernel <<<16, 64>>> (d_va);
15   ...
16   cudaMemcpy(va, d_va, 1024*sizeof(int),
17                        cudaMemcpyDeviceToHost);
18   ...
19   cudaFree(d_va);
20   free(va);
21   return 0;
22 }
```

Listing 1: CUDA Code with Synchronous Communication

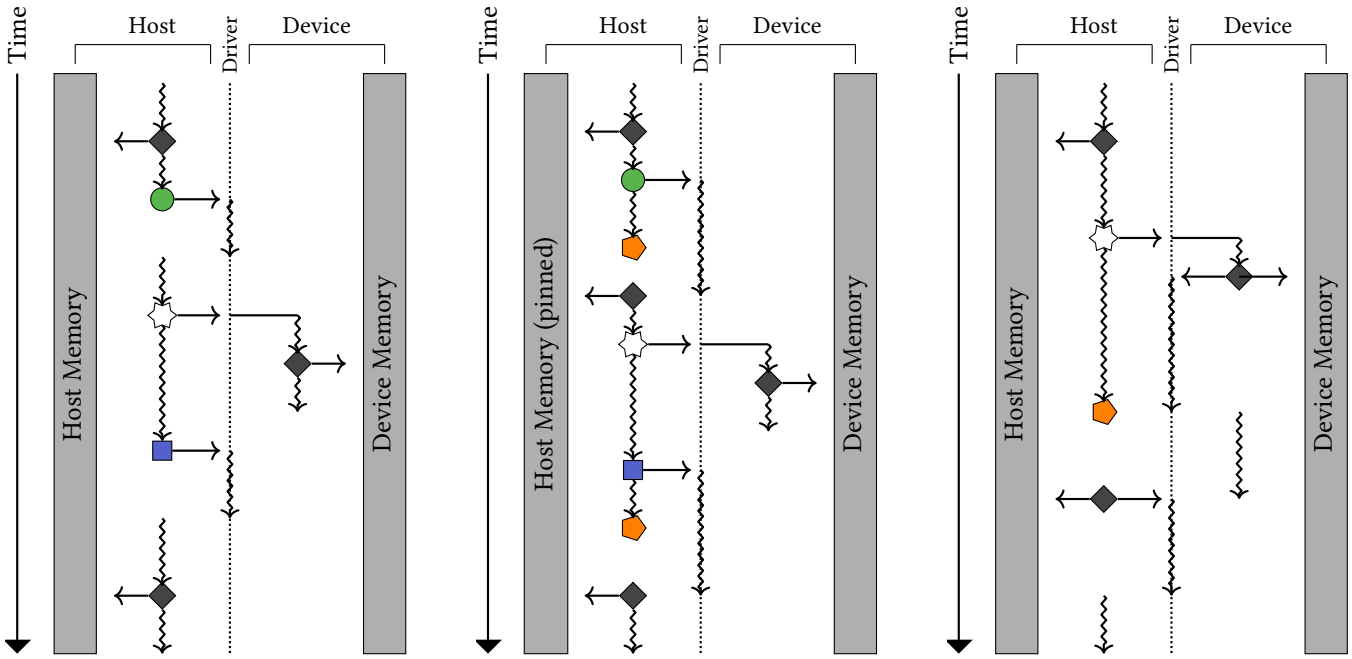
`main` function essentially consists of five phases: memory allocation (lines 8 and 9) for the host and the device, transfer of the kernel argument from host-to-device (lines 11–12), kernel invocation (line 14), transfer of the result back from the device to the host (lines 16–17) and finally memory de-allocation in lines 19 and 20. We assume further host code to exist between these phases as indicated by ellipses. This omitted code performs the actual host operations including the initialisation of the host array and the interpretation of results that have come back from the GPU device. Since we are only interested in memory management and communication, we omit these particulars.

Figure 1 provides a comparison of how our canonical example is executed using the different memory and transfer options of NVIDIA's CUDA. For each model, we demonstrate how host and device interact over time. The time axis is vertical; to follow the flow of time we have to follow the diagrams from top to bottom. Handshakes between host and device are indicated by horizontal arrows. The host and device regions are bridged by the driver, which acts as the middle layer for communications and kernel launches. We do not show allocations, only accesses to memory. In the following text we discuss each model and relate them to the canonical example from Listing 1, including looking at the required code changes for implementing different communication models.

2.1 Synchronous Communication

In Figure 1a, we show the timeline of events when running our code example with synchronous transfers. We assume that memory has been allocated on both the host and device.

The first operation indicated in the diagram by a black rhombus with an arrow to the left is an access of the CPU to the host memory.



(a) Synchronous Model (b) Asynchronous Model (c) Managed Model

Figure 1: CUDA Communication Models: Shown are the operations that happen between host and device for three communications models available through CUDA. For each model, we use the following symbols to indicate memory accesses: ◆, synchronisations: ⬟, kernel launches: ☆, host-to-device transfers (*H2D*): ●, and device-to-host transfers (*D2H*): ■. Solid arrows indicate handshake events, while squiggly arrows indicate computation. Gaps indicate waiting between events.

Thereafter, data is transferred from the host-to-the device using `cudaMemcpy`. The direction of the transfer is defined through the parameter *cudaMemcpyHostToDevice*. In the diagram, this operation is symbolised by the green filled circle. It transfers control to the device driver which performs the transfer indicated by the squiggly line on the driver. During this operation, the CPU is blocked as indicated by the gap in the host line. Once the transfer has been completed, the host resumes control and continues executing.

In our example, the CPU continues doing some work before the next GPU relevant action is triggered: the kernel launch, shown by the white star symbol in the host line. The kernel launch is the only host device interaction of asynchronous nature here. Since both, the CPU on the host and the GPU on the device operate on separate memories, no further synchronisation is necessary until the results of the kernel execution are needed, *i.e.*, the host can execute work completely independent of the kernel execution on the device.

When the host eventually calls `cudaMemcpy` to transfer the data back (indicated by a blue square in the diagram), this causes a synchronisation of the two activities: the host waits for the kernel and communication to complete before doing more host side work. Finally, we de-allocate the host and device memories.

Given the tight synchronisation of the synchronous model, any possible latency hiding that could be gained from direct memory access (DMA) capabilities of modern GPUs can not be leveraged here.

2.2 Asynchronous Communication

Figure 1b shows the timeline of events for the canonical example when using asynchronous communication, which allows for overlapping host and device transfers with further operations on the host: after the CPU has initiated the host-to-device transfer (green circle) the host can proceed executing further code while the GPU performs DMA operations to execute the actual transfer. On the software side, CUDA implements this through the introduction of a queue-like structure into which device-related operations, like memory transfers and kernel launches, can be staged. This so-called *stream* moves the scheduling of device-related operations away from the host application and into the CUDA driver. In Figure 1b, we represent this again by a squiggly line on the driver.

Allowing the CPU to proceed while the transfer is happening does come at a risk though. Changing the memory that is being transferred while the transfer is taking place would introduce a race condition. To avoid this, we need to make sure that the transfer has been completed before modifying the memory that is being transferred. We indicate such a situation in Figure 1b. Before the CPU attempts to write into the memory to be transferred (indicated by the black rhombus) we insert an explicit synchronisation operation into the host code (indicated by the orange pentagon). The synchronisation suspends CPU execution until the transfer is complete and thus prevents a race condition from occurring.

In the source code, this means that apart from changing **cudaMemcpy** into **cudaAsyncMemcpy** in lines 11 and 16 of our canonical example in Listing 1, we also need to inject a call to **cudaDeviceSynchronize** should the host code between the transfer and the kernel launch contain instructions that change the memory behind the host pointer va.

In Figure 1b, we demonstrate a similar situation on the transfer back from the device to the host. Again, we inject an explicit synchronisation (orange pentagon) before executing a memory access to the expected result (black rhombus).

Besides changing the transfer function and potentially injecting explicit synchronisations, the asynchronous model also requires extra provisions for the host memory to enable DMA transfers: paging of the host memory needs to be inhibited and the CUDA driver needs to be notified about that. CUDA offers two ways of achieving this: CUDA host allocation and CUDA host registration.

CUDA Host Allocator. CUDA offers **cudaHostAlloc** as a possible replacement for the system `malloc`. By using it, we allocate host memory and mark it as page-locked or *pinned*, preventing it from being swapped out. Additionally, the memory is registered with the CUDA driver, *i.e.*, the driver is made aware of the fact that this address points to pinned memory. Consequently, the pointer can be passed directly to **cudaAsyncMemcpy**. As the pointer is handled by CUDA, we can only free it by using **cudaFreeHost**. Modifying our code example, we replace our `malloc` on line 8 and the `free` in line 20 as follows:

```
8  cudaHostAlloc(&va,1024*sizeof(int),cudaHostAllocDefault);
|  ...
20 cudaFreeHost(va);
```

The pointer to the allocated memory is returned through the first parameter of the function **cudaHostAlloc**, which is followed by the number of bytes to allocate.

While the combined functionality of this allocator is programmer friendly, it requires immediate allocation of physical memory in order to enable the pinning. This increases the latency of allocating memory. While this may be acceptable for memory whose content is ultimately transferred to the GPU, it affects all host memory allocations, including those of memory that never will be transferred to the GPU.

CUDA Host Register. Instead of using CUDA's host allocator, we can manually pin memory allocated by the system allocator and register it with the CUDA driver. The effect is identical to using CUDA's allocator, but provides one key advantage — we can delay the pinning of the memory. Furthermore, as the operation itself does not allocate physical memory, we can leverage the system allocator's delayed allocation, reducing the overheads that happen using CUDA's allocator. Staying with our example code, pinning by calling **cudaHostRegister** can be done at any point between the initial allocation and the transfer call. In our example we do this directly after `malloc`, and we unpin the memory after the last transfer. The register function passes the allocated pointer and a flag indicating what properties the returned pointer should have. The *cudaHostRegisterDefault* flag ensures that the pointer is treated the same in all contexts. This leads to the following changes of our example from Listing 1:

```
8  va = (int *)malloc(1024*sizeof(int));
9  cudaHostRegister(va, cudaHostRegisterDefault);
|  ...
20 cudaHostUnregister(va);
21 free(va);
```

Notice that we use the system's `free` function when de-allocating the memory, this can only happen after the pointer has been unpinned.

2.3 CUDA Managed Memory

With CUDA version 4.0, the memories of the host system and GPU device were combined into a single virtual address space, called Unified Virtual Addressing (UVA). This allows for pointers created by the CUDA API to be used on both the host and the device. Additionally, the concept of zero-copy memory was introduced, which allows the GPU to access pinned host memory without an explicit transfer operation.

Later in CUDA version 6.0, UVA was extended by the unified memory (UM) model, which introduced the concept of *managed* memory [21]. Managed memory eliminates the idea of two explicit memories and explicit transfers between them. Memory on host and device is allocated at the same time using a single call to a CUDA-specific memory allocator, **cudaMallocManaged** which allocates memory through UM. The resulting pointer is tracked and if it is accessed from a non-local context (*e.g.* GPU device accessing host memory), the data is transferred implicitly.

Depending on what version of CUDA is used, and even what generation of CUDA device is used, the underlying behaviour of UM can vary. For versions of CUDA older than 8.0, and CUDA devices architectures before Pascal, the implicit transfer of data happens as part of the kernel launch, where the entire memory associated with a managed pointer is transferred. Because of this, explicit synchronisations after the kernel launch are needed to keep the view of memory consistent in all contexts.

In versions of CUDA after 8.0, and device architectures like Pascal and newer, the transfers are initiated by paging on demand. Here an access to host-based memory from the GPU device causes a page-fault, which the CUDA driver reacts to by sending the missing page. The driver actually sends several consecutive pages, in varying quantities, whenever a page fault occurs [20]. Host-side accesses to data located in device memory are resolved implicitly by the CUDA driver without an explicit synchronisation.

We depict the latter behaviour in the timeline for the canonical example when using the managed model in Figure 1c. The most obvious change is that both explicit transfers (green circle and blue square) are gone. The transfers to the GPU are triggered after the kernel launch, whenever the GPU tries to access memory. We indicate this in the diagram by the squiggly line on the driver and the memory accesses on the GPU by the black rhombus. As in the asynchronous model, the managed model requires a synchronisation before the CPU can write to the host memory used for the arguments of the kernel, and another one before it can start reading the results to be computed by the GPU in the device memory. In Figure 1c, this is shown as the orange pentagon again, followed by the CPU waiting. Once the kernel has completed, the CPU can start

reading again which implicitly triggers transfers from the GPU back to the host, indicated by the black rhombus on the CPU side.

To implement the managed model in our example, we replace both memory allocations by a single call to **cudaMallocManaged**. Additionally, we can remove the device memory allocations from line 9 as we no longer have a notion of host or device memory. In this way we update the parameter of our kernel to be va. We also remove the explicit transfer operations and insert a synchronisation before the results are being used, resulting in the following code:

```
9   cudaMallocManaged(&va, 1024*sizeof(int),
10                    cudaMemAttachGlobal);
|   ...
14  increment_kernel <<<16, 64>>> (va);
15  ...
16  cudaDeviceSynchronize();
|   ...
19  cudaFree(va);
```

One more aspect worth mentioning here is that the use of a unified view on the pointers va and va_dev has consequences if the canonical example makes further use of va while the data resides on the device. In the asynchronous model we already noticed that in such cases additional synchronisation is required to ensure that the data has been transferred completely to the GPU device. In the managed model, such a synchronisation is not possible at all as there is no way to enforce the data to reside on either the host or the GPU device only. If such a case arises, separate host memory needs to be allocated and the data of va may need to be copied. In our example, this implies that, depending on the code in the ellipses, further changes of the code may be required.

Memory Prefetch. Provided that data from all memory pages is being needed on both the host and the device, the UM system's reliance on demand-driven transfers makes it less efficient in comparison to the explicit communication of the other communication models. The added latency can be avoided by explicitly prefetching the data. CUDA provides the function **cudaMemPrefetchAsync** to trigger prefetching. In our example, calls to **cudaMemPrefetchAsync** can be inserted in those positions where the explicit transfers in the original example are placed.

2.4 Summary

With these different CUDA host communication models, we identify five distinct methods for performing transfers: (1) synchronous communication, (2) asynchronous communications using host allocation (which implicitly pins memory), (3) asynchronous communications with separately registered host memory, (4) implicit communication using CUDA managed memory and, finally, (5) implicit communication with explicit prefetch. For the rest of this paper we will refer to these respectively as *sync*, *async_alloc*, *async_reg*, *man*, and *man_prefetch*.

3 MEMORY TRANSFER PERFORMANCE

From the previous section, we can see that switching the communication model of a CUDA application has the potential to lead to improved overlapping of host and GPU device activity. We can also see that such a switch requires several subtle changes beyond just switching the allocation and transfer functions.

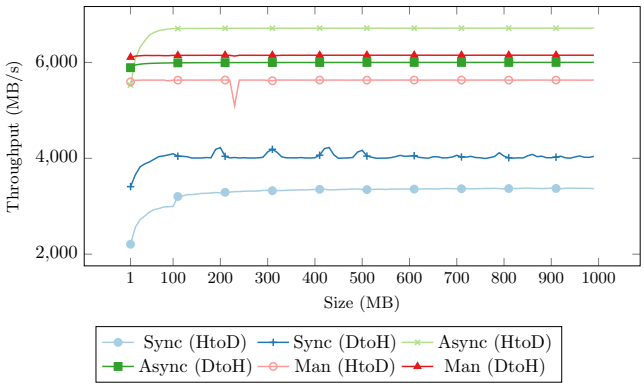


Figure 2: Data throughput of NVIDIA K20 GPU

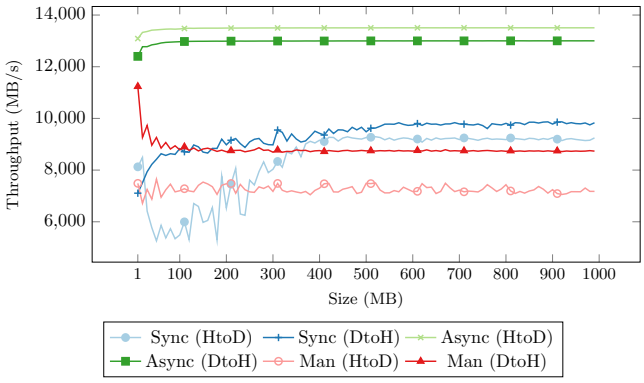


Figure 3: Data throughput of NVIDIA RTX 2080 Ti GPU

This section presents an investigation into whether we can expect gains in transfer bandwidth when switching the communication model. We use a synthetic workload similar to the canonical example of the previous section as our test vehicle. We allocate and transfer a single array to the device and perform an element-wise increment on the array before transferring the result back to the host. Doing only an increment ensures that our computation is IO-bound. We systematically vary the size of the array from a few bytes up to 1 GB.

Due to the simplistic nature of the benchmark, we restrict ourselves to the communication models *sync*, *async_reg*, and *man*. We run these on two GPU devices, an NVIDIA K20 (Kepler architecture from 2012) and an NVIDIA RTX 2080 Ti (Turing architecture from 2018)[2]. We use NVIDIA's profiling tool nvprof to measure the bandwidth of the memory transfers that is being achieved. The results of our experiments are shown in Figures 2 and 3. For each memory transfer model, we distinguish between host-to-device (*HtoD*) and device-to-host (*DtoH*) communication as the corresponding bandwidths differ significantly.

[2]Further details of these systems can be found in Table 1 in Section 7.

In Figure 2 the K20, based on the older Kepler architecture, achieves the highest throughput using asynchronous communication at over 6 GB/s followed closely by managed memory communication. The default synchronous communication lags behind at just under 4 GB/s. In all cases there is a gap of about 1 GB/s in throughput between the host-to-device and device-to-host transfers.

Using the more recent Turing architecture of the RTX 2080 Ti, we see in Figure 3 a different picture. Here, asynchronous communication is the best at a throughput of about 13 GB/s. Synchronous and managed memory communication lag significantly behind, with peak throughput of about 9 GB/s.

It is not surprising that asynchronous communication achieves the highest throughput on both systems, as it makes use of the DMA, avoiding CPU IO overheads. The behaviour of managed memory on both systems is different, with the measurements for the RTX 2080 Ti showing a large amount of variance in throughput as we change the size of the input array. Given that both synchronous and managed memory rely on the CPU for memory IO, the variance can be attributed to the dynamically changing clock-rate of the CPU for the given system. As we could not change that behaviour of the RTX 2080 Ti system, we turned the frequency scaling on in the K20 system and saw similar jitter for smaller memory sizes as on the RTX 2080 Ti system.

From these results, we draw the following conclusions. Firstly, the differences in bandwidth performance between the chosen communication models can be significant. For both architectures, we see up to a factor of 2 difference between the smallest and largest bandwidth measurements. Secondly, the relative behaviour depends not only on the GPU architecture, but also on the host capabilities and configuration of the host system (*e.g.* whether CPU frequency scaling occurs). Finally, while asynchronous communication bandwidth performance seem to be almost agnostic to the amount of data that is being transferred, this is less so for the other two communication models. In particular on the Turing architecture, it seems that the bandwidth performances for managed memory transfers outperform synchronous transfers while data less than 400 MB is being transferred; this observation reverses for larger transfer sizes.

Given these observations, it might seem advisable to always prefer the use of asynchronous communication over the other memory models. This choice though does not account for the type of workload being done. For this example we chose a very simple, IO-bound, workload which only communicates all the data once. Most GPU application are more complex, with computations being shared by both the CPU and GPU, and with many memory transfers being made. Given this, purely relying on transfer bandwidth to chose which communication model to use is not sufficient. With these results, it appears necessary to adjust the memory transfer model to a given combination of algorithm, host system and GPU device when trying to achieve the best possible overall runtime performance.

4 GENERATING CUDA FROM SAC

We will now look at the pre-existing work on how CUDA code is generated from SaC as described in [11, 12]. It uses synchronous transfers only. In Section 5, we expand on this work to support all transfer models from Section 2.

SaC is a functional array programming language that exposes no notion of hardware to the programmer: the use of GPUs, threads or even the notion of memory, on the host or the GPU-device, is hidden completely[3]. Our working example from Section 2, rewritten in SaC, reduces to the purely computational aspects. If we inline the increment function, we get the following SaC code snippet:

```
1  int main () {
2    ...
3    va = { iv -> va[iv] + 1; };
4    ...
5    return 0;
6  }
```

Here *set notation* [7] is used to represent the incrementation loop, where `iv` is a index-vector used to access each element of `va` and perform the addition. Note that all memory related operations are gone; the notion of a kernel has disappeared too, along with any indication that the variable `va` on the left hand side of line 3 can denote the same memory location as the variable `va` on the right hand side of that line.

This completely implicit notion of memory and memory transfers makes SaC an ideal starting point for generating CUDA code variants of different memory transfer models, adhering to all the synchronisation particulars as discussed in Section 2.

Several techniques are already developed and implemented in the context of SaC, to transform, optimise, and eventually, generate target architecture and resource-aware codes for efficient execution on a wide range of platforms [6, 12, 19, 24]. This includes a backend for generating CUDA code from SaC programs.

To help understand how we generate code for the different communication models explained in Section 2, we will describe the relevant stages of the compilation into CUDA. As described in [11, 12], the CUDA backend introduces the notions of host memory and device memory, as well as explicit transfers between them. It also tries to minimize memory transfers between the two. For our given example, the initial computation of `va` might be the result of a fusion with whatever happens in the code represented by the three dots in line 2 and the increment in line 3. It would also fuse that computation with whatever happens with the incremented version of the array `va` in the three dots of line 4.

For the sake of presentation, let us assume here that such fusion is not performed and that there is no way of producing or consuming `va` directly on the device. Consequently, the code generator described in [12] would generate some intermediate code of the form:

```
1  int main() {
2    ...
3    va_dev = _host2device_(va);
4    vb_dev = { i -> va_dev[i] + 1;}@CUDA;
5    vb  = _device2host_(vb_dev);
6    ...
7    return 0;
8  }
```

Here, we see how the compiler has introduced the notion of two different memories (host and device), explicit transfers between

[3] More details on SaC can be found in [8, 25] or at https://www.sac-home.org.

them, and has identified the kernel itself as the array computation in line 4 (denoted with postfix @CUDA). For readability, we add a postfix to all device-allocated memories with _dev, and leave all host allocated memories without postfixes. At this level of abstraction, the identifiers do not refer to memory locations. The notion of memory is introduced at a later stage, as well as the notion of references and operations for dynamic reference counting. At that stage, the code roughly looks like this:

```
1  int main () {
2    ...
3    va_dev = _dev_alloc_(1024, int);
4    va_dev = _host2device_(va);
5    vb_dev = _dev_reuse_(va_dev);
6    vb_dev = { i -> va_dev[i] + 1;}@CUDA;
7    vb = _alloc_or_reuse_(1024, int, va);
8    vb = _device2host_(vb_dev);
9         _dev_decrc_(vb_dev);
10   ...
11   return 0;
12 }
```

On this level of abstraction, we have explicit operations for allocating memory (**_alloc_**), reusing pointers (**_reuse_**), potentially reusing pointers (**_alloc_or_reuse_**), freeing memory (**_free_**) and potentially freeing memory (**_decrc_**). All these operations have two variants depending on whether they pertain to device memory (prefixed by _dev_) or to host memory. The uncertainty in some of the operations stems from the fact that aliasing analyses are undecidable in principle. As a consequence, dynamic inspections of reference counters are necessary, to determine whether some memory can be reused or needs to be freed. Details on reference counting in general and the specifics of the SaC compiler can be found in [1] and in [9], respectively.

From this stage, the SaC compiler generates C code. Primitives like **_alloc_** are transformed into intermediate code macros (ICMs), which are later resolved by the C compiler. This allows for composing variants of code depending on parameters set by the SaC compiler (and the user). Additionally, certain statically determined properties for array variables are set, these include shape information and the reference count. This information is stored as adjacent variables that share the same name as the array but have a postfix, indicating their purpose. This information is used by the runtime system to, for instance, determine if a variable can be freed, or even reused, at a particular point. With that we get the following C source code:

```
1  __global__ void knl_1024(int * va_dev) {
2      int i = blockIdx.x * blockDim.x + threadIdx.x;
3      va_dev[i] = va_dev[i] + 1;
4  }
5  int main () {
6    ...
7    SAC_CUDA_ALLOC (va_dev, 1024, int)
8    SAC_CUDA_MEM_TRANSFER (va, va_dev, 1024, int,
9                           cudaMemcpyHostToDevice)
10   SAC_ND_REUSE (vb_dev, va_dev);
11   dim3 block(16);
12   dim3 grid(1024/16);
13   SAC_CUDA_KERNEL_CALL (knl_1024, block, grid, vb_dev)
14   SAC_ND_ALLOC_OR_REUSE (vb, 1024, int, va)
15   SAC_CUDA_MEM_TRANSFER (vb_dev, vb, 1024, int,
16                          cudaMemcpyDeviceToHost)
17   SAC_CUDA_DEC_RC_FREE (vb_dev)
18   ...
19   return 0;
20 }
```

At this stage, the generated code looks similar to our example code in Listing 1. All of the ICMs are direct translations from the SaC primitives, the only difference is the explicit computation of the grid and block sizes, where the compiler has set the block size to 16. When this source code is passed to the C compiler, the ICMs will resolve into small snippets of C code by means of possibly recursive macro expansion. For instance, SAC_ND_ALLOC_OR_REUSE will resolve into something similar to:

```
vb = va_refcnt == 1 ? va : SAC_ND_ALLOC (vb,1024,int);
```

where SAC_ND_ALLOC is the standard host allocation and resolves into a call of malloc.

Through the definition of the ICMs, local changes can be easily materialised. From our running example in Section 2, we can see that the changes needed for using different CUDA memory regimes are fairly local.

The ICMs relevant in this context for synchronous transfers are:

```
SAC_ND_ALLOC (var, size, type)
↪ var = malloc (size*sizeof (type));
```

for allocating host memory;

```
SAC_ND_FREE (var, size, type)
↪ free (var);
```

for de-allocating host memory;

```
SAC_CUDA_ALLOC (var, size, type)
↪ cudaMalloc (&var, size*sizeof (type));
```

for allocating memory on the GPU device;

```
SAC_CUDA_FREE (var)
↪ cudaFree (var);
```

for de-allocating memory on the GPU device;

```
SAC_CUDA_MEM_TRANSFER (src, dst, size, type, dir)
↪ cudaMemcpy (dst, src, size*sizeof (type), dir);
```

for transfers between host and device; and

```
SAC_CUDA_KERNEL_CALL (fun, block, grid, v1, ..., vn)
↪ fun <<<block,grid>>> (v1, ..., vn);
```

for launching CUDA kernels.

For more details on the code generation process, please refer to [11, 12, 25].

5 GENERATING CODE FOR CUDA TRANSFER MECHANISMS

We now focus on the required changes in our translation process for realising the different variants of memory management in CUDA. Ideally, these boil down to redefinitions of the most relevant ICMs presented in the previous section.

5.1 Generating CUDA for Asynchronous Transfers

As a starting point for using asynchronous transfers, we redefine these ICMs as:

```
SAC_ND_ALLOC (var, size, type)
↪ cudaHostAlloc (&var, size*sizeof (type),
                 cudaHostAllocDefault);
SAC_ND_FREE (var, size, type)
↪ cudaFreeHost (var);
SAC_CUDA_ALLOC (var, size, type)
↪ cudaMalloc (&var, size*sizeof (type));
SAC_CUDA_FREE (var)
↪ cudaFree (var);
SAC_CUDA_MEM_TRANSFER (src, dst, size, type, dir)
↪ cudaAsyncMemcpy (dst, src, size*sizeof (type), dir);
    cudaDeviceSynchronize ();
SAC_CUDA_KERNEL_CALL (fun, block, grid, v1, ..., vn)
↪ fun <<<block,grid>>> (v1, ..., vn);
```

Note here that we only change the host memory handling from the system allocator `malloc` to CUDA's **`cudaHostAlloc`**, and replace the synchronous **`cudaMemcpy`** by the asynchronous **`cudaAsyncMemcpy`** and a **`cudaDeviceSynchronize`** call.

While this works in principle, it has two glaring short-comings. First, it forces *all* memory allocations on the host to be done by **`cudaHostAlloc`**, even those for memory that is never being transferred to the device. Second, we always synchronise *directly* after starting an asynchronous transfer which effectively renders these transfers synchronous again. We still benefit from the improved throughputs identified in Section 3 due to using DMA, but we lose all opportunities for latency hiding.

To tackle the first issue, we need to find out which arrays are ultimately being transferred to the device and which ones are not. As this in general is undecidable, we would need to implement a conservatively approximating analysis. We can avoid these problems when switching to the host register technique explained in Section 2. We can switch to this technique by redefining `SAC_ND_ALLOC` and `SAC_ND_FREE` as:

```
SAC_ND_ALLOC (var, size, type)
↪ var = malloc ( size*sizeof (type));
    cudaHostRegister (&var, cudaHostRegisterDefault);
SAC_ND_FREE (var, size, type)
↪ free (var);
```

Once we have gone back to normal allocations, we can make the CUDA memory registration a conditional action triggered in the expansion of memory transfers. All that is needed to implement this is a shift of **`cudaHostRegister`** from the `SAC_ND_ALLOC` ICM to `SAC_CUDA_MEM_TRANSFER`:

```
SAC_ND_ALLOC (var, size, type)
↪ var = malloc ( size*sizeof (type));
SAC_CUDA_MEM_TRANSFER (src, dst, size, type, dir)
↪ if (notYetPinned (src))
      cudaHostRegister (src, cudaHostRegisterDefault);
    cudaAsyncMemcpy (dst, src, size*sizeof (type), dir);
    cudaDeviceSynchronize ();
```

For latency hiding during memory transfers, this cannot be resolved by simple redefinitions of ICMs. Here, we need to manipulate the code on the intermediate level described in Section 4. At that stage, any array `va` that is needed on the device is transferred as early as possible by an operation

```
1  ...
2  va_dev = _host2device_ (va);
3  ...
```

In order to create an opportunity for latency hiding we need to split these operations into two parts, denoting a synchronisation window: We use **`_host2device_start_`** to initiate the transfer, and a **`_host2device_end_`** to indicate that the transfer needs to be completed before proceeding. This splitting of transfers replaces the above snippet by

```
1  ...
2  va_tmp_dev = _host2device_start_(va);
3  va_dev = _host2device_end_ (va_tmp_dev, va);
4  ...
```

where `va_tmp_dev` is an artificial variable to ensure data dependence between the two. In the final code this artificial variable will just be a pointer alias to `va_dev`. Having `va` as parameter of **`_host2device_end_`** ensures that the original host value is kept intact until **`_host2device_end_`** is executed. Once all transfers are split this way, we move the **`_host2device_end_`** operations as far down the data flow as possible creating potential for latency hiding. The splitting of **`_device2host_`** operations is done in the same fashion. Finally, we generate new ICMs for those four operations which are ultimately translated into the actual transfers and device synchronisations, respectively.

5.2 Generating CUDA Managed Memory Code

In the managed case, a simple implementation based on ICM redefinitions does not suffice either. The main problem here is that unification of host and device memory requires a completely different handling of memory; at the stage where ICMs are introduced, fixed memory choices have already been introduced by the compiler. Instead, modifications are made to the first stage, outlined in Section 4, which does not have a notion of memory other than talking about device variables and host variables. At that stage, all device variables are replaced by host variables, and transfers are replaced by assignments which eventually will lead to pointer aliasing.

Once that is done, implicit memory management support in the SaC compiler provides an optimised memory organisation adapted to the new data dependencies for free. We neither need to worry about renaming of identifiers and potentially required copying of data, nor about synchronisations before the kernel launch as explained in Section 2. The side-effect-free setting of SaC ensures that the kernel arguments cannot be modified before the kernel is invoked. With a synchronisation directly after a kernel launch, we avoid all possible race conditions.

All transfers back from the device to the host are implicit, whenever the host accesses the memory holding the result of the kernel computation. This leaves us with the standard memory and kernel ICMs implemented as:

```
SAC_ND_ALLOC (var, size, type)
↪ cudaMallocManaged (&var, size*sizeof(type),
                     cudaMemAttachGlobal);
SAC_ND_FREE (var, size, type)
↪ cudaFree (var);
SAC_CUDA_KERNEL_CALL (fun, block, grid, v1, ..., vn)
↪ fun <<<block,grid>>> (v1, ..., vn);
    cudaDeviceSynchronize ();
```

All memory allocations and de-allocations are now done using the **cudaMallocManaged** and **cudaFree** functions, respectively, and all kernel calls are followed by a device synchronisation.

The current extension makes using the *man* communication model possible, in order to support *man_prefetch* we must make further changes to the compiler. When transforming the code to CUDA UM, rather then replacing all transfers with aliasing assignments, we instead replace them with new primitive functions called **_prefetch2host_** and **_prefetch2device_**. Unlike with **cudaAsyncMemcpy**, for instance, the prefetch operation does not modify any memory. For this reason we cannot reuse the memory transfer ICMs to transform the primitive into real code. Instead we create a new ICM called SAC_CUDA_MEM_PREFETCH which resolves into a **cudaMemPrefetchAsync** function call. The ICM definition is implemented as:

```
SAC_CUDA_MEM_PREFETCH (var, size, type, context)
↪ cudaMemPrefetchAsync (var, size*sizeof(type), context);
```

In addition to the variable to be prefetched, we also pass in a ordinal value which either refers to the host system or a GPU device.

6 ENHANCED MEMORY REUSE OPTIMISATION

With the SaC compiler extensions of the previous section, we can now generate code for the different CUDA memory transfer models. Our first evaluations, however, show particularly poor performance for the *async_reg* case whenever many pinning operations on host allocations are performed. A simple yet rather common example of such a situation is the following loop:

```
1  do {
2      va = { i -> va[i] + 1; }
3  } while (property(va) == false);
```

If the computation of property(va) cannot be done on the GPU, we obtain an intermediate code that requires va to be communicated from the device to the host within each iteration. After the compiler has included explicit memory management, the code looks like this:

```
1  ...
2  va_dev = _dev_alloc_(size, int);
3  va_dev = _host2device_(va);
4  _decrc_(va);
5  do {
6      va_dev = { i -> va_dev[i] + 1; }@CUDA;
7      va_tmp = _alloc_(size, int);
8      va_tmp = _device2host_(va_dev);
9  } while (property(va_tmp) == false);
10 vb = _alloc_(size, int);
11 vb = _device2host_(va_dev);
12 _dev_free_(va_dev);
13 ...
```

The pitfall is that va_tmp is local in the loop body and not needed after the termination check. Consequently, on each iteration va_tmp is allocated, pinned, and freed during the execution of property(va_tmp).

While this behaviour adheres to the claim of reference counting to keep the amount of allocated memory to a minimum at all times, it is counter-productive here. To avoid this, the allocation of va_tmp must occur before the loop is entered and it needs to be kept alive across all iterations, enabling reuse of va for va_tmp before and va_tmp for vb after the loop:

```
1  ...
2  va_dev = _dev_alloc_(size, int);
3  va_dev = _host2device_(va);
4  va_tmp = va;
5  do {
6      va_dev = { i -> va_dev[i] + 1; }@CUDA;
7      va_tmp = _device2host_(va_dev);
8  } while (property(va_tmp) == false);
9  vb = va_tmp;
10 ...
```

As it happens, such a transformation already exists within the compiler, but it *only* applies to memory allocations that are needed for the results of *with*-loops, not for the results of built-in primitives such as **_device2host_**. Called the Extended Memory Reuse (*EMR*) optimisation [27], it can propagate allocations out of loops to enable reuse of memory by artificially keeping memory alive across iterations. It can also reuse memory for sequences of *with*-loops or loops, such that the results of these are reused in later stages. *EMR* builds on top of reference counting techniques for in-place reuse [5] and reuse through polyhedral analysis [10]. It infers a set of pointers to memory that normally would be freed before the current allocation. If such a reuse candidate is suitable in type and shape, a direct reuse is possible and done at compile time. Otherwise at runtime we check the reuse candidates reference count and reuse it if there are no further references.

As the same logic can be applied to all primitive functions including the memory transfer and prefetch primitives, we extended *EMR* accordingly. Whenever we encounter a transfer primitive, we apply the same inference operation as we do with *with*-loops. For the split transfer primitives used for the asynchronous communication models, the reuse candidates are linked only to the _start_ primitive as these resolve into the CUDA memory transfer functions. When using the *man_prefetch* communication model, upon transforming the memory transfer primitives to prefetch primitives, the previously inferred reuse candidates are preserved. With this extension to *EMR*, we can now generate the improved code version from above, resolving the repeated allocation issue within the loop.

7 EVALUATION

From our analysis of bandwidth from Section 3, we expect to see benefits for the asynchronous model on both architectures studied there and a better performance of the managed cases on the K20 than on the RTX 2080 Ti. To avoid the noise that we have observed

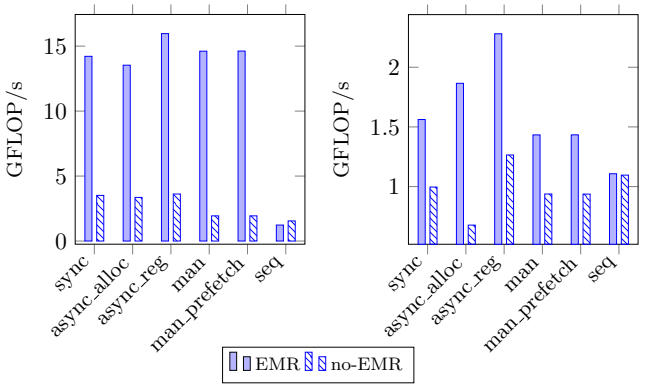
on smaller transfers we base our examples on large transfers. We choose two examples which are similar in what they compute but have have rather different compute-to-communication ratios.

We look at two variants of a 5-point stencil code on a matrix of 10k by 10k elements. Each of these matrices requires 100M of doubles equating 800 MB of memory; a single matrix update requires almost exactly 500M floating point operations. In our first variant, we perform a fixed number of iterations, here 100. This means we have a compute-to-communication ratio of $(500 * 100)/(100 + 100) = 250$ FLOP/double. In our second variant, we communicate the updated matrix back to the host after each iteration to compute the termination criterion on the host. Here the compute-to-communication ratio is much less favourable with $500/100 = 5$ FLOP/double. These two variants do not only provide two rather extreme points of compute-to-communication ratio, they are also a typical pattern found in scientific codes.

Our systems setup is shown in Table 1. We used version 1.3.3-482 of the SaC compiler on both systems. We ran our experiments with *EMR* turned on and off, to show the effect of memory allocations on the performance of the different transfer models. We ran each experiment five times and computed the median value of the five runs. As our measurements are all within less than 1% of variation, our measurement figures leave out error bars for better readability.

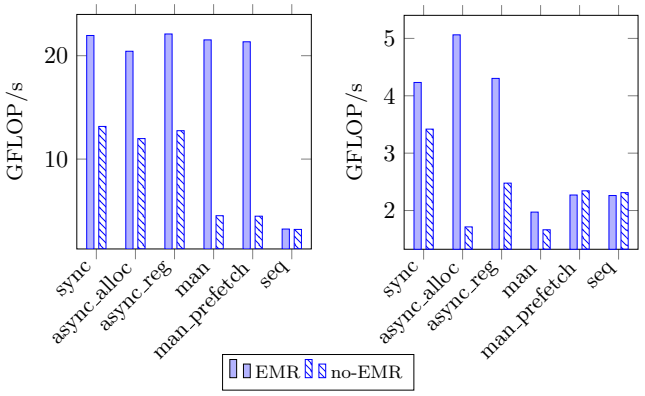
Table 1: Details of Systems used for Experiments

System	Hardware	Software
A	4× AMD Opteron 6376 — 64-cores @ 2.3 GHz 1 TB RAM NVIDIA K20 (driver v. 418.87)	Scientific Linux 7.6 GCC 7.2.0 HWLOC 1.11.8 CUDA 10.1
B	4× AMD Ryzen 7 2700 — 8-cores @ 1.5 GHz and 3.5 GHz 32 GB RAM NVIDIA RTX 2080 Ti (driver v. 418.39)	CentOS 7.6 GCC 7.4.0 HWLOC 1.11.13 CUDA 10.1



(a) Fixed-iteration **(b) Cond.-iteration**

Figure 4: FLOP/s measurements on K20



(a) Fixed-iteration **(b) Cond.-iteration**

Figure 5: FLOP/s measurements on RTX 2080Ti

7.1 Results

Our results for the two different systems are shown in Figures 4 and 5. We use GFLOP/s as the measure of performance since this (a) gives an impression of the overall performance achieved, and (b) allows for a more meaningful comparison to sequential execution. Within each figure, we present two graphs, one for each benchmark. Each graph shows the performance for all 5 CUDA variants as well as for the sequential host-only execution. Each variant has two bars, the left bar (solid) is the performance with *EMR* turned on and the right bar (striped) is the performance with *EMR* disabled.

Starting with the fixed-iteration variant on the K20 in Figure 4a we can see that *EMR* makes a huge difference in performance for all CUDA variants. This may come as a surprise as there is no communicate between the individual iterations at all. Upon closer inspection, we see that it is not our extension of *EMR* discussed in Section 6 that is responsible for this effect, but it is the *EMR* as described in [27]. Instead of allocating memory for the result of the update and freeing the old one within each iteration, it effectively implements pointer swaps. The effect is an overall performance improvement from roughly 4 GFLOP/s to 15 GFLOP/s, an almost 15*x* improvement over the sequential version. Not very surprisingly, we see that the *async_reg* variant performs best here with roughly 10% better performance than the other versions even though we have a high compute-to-communication ratio. What is surprising, is that the comparison of variants looks different when looking at the versions without *EMR*. Here, *sync* and *async_reg* perform almost identical and *man* performs relatively poorly. This shows how sensitive performance is to memory allocations. Clearly, the allocations in *sync* are the cheapest, while those in *man* are the most time consuming ones. The latter is due to the latency caused by allocating managed memory, which affects physical memory on both the host and device.

For the second benchmark, where the iteration depends on the convergence rate, the most striking difference is that the overall performance we can achieve is much lower. The best performance we see here is about 2.3 GFLOP/s. The reason lies in the very low compute-to-communication ratio of 5 FLOP/double. Again, we see how *EMR* makes a difference here. With this high level of communication, *EMR* is crucial for achieving any significant improvement over the sequential execution on the host at all. In contrast to the

previous example, our extension of *EMR* from Section 6 plays an essential role here. We can also see that the differences between the CUDA versions are more pronounced with *async_reg* outperforming *sync* by 50%, an effect that directly results from the low compute-to-communication ratio.

For the RTX 2080 Ti, we observe for fixed iteration in Figure 5a the same performance pattern as in Figure 4a, though with higher absolute performance.

Conditional iteration in Figure 5b, shows less overall performance and more pronounced differences, but the best performances with *EMR* on or off now lie with different versions than before. The *async_alloc* variant performs the best overall, outperforming *async_reg* by 25%, and the *sync* variant achieves the best performance without *EMR*.

8 RELATED WORK

An overview and analysis of CUDA unified memory on three platforms using NVIDIA Pascal and Volta architectures is given in [3]. The authors evaluate several benchmarks, looking at the *sync*, *man*, and *man_prefetch* models. They also look into facilities to provide hints to the CUDA driver on how memory is used. Their findings are similar to ours, in that each platform behaved differently on different communication models. In [13], the authors perform a similar analysis comparing the *sync* and *async_alloc* models, focusing on latency hiding and its effect on runtime performance using two GPUs. The authors in [4] make a comparison between the different CUDA communication models, but on a Tegra SoC-based system where both host and device memory is shared. None of this work, however, looks at all models or at code generation for them.

The work in [17] looks at using a cost model for deciding which model to use: *sync* or *async_alloc*. They take hardware features of the GPU device into account. In contrast to our work, they do not start from a memory-agnostic source, instead generating their code from a task graph with an explicit notion of memory.

Runtime systems for handling task-parallelism, and inferring transfers as part of the asynchronous model are described in [2]. In [18] the authors showcase their tool *CuMAPz* which at compile-time determines suitable memory orchestration on the GPU device itself, but does not look into the different models.

9 CONCLUSION

This paper looks at optimising the code generation of CUDA code from resource-agnostic source code. In particular, we look at the performance differences that the choice of communication model in the CUDA interface makes.

A bandwidth comparison between two different hardware systems shows a $2x$ difference in communication bandwidth between the different communication models. Unfortunately, we also see that the differences in bandwidth performance depend very much on which GPU architecture is used. Even worse, the capabilities and configuration of the host system has an impact on the achieved bandwidth as well. This already suggests that the final choice in communication model ideally needs to target the given hardware specifically.

We show that switching from one communication model to another is not quite as simple as it may seem. Depending on the model used, different ways of allocating memory are needed and synchronisations need to be carefully inserted into the code. This is done to ensure correctness and efficiency at runtime. In order to fully utilise the potential of asynchronous transfers, we need to identify synchronisation windows which, due to the side-effect free setting of SaC, can be easily determined from the data dependencies within the program.

When switching to managed memory, the challenges are even bigger; a completely different memory organisation is required. Here, we learn that generating code from a resource-agnostic source code is particularly useful. It enables the compiler to implicitly adapt to a new memory allocation regime without requiring any additional effort.

Our performance evaluation shows that the choice of communication model cannot simply be derived from the bandwidth performance alone for a given system. Additionally, how memory is allocated changes with the communication model and can affect the number of memory allocations needed, which plays a major role here. Excessive allocations can easily outweigh potential gains made by switching communication model. In the context of code generation through the SaC compiler, the optimisation *EMR* can be leveraged here to reduce the number of allocations by reusing memory. We extend it in order to additionally avoid allocations for memory transfers. Without it, even rather simple examples do not necessarily benefit from changing to a different communication model.

Finally, the compute-to-communication ratio has an overarching impact on the effectiveness of the chosen communication model: the lower this ratio gets the more critical the choice becomes. For a ratio of 250 FLOP/double we see a difference of about 10% between the models, whereas a ratio of 5 FLOP/double yields a difference of up to a factor of $3x$.

In summary, the choice of communication model needs to be based on the combination of hardware used, the size of the data involved, and the ratio of compute-to-communication and the number of memory allocations made. Deciding, whether by using some heuristics, or using dynamic adaptation, is beyond the scope of this paper. What we learn here is that it is crucial to be capable to generate all variants from a single source code. Furthermore, using resource-agnostic and side-effect free source languages, such as SaC, pose an ideal starting point for such an endeavour. The side-effect free setting guarantees that data will stay unmodified while it is referenced, which facilitates race-condition free synchronisation windows. Code generation from a resource-agnostic source code implicitly comes with built-in support for introducing the notion of memory, which is a key capability to piggyback on when switching between explicit communication and CUDA's managed memory.

ACKNOWLEDGMENTS

This work is supported by the Physical Sciences Research Council through grants EP/N028201/1 and EP/L016834/1. We would like to thank the annonymous reviewers and Robert Bernecky for their valuable feedback and suggestions, and Simon McIntosh-Smith from the University of Bristol for giving us access to the HPC Zoo Cluster[4] for our experiments.

[4]For details, see https://uob-hpc.github.io/zoo/

REFERENCES

[1] D.C. Cann. 1989. *Compilation Techniques for High Performance Applicative Computation.* Technical Report CS-89-108. Lawrence Livermore National Laboratory, LLNL, Livermore California.

[2] Sanjay Chatterjee, Max Grossman, Alina Sbîrlea, and Vivek Sarkar. 2013. Dynamic Task Parallelism with a GPU Work-Stealing Runtime System. In *Languages and Compilers for Parallel Computing*, Sanjay Rajopadhye and Michelle Mills Strout (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 203–217. https://doi.org/10.1007/978-3-642-36036-7_14

[3] S. Chien, I. Peng, and S. Markidis. 2019. Performance Evaluation of Advanced Features in CUDA Unified Memory. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. 50–57. https://doi.org/10.1109/MCHPC49590.2019.00014

[4] Jake Choi, Hojun You, Chongam Kim, Heon Young Yeom, and Yoonhee Kim. 2020. Comparing unified, pinned, and host/device memory allocations for memory-intensive workloads on Tegra SoC. *Concurrency and Computation: Practice and Experience* (Sept. 2020). https://doi.org/10.1002/cpe.6018

[5] Steven M. Fitzgerald and Rodney R. Oldehoeft. 1996. Update-in-place Analysis for True Multidimensional Arrays. *Sci. Program.* 5, 2 (July 1996), 147–160. https://doi.org/10.1155/1996/493673

[6] Clemens Grelck. 2005. Shared memory multiprocessor support for functional array processing in SAC. *Journal of Functional Programming* 15, 3 (2005), 353–401. https://doi.org/10.1017/S0956796805005538

[7] Clemens Grelck and Sven-Bodo Scholz. 2003. Axis Control in Sac. In *Implementation of Functional Languages, 14th International Workshop (IFL'02), Madrid, Spain, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 2670)*, Ricardo Peña and Thomas Arts (Eds.). Springer, 182–198.

[8] Clemens Grelck and Sven-Bodo Scholz. 2006. SAC: A Functional Array Language for Efficient Multithreaded Execution. *International Journal of Parallel Programming* 34, 4 (2006), 383–427. https://doi.org/10.1007/s10766-006-0018-x

[9] Clemens Grelck, Sven-Bodo Scholz, and Kai Trojahner. 2004. With-loop Scalarization: Merging Nested Array Operations. In *Implementation of Functional Languages, 15th International Workshop (IFL'03), Edinburgh, Scotland, UK, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 3145)*, Phil Trinder and Greg Michaelson (Eds.). Springer. https://doi.org/10.1007/978-3-540-27861-0_8

[10] Jing Guo, Robert Bernecky, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. 2014. Polyhedral Methods for Improving Parallel Update-in-Place. In *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques*, Sanjay Rajopadhye and Sven Verdoolaege (Eds.). Vienna, Austria.

[11] Jing Guo, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. 2009. Towards Compiling SAC to CUDA. In *Proceedings of the Tenth Symposium on Trends in Functional Programming, TFP 2009, Komárno, Slovakia, June 2-4, 2009 (Trends in Functional Programming, Vol. 10)*, Zoltán Horváth, Viktória Zsók, Peter Achten, and Pieter W. M. Koopman (Eds.). Intellect, 33–48.

[12] Jing Guo, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. 2011. Breaking the Gpu Programming Barrier with the Auto-parallelising Sac Compiler. In *6th Workshop on Declarative Aspects of Multicore Programming (DAMP'11), Austin, USA*. ACM Press, 15–24. https://doi.org/10.1145/1926354.1926359

[13] Juan Gómez-Luna, José María González-Linares, José Ignacio Benavides, and Nicolás Guil. 2012. Performance models for asynchronous data transfers on consumer Graphics Processing Units. *J. Parallel and Distrib. Comput.* 72, 9 (2012), 1117–1126. https://doi.org/10.1016/j.jpdc.2011.07.011 Accelerators for High-Performance Computing.

[14] Tianyi David Han and Tarek S. Abdelrahman. 2009. HiCUDA: A High-Level Directive-Based Language for GPU Programming. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units* (Washington, D.C., USA) *(GPGPU-2)*. ACM, New York, NY, USA, 52–61. https://doi.org/10.1145/1513895.1513902

[15] Christoph Hartmann and Ulrich Margull. 2019. GPUart - An application-based limited preemptive GPU real-time scheduler for embedded systems. *Journal of Systems Architecture* 97 (2019), 304—319. https://doi.org/10.1016/j.sysarc.2018.10.005

[16] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-Programming with Nested Parallelism and in-Place Array Updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 556–571. https://doi.org/10.1145/3062341.3062354

[17] Hanwoong Jung, Youngmin Yi, and Soonhoi Ha. 2012. Automatic CUDA Code Synthesis Framework for Multicore CPU and GPU Architectures. In *Parallel Processing and Applied Mathematics*, Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Waśniewski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 579–588. https://doi.org/10.1007/978-3-642-31464-3_59

[18] Yooseong Kim and Aviral Shrivastava. 2013. Memory Performance Estimation of CUDA Programs. *ACM Trans. Embed. Comput. Syst.* 13, 2, Article 21 (Sept. 2013), 22 pages. https://doi.org/10.1145/2514641.2514648

[19] T. Macht and C. Grelck. 2019. SAC Goes Cluster: Fully Implicit Distributed Computing. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 996–1006.

[20] Nikolay Sakharnykh. 2017. Maximizing Unified Memory Performance in CUDA. https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda/. [Online; 29-May-2019].

[21] Nikolay Sakharnykh. 2018. Everything You Need To Know About Unified Memory. http://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf. [Online; 03-Nov-2019].

[22] NVIDIA Corporation. 2019. CUDA Toolkit Documentation v10.1.168. https://web.archive.org/web/20190523173815/https://docs.nvidia.com/cuda/archive/10.1/. [WayBack Machine; 02-Nov-2019].

[23] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. ACM, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[24] Sven-Bodo Scholz. 1998. With-loop-folding in Sac — Condensing Consecutive Array Operations. In *Implementation of Functional Languages, 9th International Workshop (IFL'97), St. Andrews, UK, Selected Papers (Lecture Notes in Computer Science, Vol. 1467)*, Chris Clack, Tony Davie, and Kevin Hammond (Eds.). Springer, 72–92. https://doi.org/10.1007/BFb0055425

[25] Sven-Bodo Scholz. 2003. Single Assignment C — Efficient Support for High-Level Array Operations in a Functional Setting. *Journal of Functional Programming* 13, 6 (2003), 1005–1059. https://doi.org/10.1017/S0956796802004458

[26] M. Steuwer, T. Remmelg, and C. Dubach. 2017. LIFT: A functional data-parallel IR for high-performance GPU code generation. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 74–85. https://doi.org/10.1109/CGO.2017.7863730

[27] Hans-Nikolai Vießmann, Artjoms Šinkarovs, and Sven-Bodo Scholz. 2018. Extended Memory Reuse: An Optimisation for Reducing Memory Allocations. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages* (Lowell, MA, USA) *(IFL 2018)*. ACM, New York, NY, USA, 107–118. https://doi.org/10.1145/3310232.3310242