Multi-GPU Code Generation for Out-Of-Core Problems

Patrick van Beurden[0009−0002−0202−907X],
Thomas Koopman[0009−0001−1031−7226], and
Sven-Bodo Scholz[0000−0002−8663−1043]

Radboud University, Nijmegen, Netherlands
{patrick.vanbeurden,thomas.koopman,svenbodo.scholz}@ru.nl

Abstract. We propose a light-weight solution for generating code that can use multiple GPUs from purely declarative program specifications. Building on code generation for a single GPU, we show how CUDA's unified memory can be leveraged to use multiple GPUs to collaboratively compute data-parallel tasks, and to handle computations on data that does not fit any of the GPU's memories. We describe the key ideas and implement them in SaC. We provide initial performance evaluations on two different GPU architectures for three different benchmarks: matrix multiplication, N-body simulation, and stencil computations. If allocation costs can be amortized, these experiments show efficiencies between 80% and 100% on up to four GPUs when using an explicitly memory-orchestrated CUDA version as baseline.

1 Introduction

Graphics processing units (GPUs) are processors that have higher peak compute rate and bandwidth compared to CPUs of comparable price and power consumption. This makes them attractive for a range of domains requiring extensive computation, such as physics simulations, financial applications and deep learning [10, 25, 19].

Obtaining the theoretic peak performance is challenging. The reason for this is that GPUs are not stand-alone systems but accelerators that are specialized to specific workloads. They need to be programmed differently, and have their own memory which needs to be coordinated with the CPU and RAM (also called the *host*). A straightforward way to increase the theoretic computational power of a system is to add multiple GPUs. However, realising this performance becomes difficult as the different GPUs need to be coordinated. An especially complicated class of problems are those that do not fit in the memory of a single GPU. We refer to these problems as being *out-of-core*. We are now essentially working with a distributed memory system and all its associated challenges.

Making effective use of multiple GPUs requires the programmer to have extensive knowledge of the hardware in addition to the problem domain. A large body of research tries to shift this burden from the application programmer to compiler writers by generating low-level GPU code from a high-level language.

In particular functional approaches through languages such as Accelerate [5], Futhark [14] or SaC [12] or other array languages such as APL [17] show GPU performance that is competitive with hand-written counterparts [24, 2, 13, 16]. The related work uses extensive compiler analysis, sometimes in combination with language extensions, to coordinate the GPUs.

The main idea behind this paper is that we can shift the burden even further, by making the runtime instead of the compiler responsible for data-movement. This way only minimal changes to the compiler are necessary to support using multiple GPUs on out-of-core problems.

The mechanism that makes this possible is *unified memory*: one virtual address space that can be accessed by all GPUs and the CPU. The GPU driver and hardware are then responsible for maintaining a coherent view of memory between GPUs and the host. This approach is typically associated with bad performance. A reason for this is that the runtime needs to be advised on access patterns to work efficiently. In many languages this is hard to do, but by working with a functional language we have the strong semantic guarantees necessary to generate this advice.

Our paper makes the following contributions.

- We propose a code generation scheme that can use multiple GPUs and handle out-of-core problems. It constitutes a light-weight extension of a pre-existing code generation scheme for unified memory [31];
- We show the effectiveness of our approach by implementing this scheme in the language Single-Assignment C;
- We show how memory advice can overcome performance challenges typically associated with unified memory.

2 Background

Single-Assignment C (SaC) generates CUDA code, a language for programming GPUs. We use this language to explain some GPU programming concepts, and briefly explain how SaC generates code.

2.1 GPU Programming

Computations are done through special functions called *kernel*s. These contain a small computation such as "compute $t + 1$ and store it in index t", similar to a loop-body in C (Listing 1.1).

```
void f(int *x, int n) {
    for (int t = 0; t < n; t++) {
        x[t] = t + 1;
    }
}
```

Listing 1.1: Array initalisation in C

The programmer specifies for which t this is computed at the call site. For example, to compute the set $X = \{t + 1 \mid 0 \leq t < n\}$, the kernel could be as follows.

```
__global__ void f(int *x) {
    int t = blockIdx.x * blockDim.x +
            threadIdx.x;
    x[t] = t + 1;
}
```

The initalisation of `t` (the *thread*) is complicated because there is not one canonical way to map computations to the hardware. Instructions are always executed in groups of threads, typically 32, similar to SIMD instructions on a CPU. Execution units on a GPU are grouped into *streaming multiprocessors* (SMs), which also includes cache and control logic. The threads are divided into groups called *blocks* of `blockDim.x` threads, and then divided over the SMs. The index local to one such block is stored in `threadIdx.x`, and the block index by `blockIdx.x`. Increasing the number of threads per block may help to hide latency, similar to hyperthreading in CPUs. However, it also decreases the number of blocks which can have a negative effect on scaling and load balancing. There is no choice that is optimal for all problems. The number of threads per block is limited by the hardware, typically 1024.

We specify the mapping to hardware between triple angular brackets at the call site. We can call `f` as follows.

```
f<<<n / 1024, 1024>>>(x);
```

Here the second argument `1024` is the number of threads per block `blockDim.x`, and the first argument `n / 1024` is the upper bound on `blockIdx.x`. We call this shape $[n/1024, 1024]$ the *thread space*.

The variables end with `.x` because they are actually three-dimensional structs with also `.y` and `.z` members. In this case all `.y` and `.z` members are 1, but for indexing higher-dimensional arrays it may be convenient to arrange the threads in a higher-dimensional grid. For example, 1024 threads may also be arranged in a 32×32 or $16 \times 16 \times 4$ grid.

2.2 Unified Memory

Managing multiple address spaces requires extra work for the application programmer. Systems from personal computers to the world's largest supercomputer (as of 2024) Frontier support abstractions that provide the programmer with a single virtual address space that can be accessed by the CPU and GPU. Depending on the vendor and exact implementation, these systems are called *unified memory*, *managed memory*, or *unified virtual addressing*. We will be using the term unified memory.

The implementation of unified memory in CUDA is proprietary, but it seems that by default a page of memory can be accessed by only one processor — either a GPU or the CPU — at a time, and if a page is accessed that is not present

in the respective processor's memory, a page fault occurs [7]. These page faults result in on-demand page migration, which can lead to mixed results [22, 21, 31]. This behaviour can be changed by giving the runtime *memory advice* on how the memory is used. For example the CUDA option *cudaMemAdviseSetReadMostly* has the device create a local copy.

2.3 Single-Assignment C

SaC is a functional array language for numeric programming. It tries to be as close to the mathematics as possible. For example, the array $X = \{t + 1 \mid 0 \leq t < n\}$ is specified as `X = {[t] -> t + 1 | [0] <= [t] < [n]}`. These array specifications, called *tensor comprehensions*, expose concurrency: each element can be computed independently from the others. The user can specify a compilation target, which the compiler `sac2c` will use to generate parallel code. For a GPU target, `sac2c` will generate a kernel from this specification. We have a correspondence between the *index set* `[n]` and the thread space `[n / 1024, 1024]`.

We can specify different expressions on different subsets of this index set, as illustrated in Figure 1 and Listing 1.2. In the related work such a subset is also called a partition. In Section 3 we will talk about the distinct mathematical notion of a partition, so to avoid confusion we will use subset in this paper.

We generate one kernel for each subset. The compiler contains extensive machinery for generating a good bijection between index set and thread space [18]. The compiler has a target `cuda` that explicitly defines arrays on the GPU and host. Transfers between the two are then optimised away as much as possible. The target `cuda_man` uses unified memory instead (also called *man*aged memory). The former has been observed to be more performant [31].

$f(0)$	$f(1)$	$g(2)$	$g(3)$	$f(4)$	$f(5)$	$g(6)$	$g(7)$

Fig. 1: Array specified by Listing 1.2

```
{[i] -> f(i) | [0] <= [i] < [8]
               width [2] step [4];
 [i] -> g(i) | [2] <= [i] < [8]
               width [2] step [4]}
```

Listing 1.2: Array specified by two index sets

3 Design

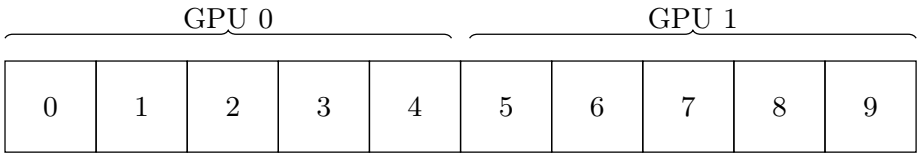
To keep the implementation lean, we only make minor modifications to the existing target using unified memory and one GPU [31]. We need to divide

the work, and appropriately instruct the CUDA runtime on the memory access patterns.

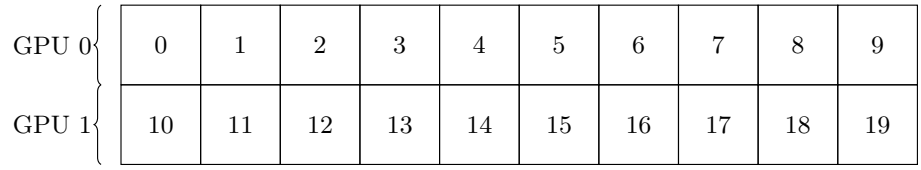
3.1 Work Distribution

In SaC, the index subsets may overlap, which requires us to process them sequentially. For this reason we do not distribute the subsets over the GPUs, but split up each subset individually.

We need to split up such an index set into non-overlapping pieces that together make up the original set (also called a *partition*). We do this by splitting along the first dimension, as illustrated in Figure 2. To be more precise, for the simple case where the index set is of the form $[n]$, we give GPU g rows $g \cdot b$ (inclusive) to $\max((g+1) \cdot b, n)$ (exclusive), where $b = \lceil \frac{n}{G} \rceil$, the smallest integer b such that $n \leq Gb$. This choice minimizes the maximum number of rows per GPU, so for regular computations this gives optimal load balance.



(a) 1D distribution



(b) 2D distribution

Fig. 2: Distribution of arrays over two GPUs.

General Index Subsets The most general way of specifying an index set in SaC is with a step and width. We need only minor modifications to the previous scheme. In set notation, an index set `[l] <= [i] < [u] step [s] width [w]` is equal to $[l, u) \cap S_{slw}$ where $S_{slw} = \{l + is + j \mid i \in \mathbb{Z},\ 0 \leq j < w\}$.

By distributing the intersection over the union, it becomes clear that a partition without strides over G GPUs, say

$$[l, u) = \bigcup_{g=1}^{G} [l_g, u_g),$$

also gives a strided partition

$$[l, u) \cap S_{slw} = \bigcup_{g=1}^{G} ([l_g, u_g) \cap S_{slw}).$$

For $l_g \equiv l \bmod s$, we have $S_{sl_gw} = S_{slw}$. So under this restriction the strides and widths are the same.

In a similar vein to the simple case $[l, u) = [0, n)$, we set $l_g = l + gb$, $u_g = \min(l + (g + 1)b, u)$ for some b. To satisfy $l_g \equiv l \bmod s$, we must take b of the form sk. The best load balance is now obtained by finding the smallest b such that $bG \geq u - l$, which is $\left\lceil \frac{u-l}{sG} \right\rceil \cdot s$.

Implementation of Work Distribution The partition of Figure 2a corresponds to rewriting the SaC code

```
{[i] -> i | [0] <= [i] < [10]}
```

to

```
{[i] -> i | [0] <= [i] < [5];
 [i] -> i | [5] <= [i] < [10]}
```

It is therefore tempting to also do this in the implementation. However, this leads to unnecessary code duplication as one kernel per GPU would be generated. Furthermore, the number of GPUs would need to be known at compile-time.

To avoid this, we reuse the thread space mechanism. This mechanism must know what the index subset is, so parameters describing the subset must be passed into the kernel (at the very least for cases where these parameters are not known at compile-time). This means we can restrict the execution of the kernel to the subsets defined above by only manipulating the call site, which allows us to treat the kernel generation, thread space mechanism included, as a black box. We insert some code around the kernel call at the very end of the compilation process, when we generate the CUDA code, which we sketch in Listing 1.3. The variables `l0`, `u0` correspond to the lower and upper bound in the first dimension. In the loop we replace these by l_g and u_g. To execute a kernel on a different GPU, we first call `cudaSetDevice(g)` where `g` is an index for that GPU. Then we call the kernel the same way as if we had one GPU. While the CPU will immediately continue after calling the kernel, the GPU has only finished computing after a call to `cudaDeviceSynchronize()`.

```
int old_l0 = l0;
int old_u0 = u0;

for (int g = 0; g < NUM_GPUS; g++) {
  /* Set l0, u0 to lg and ug */
  l0 = ...
  u0 = ...
  cudaSetDevice(g);
  /* The following three lines are
     unmodified from the
     cuda_man target. */
  grid = ...
  block = ...
  kernel<<<grid, block>>>(l0, u0, ...);
}

for (int g = 0; g < NUM_GPUS; g++) {
  cudaSetDevice(g);
  cudaDeviceSynchronize();
}

l0 = old_l0;
u0 = old_u0;
```

Listing 1.3: Kernel call distributed over `NUM_GPUS` GPUs

3.2 Memory Advice

An initial implementation of this code generation scheme has lead to poor results. We address this with an additional optimisation.

For example, distributing the work of Listing 1.4 over two GPUs naively leads to a 6× slowdown compared to using a single GPU.

```
double[m, k] matmul(double[m, k] A,
                    double[k, n] B)
{
    C = {[i, j] -> sum({[p] ->
              A[i, p] * B[p, j]})};
    return C;
}
```

Listing 1.4: Matrix Multiplication in SaC

The entire `j`th column of `B` is needed to compute the `j`th element of a row of `C`. As discussed in Subsection 2.2, the memory of `B` can only be on one GPU at a time by default. Whatever GPU accesses the memory first will get the memory page, and the other GPU will have to wait until it can be migrated

back. This serializes the computation between the two GPUs. Instead, it would be better if both GPUs created a copy of this page. This eliminates contention on reads, at the cost of making writes to the same page by multiple GPUs more expensive. This never happens in the code we generate because we only read from `B`. We can instruct the runtime to use this copy-on-read scheme by giving the `cudaMemAdviseSetReadMostly` memory advice.

This is not necessary for `A`. To compute row `i` of `C`, only row `i` of `A` is needed. As both `C` and `A` have `m` rows, the distribution over GPUs is the same. The data that has to be accessed is already present on that GPU, so there is no contention between accesses.

Unfortunately, adding memory-advice has overhead, so we cannot insert it everywhere. In the example Nine-Point Stencil, whose most important computation is in Listing 1.5, adding memory-advice leads to a $2 - 5\times$ slowdown compared to not inserting the memory advice.

```
{[i, j] -> w[0, 0] * x[i - 1, j - 1] +
            w[1, 0] * x[i    , j - 1] +
            w[2, 0] * x[i + 1, j - 1] +
            ...
            w[2, 2] * x[i + 1, j + 1];
  | [1, 1] <= [i, j] < [m - 1, n - 1];
...}
```

Listing 1.5: Sketch of the most computationally intensive part of a nine-point stencil.

This case is slightly more complicated than the access of `A` in Listing 1.4. The last row of GPU 0, which is the first row of GPU 1 is accessed by both GPUs. By GPU 0 as `x[i + 1, ...]` and by GPU 1 as `x[i - 1, ...]`. Except for the first and last row, all required data is already present on that GPU. We call this a *local* selection. The contention on these two rows is not enough to offset the cost of inserting the memory advice.

We can generalize this by looking at how the selection vector (for instance `[p, j]` in the case of `B` in Listing 1.4) relates to the index vector of the parallel tensor comprehension (`[i, j]` in both examples).

Let us write `[iv1, ..., ivd]` for the selection vector, and `[jv1, ..., jvd]` for the index vector. If `jv1` is equal to `iv1`, the selections are all local, and we do not have to insert the advice. Suppose our GPU has rows l to u. If `iv1` is equal to `jv1 + c` for some constant `c`, then the accesses are local for rows $l + c$ to $u - c$. This is most accesses for small `c`. For this reason, we mark accesses for which `iv1` is of the form `jv1 + c`. Given the functional nature of SaC we can then check for all relatively-free variables of tensor comprehensions whether all accesses are marked. If that is the case, we do not give memory advice. Otherwise, we mark them as read-mostly.

4 Performance Evaluation

We evaluate our approach in the style of distributed computing: we look at how much faster our code gets when adding more GPUs. The runtime of using one GPU divided by the runtime of using multiple GPUs is called the *speedup*. We expect at most a $G\times$ speedup when using G GPUs. To measure how close we get to this, we also compute the *efficiency*: speedup divided by the number of GPUs. The closer this is to one, the better our system scales.

4.1 Evaluation Platform

Scientific computations typically use double precision, but this is poorly supported in consumer-grade GPUs. To capture this, we use two different evaluation platforms. The first node (`icis`) has a similar double precision to bandwidth ratio as a consumer-grade GPU. The second node (`snellius`) has first-rate support for double precision. The hardware and software specifications are given in Table 1.

We use the SaC compiler at commit `f5890fa` with flags `-gpu_mapping_strategy jings_method_ext` for all benchmarks. Additional flags for a specific benchmarks are mentioned in their description. We used the standard library at commit `78c74b2`.

	icis	snellius
GPU	RTX A6000 (2x)	A100 (4x)
VRAM (GB)	48	40
Peak Bandwidth (GB/s)	768	1560
Peak FP64 (GFLOP/s)	604.8	9746
C compiler	gcc 11.3.0	gcc 12.3.0
NVCC	12.0.14	12.1.105
NVLink bandwidth (GB/s)	14	25
Number of NVLinks	4	4

Table 1: Overview of the test systems, hardware and software.

4.2 Methodology

We consider three benchmarks, two with a high ratio of computations to array size (Matrix Multiplication and N-Body Simulation), and one with a low ratio (Nine-Point Stencil). We use floating point operations per second (*flops*) as a measure of performance for those with a high ratio, and bandwidth for the benchmark with a low ratio.

We evaluate the performance on a range of problem sizes, shown in Table 2. Class D of Matrix-Multiplication is out-of-core on both systems, the rest of the

experiments are in-core. The reason for the small number of out-of-core examples is that SaC uses 32-bit integers for shapes, which limits the problem sizes.

We measure the costs inherent to the computation, that is the kernel-execution time, allocation of the result, and any inter-GPU communication. We do not measure any CPU-GPU transfer time of input and output because this may not be needed, for instance when the data is created and consumed on the GPU. The exact implementation and instructions for reproducibility are available [27]. We have used `pragma`s to manually reduce the number of threads per block for N-Body.

Class	**Matrix Multiplication**	**N-Body**	**Stencil**
A	10.1	0.010	4.29
B	19.7	0.021	9.66
C	38.3	0.042	17.2
D	51.0	0.084	34.0

Table 2: Overview of the memory footprints in GB.

We compute and efficiency compared to the performance of our fastest, manually orchestrated target *cuda*. The efficiency for an out-of-core run is compared to the performance of the *cuda* target of the closest in-core class.

4.3 Matrix Multiplication

A matrix multiplication computes the composition of linear maps, making it a fundamental operation in linear algebra with applications in many areas. A few examples are linear programming, cryptography and deep learning [8, 6]. The implementation, Listing 1.4, is straightforwardly translated from the mathematical definition

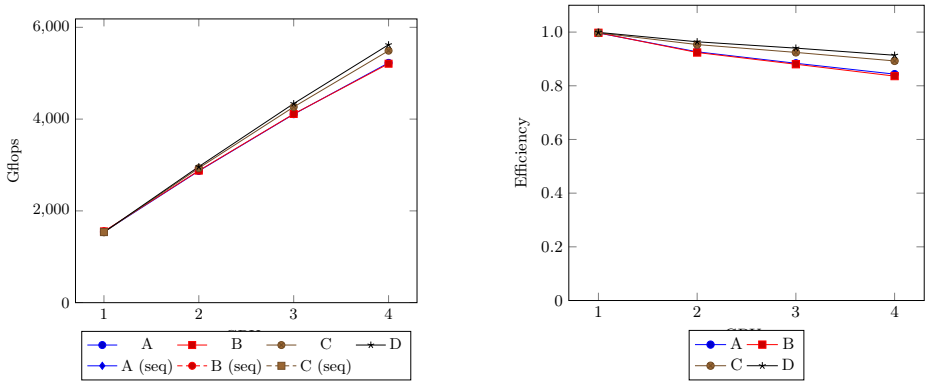
$$C_{ij} = \sum_{p=1}^{k} A_{ip} \cdot B_{pj}$$

for $m \times k$ matrix A and $k \times n$ matrix B.

Data from A and B are reused k times. It is well-known that algorithmic changes that improve the temporal locality of accessing `A` and `B` can speed up the computation, but we refer this to future work.

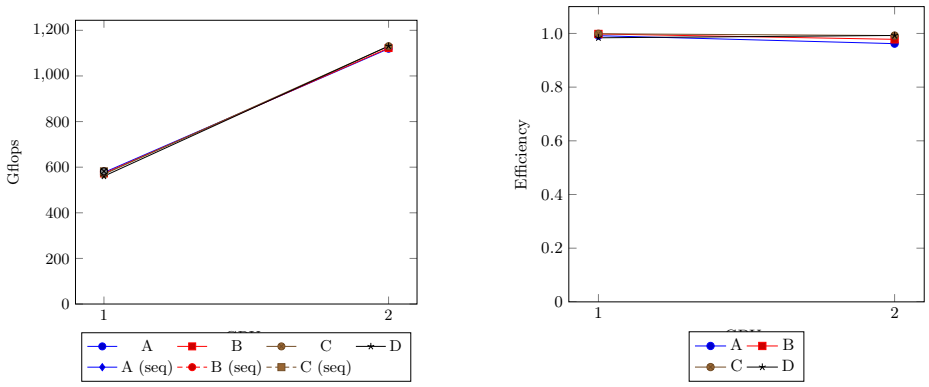
This computation takes $2mn(k-1)$ flops. The matrices A and C are perfectly distributed over the GPUs, but all GPUs need to access B. That means we need $O(n^2)$ data-movement and $O(n^3/G)$ computation for multiplying two $n \times n$ matrices on G GPUs. The communication does not scale, but for large n this term is dominated by the computation. So we expect better efficiencies for large n.

This is supported by the results of Figure 3, where we see efficiencies ranging from 84% for classes A and B, to 91% for class D on four GPUs. The efficiencies



(a) Compute rate (b) Efficiency

Fig. 3: Matrix Multiplication on Snellius.



(a) Compute rate (b) Efficiency

Fig. 4: Matrix Multiplication on Icis.

are even higher in Figure 4, where they range from 95% to 98%. The A6000 has a much higher bandwidth to FP64 ratio than the A100, so data-movement contributes even less to the compute rate.

It is interesting that class D does not seem to suffer from being out-of-core. This is likely caused by the unified memory overlapping this extra communication with computation [28].

4.4 N-Body Simulation

The all-pairs N-body algorithm computes how the position and velocities of a system of bodies affect each other through gravity. In larger applications this exact algorithm is typically used to compute the forces between bodies that

are near each other, and is combined with an approximate algorithm for bodies further away [4].

The main structure of this algorithm is described in Listing 1.6. We have $[N, 3]$ arrays `pos`, `accel`, and `velocity` to represent vectors in three-dimensional space, and an array of shape $[N]$ for the masses.

```
accel = {[i] -> sum{[j] ->
          acc(pos[i], pos[j], mass[j]}};
velocity = velocity + accel * dt;
pos = pos + velocity;
```

Listing 1.6: Structure of nbody-simulation in SaC. The function acc computes the acceleration caused by the force of particle j on particle i.

Most time is spent in the $O(N^2)$ computation of `accel`. The total cost is $20N^2 + 12N$ flops per iteration.

The performance characteristics are very similar to Matrix Multiplication, but even more pronounced. The computation is evenly distributed: for G in total, each GPU does $O(N^2/G)$ computation per iteration. All bodies need to be replicated on each GPU, so that is $O(N)$ communication per iteration. In contrast to matrix-multiplication, this computation is typically in-core.

As expected, we see high efficiencies in Figure 6, from 95% to 103%. This is the only benchmark where we see the efficiency exceeding 100%. This cannot be explained by variability in the measurements as the standard deviation is about 0.3%. The A6000 has 6 MB of L2 cache. Class A does not fit in the L2 cache of one GPU, but does fit in the combined cache of both. We suspect this is the cause.

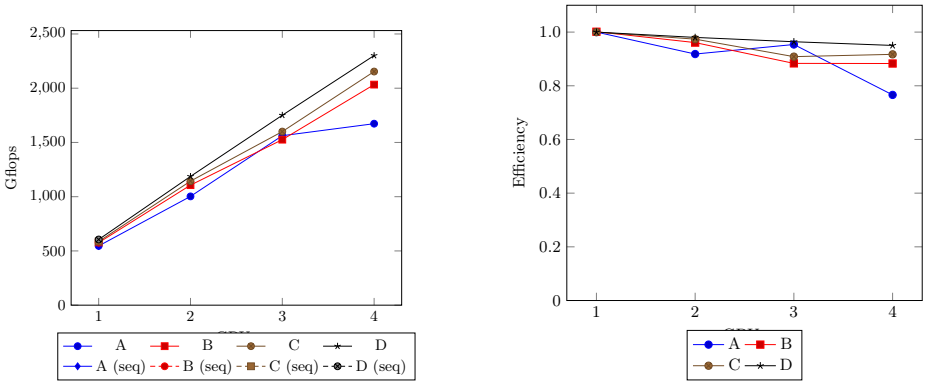
In Figure 5 we have a high efficiency of 95% for the largest problem class D. For the smallest problem class A this drops sharply for four GPUs, to 77%. The A100 has 6912 execution units per GPU, so 24768 in total. As problem class A has 20480 bodies, we can keep only 83% of these units busy.

4.5 Nine-Point Stencil

The final algorithm we benchmark is a 2D iterative nine-point stencil computation. This updates each point in a 2D grid with a weighted averages of its nine neighbours, where we wrap around the edges. This is a common pattern in finite-difference methods for solving partial differential equations [3].

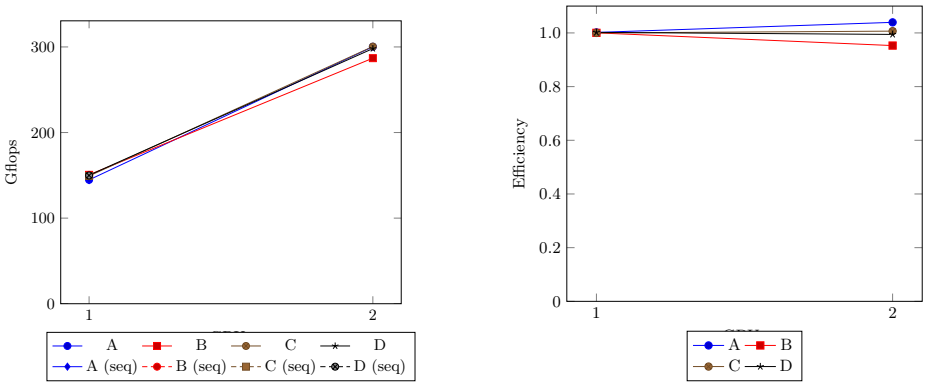
More recently, it has found application in convolutional neural networks [29, 32]. In contrast to the other two algorithms, each iteration does a small number of computations on each input element.

This tensor comprehension has nine index subsets because of the wrapping around the edges. In Listing 1.5 we have shown the generated code for the most computationally expensive one. Specifying these nine subsets is cumbersome, so we have used the more generic code of Listing 1.7. This can handle analogues in any dimension `d`, and also any shape `wshp` of weights. For example, if we want to take a weighted average of points at most two removed (Manhattan-distance) in



(a) Absolute performance (b) Efficiency

Fig. 5: N-Body simulation on Snellius.



(a) Absolute performance (b) Efficiency

Fig. 6: N-Body simulation on Icis.

three dimensions, we would have `d = 3` and `wshp = [5, 5, 5]` [1]. The compiler is able to recognize the `mod` operator and generate the index subsets [30].

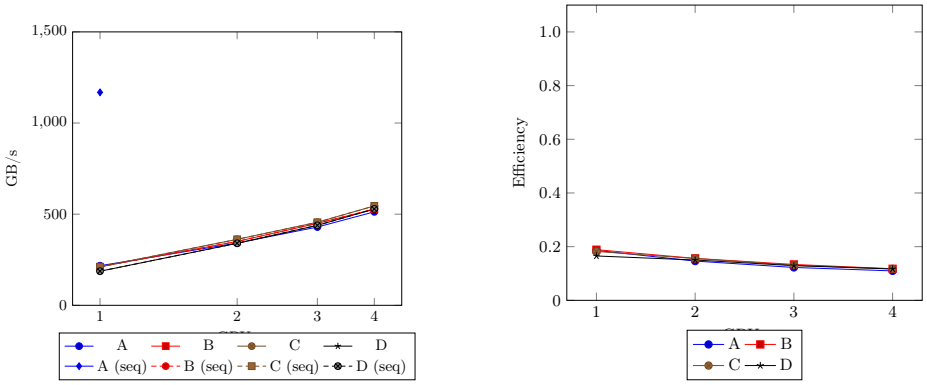
We use the flag `-maxwlur 9` to ensure the weighted average of nine points in the inner computation is unrolled. This takes $17n^2$ flops per iteration for an $n \times n$ array, but it is more common to measure bandwidth as data-movement is the limiting factor. This is $16n^2$ bytes per iteration.

For G GPUs, we need to do $O(n^2/G)$ data-movement within a GPU, and $O(n)$ data movement between GPUs. Therefore, we expect good results, similar to Matrix Multiplication and N-Body. However, Figure 7 and Figure 8 show no improvements over the `cuda`-baseline.

This is caused by a high cost for the first write to unified memory. When increasing the number of iterations, in Figure 9 and Figure 10, we do see speedups.

```
double[d:shp] relax(double[d:shp] x, double[d:wshp] w)
{
  return {iv -> sum({jv -> w[jv] * x[mod(iv + jv - wshp / 2, shp)]})
              | iv < shp};
}
```

Listing 1.7: Stencil computation in SaC.



(a) Bandwidth

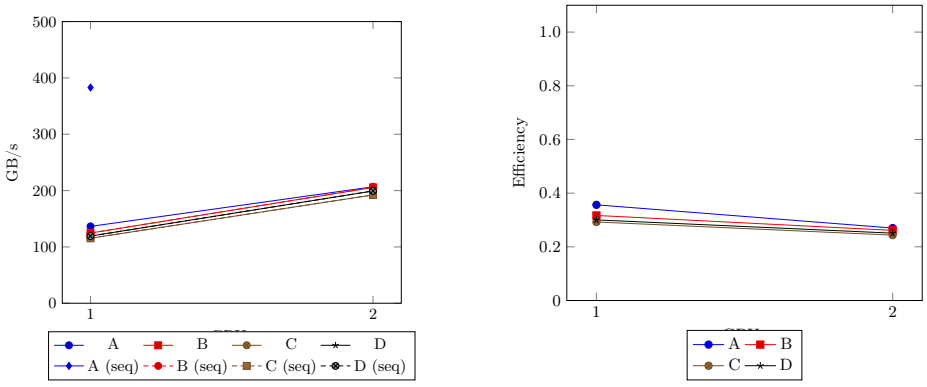
(b) Efficiency

Fig. 7: Stencil on Snellius, 10 iterations

We can model the execution time for T iterations as $Tx + c$, where x is the cost per iteration, and c the cost for allocation, deallocation and constant overheads. From the measurements for $T = 100$ and $T = 10$, we can solve for x. The bandwidth based on this number is graphed in Figure 11 and Figure 12. Here we have efficiencies between 48% and 81%. The efficiencies are lower than the other two benchmarks, especially for the smaller problem classes. This is because the synchronisation cost is not negligible in this benchmark. Each iteration takes 0.009 seconds for problem class D on Snellius. For comparison, in Matrix Multiplication we have one synchronisation every 35 seconds, and in N-Body one synchronisation every 96 seconds.

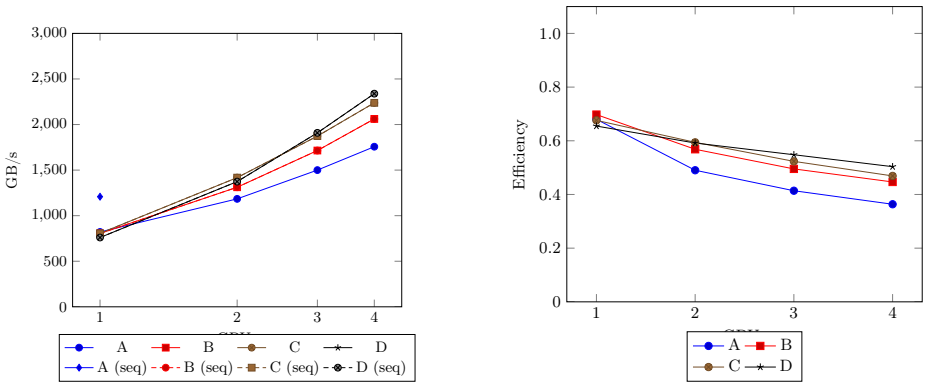
4.6 Effect of Memory Advice

We can only investigate the result of memory advice on small problems and few GPUs because we quickly run into time limits. We have done measurements on Snellius for two GPUs. For Matrix Multiplication we take three 1024×1024 matrices, for N-Body we take 32768 bodies and 10 iterations, for Stencil we take a grid of size 1024×1024 and 100 iterations. We divide the performance of either always inserting memory advice or never inserting memory advice by the runtime of our optimisation in Table 3.



(a) Bandwidth (b) Efficiency

Fig. 8: Stencil on Icis, 10 iterations



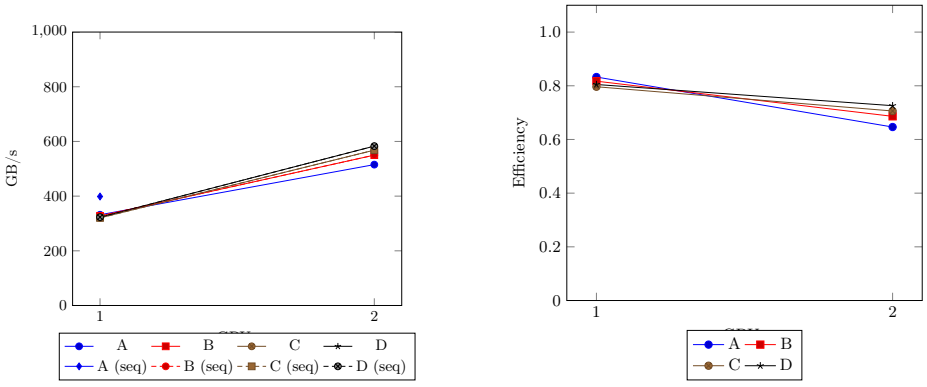
(a) Bandwidth (b) Efficiency

Fig. 9: Stencil on Snellius, 100 iterations

We see that always inserting advice gives good results for Matrix Multiplication and N-Body, but poor results for Stencil. If we never insert advice, the situation is reversed. Our optimisation makes the best choice in all three benchmarks. It is even a little bit faster in Matrix Multiplication and N-Body because we do not insert the advice for arrays where we do not need to.

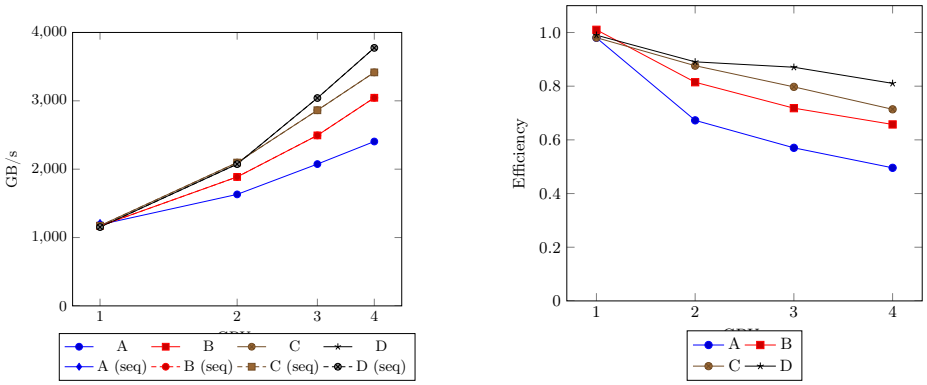
5 Related Work

Several studies have investigated compiling high-level specifications in functional array languages to multi-GPU code with manually-allocated data. One work presents envisioned compiler and runtime systems extensions for SaC for leveraging multiple multi-core CPUs and multiple GPUs without requiring additional



(a) Bandwidth (b) Efficiency

Fig. 10: Stencil on Icis, 100 iterations

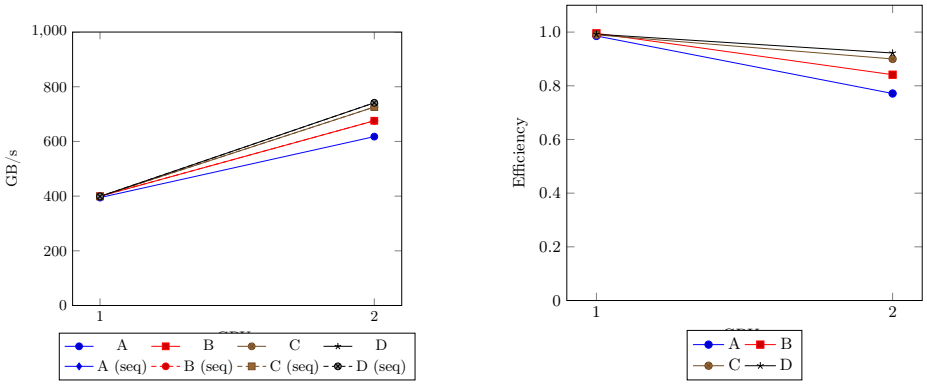


(a) Bandwidth (b) Efficiency

Fig. 11: Stencil on Snellius, without allocation and deallocation

programmer effort [9]. A new distributed-variable type is proposed to enable partial transfers of manually-allocated arrays between host and device. Furthermore, control structures are suggested to keep track of which parts of an array are present where, essentially implementing a cache coherency protocol. The authors evaluate a hand-coded prototype on a two-dimensional five-point stencil computations with two different combinations of parameters using GTX 580 GPUs. Their experimental results find that assigning almost all the workload to the GPU leads to the best results, and achieve a nearly two-fold speedup with two GPUs, but run into significantly less improvement with three or more GPUs.

The next study extends the Futhark compiler with a new internal streaming operator called the *husk* operator [15]. The author introduces a new type and semantics to further present transformations for the map and reduce second order



(a) Bandwidth (b) Efficiency

Fig. 12: Stencil on Icis, without allocation and deallocation

	Matrix Multiplication	**N-Body**	**Stencil**
Always	0.93	0.94	0.20
Never	0.09	0.72	1.00

Table 3: Speedup of always giving advice, never giving advice compared to our memory advice optimisation.

array combinators of Futhark, essentially allowing an intermediate Futhark program to expose opportunities for distributable data and operations. The study then also extends Futhark's CUDA C backend, leveraging the husk operator for multi-GPU execution and introducing a worker-thread runtime environment for concatenation or reduction of the results on multiple devices. Next, the entire implementation is tested on Futhark's benchmark suite, investigating both the effect of the husk operator in single and double GPU scenarios. In the multi-GPU case, their results vary from near-linear speedups to limited improvements, depending on the amount of data that has to be transferred between the two GPUs. In single GPU scenarios the overhead of the husk operator is generally negligible, except for some outlier applications — ranging from an improvement of roughly 1.7× to significant slowdowns of 2.5×.

Accelerate [26] adds multi-GPU support by adding a fissioning program transformation during the compilation process, essentially converting intra-kernel data parallelism to inter-kernel task parallelism. Furthermore, a runtime multi-GPU scheduler is added, which decides what task is executed on which device. This scheduler resembles a typical approach, introducing a worker thread for each CUDA device. These worker threads are handling both kernel launches and copying dependencies to the GPU they represent. The authors evaluate their implementation against three benchmarks on a two-GPU system, where two of those three benchmarks are Matrix Multiplication and N-Body. For the N-Body, they achieve linear speedups and even a 2.4× speedup when the

bodies fit in the cache of a single GPU. Their matrix multiplication benchmark achieves more modest speedups (around 1.3×), which the authors attribute to needing to communicate larger partial results and more expensive combination steps compared to their other benchmarks.

Another class of related work consists of more specialised transpilers and frameworks that tend to either implement a custom unified-memory protocol or extend CUDA's unified-memory driver [23, 20]. One such example is the framework SnuRHAC, which automatically and transparently distributes workloads to multiple GPUs across multiple nodes [20]. The authors extend CUDA unified memory with an additional Linux kernel module to achieve scaling beyond a single node of multiple GPUs. Within a node, kernel workloads are distributed by dividing the thread blocks in CUDA grids among multiple GPUs. Page faulting is also reduced by implementing both static and dynamic page-prefetching techniques. The implementation also includes heuristics to decide whether to execute on a single GPU instead if the workload *e.g.* contains too many shared-write accesses.

Finally, we take a look at related work that conducts manual experimentation with CUDA unified memory. One study evaluated the effect of CUDA performance hints for both in-core and out-of-core single-GPU scenarios on two platforms across six different applications [7]. On one platform, they find that memory advice only lead to benefits in in-core scenarios, whereas on the other platform exclusively in out-of-core situations. When it comes to page prefetching, they find that it leads to significant improvements for one system, but no benefits on the other. Overall, the authors conclude that unified memory is a promising technology and point out that more research into optimal advice placement would benefit programmer productivity.

Another study investigates a manual multi-GPU implementation using unified memory [11]. The authors analyse both programmability and performance, focussing on the effect of performance hints. Their study considers two types of memory advice: access patterns and placement of memory, and experiments are also conducted with prefetching. The authors observe speedups ranging from 1.10× to 1.85× on their selected benchmarks. They find that programmability is generally much improved due to not needing to do manual allocations nor add explicit memory transfers, but at the same time performance hints are necessary to achieve good performance. When it comes to performance, unified memory can introduce communication optimisations due to only initiating transfers that are strictly necessary for a given computation.

6 Conclusions and Future Work

We have implemented multi-GPU support for the language SaC by a light-weight extension of its existing unified memory compilation target. The crucial insight is that we can reuse the existing mechanism of specifying index subsets. To deliver on SaC's promise of performance, we insert memory advice based on access patterns.

Our benchmarks show promising results for both the N-Body and Matrix Multiply algorithms. With large enough problem sizes, we see efficiencies above 90% with respect to SaC's manually-allocated *cuda* target. For a nine-point stencil we can obtain efficiencies in excess of 80%, but only when we do not count allocation time. If the compiler cannot sufficiently reuse allocations, our approach yields efficiencies as low as 12%, even for large problem sizes.

Future work is needed to address the efficient support of reductions or fold on multiple GPUs. We use a basic device-wide synchronisation with expensive context switches between GPUs. This does not stop us from obtaining high efficiency on large problem sets, but fine-tuning this with event-based synchronization may lead to performance benefits on smaller problem sizes.

7 Acknowledgements

We thank Jordy Aaldering and Peter van Achten for their helpful comments. We also thank SURF in Amsterdam, the Netherlands, for giving us access to the supercomputer Snellius. We also thank the Dutch Research Council (NWO) for funding the computing time we used, under project EINF-10769.

References

1. Aaldering, J., Scholz, S.-B., and Gastel, B. v. Type patterns: Pattern matching on shape-carrying array types. In *Proceedings of the 35th Symposium on Implementation and Application of Functional Languages* (New York, NY, USA, June 2024), IFL '23, Association for Computing Machinery.
2. Andreetta, C., Bégot, V., Berthold, J., Elsman, M., Henglein, F., Henriksen, T., Nordfang, M.-B., and Oancea, C. E. FinPar: A Parallel Financial Benchmark. *ACM Trans. Archit. Code Optim. 13*, 2 (June 2016).
3. Barkley Rosser, J. Nine-point difference solutions for Poisson's equation. *Computers and Mathematics with Applications 1*, 3 (1975), 351–360.
4. Barnes, J., and Hut, P. A hierarchical O (N log N) force-calculation algorithm. *nature 324*, 6096 (1986), 446–449.
5. Chakravarty, M. M., Keller, G., Lee, S., McDonell, T. L., and Grover, V. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming* (New York, NY, USA, 2011), DAMP '11, Association for Computing Machinery, p. 3–14.
6. Chen, H., Kim, M., Razenshteyn, I., Rotaru, D., Song, Y., and Wagh, S. Maliciously secure matrix multiplication with applications to private deep learning. In *Advances in Cryptology–ASIACRYPT 2020: 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7–11, 2020, Proceedings, Part III 26* (2020), Springer, pp. 31–59.
7. Chien, S., Peng, I., and Markidis, S. Performance Evaluation of Advanced Features in CUDA Unified Memory. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)* (2019), pp. 50–57.
8. Cohen, M. B., Lee, Y. T., and Song, Z. Solving linear programs in the current matrix multiplication time. *J. ACM 68*, 1 (jan 2021).

9. Diogo, M., and Grelck, C. Towards Heterogeneous Computing without Heterogeneous Programming. In *Trends in Functional Programming* (Berlin, Heidelberg, 2013), H.-W. Loidl and R. Peña, Eds., Springer Berlin Heidelberg, pp. 279–294.
10. Glaser, J., Nguyen, T. D., Anderson, J. A., Lui, P., Spiga, F., Millan, J. A., Morse, D. C., and Glotzer, S. C. Strong scaling of general-purpose molecular dynamics simulations on GPUs. *Computer Physics Communications 192* (2015), 97–107.
11. González, M., and Morancho, E. Multi-GPU systems and Unified Virtual Memory for scientific applications: The case of the NAS multi-zone parallel benchmarks. *Journal of Parallel and Distributed Computing 158* (2021), 138–150.
12. Grelck, C., and Scholz, S.-B. SAC—A Functional Array Language for Efficient Multi-threaded Execution. *International Journal of Parallel Programming 34* (2006), 383–427.
13. Guo, J., Thiyagalingam, J., and Scholz, S.-B. Breaking the GPU programming barrier with the auto-parallelising SAC compiler. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming* (New York, NY, USA, 2011), DAMP '11, Association for Computing Machinery, p. 15–24.
14. Henriksen, T., Serup, N. G., Elsman, M., Henglein, F., and Oancea, C. E. 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* (2017), pp. 556–571.
15. Holst Larsen, S. Multi-GPU Futhark Using Parallel Streams, 2019.
16. Hsu, A. W. Accelerating information experts through compiler design. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming* (New York, NY, USA, 2015), ARRAY 2015, Association for Computing Machinery, p. 37–42.
17. Iverson, K. E. A programming language. In *Proceedings of the May 1-3, 1962, spring joint computer conference* (1962), pp. 345–351.
18. Janssen, N., and Scholz, S.-B. On Mapping N-Dimensional Data-Parallelism Efficiently into GPU-Thread-Spaces. In *Proceedings of the 33rd Symposium on Implementation and Application of Functional Languages* (New York, NY, USA, 2022), IFL '21, Association for Computing Machinery, p. 54–66.
19. Jeon, W., Ko, G., Lee, J., Lee, H., Ha, D., and Ro, W. W. Chapter Six - Deep learning with GPUs. In *Hardware Accelerator Systems for Artificial Intelligence and Machine Learning*, S. Kim and G. C. Deka, Eds., vol. 122 of *Advances in Computers*. Elsevier, 2021, pp. 167–215.
20. Jung, J., Park, D., Jo, G., Park, J., and Lee, J. SnuRHAC: A Runtime for Heterogeneous Accelerator Clusters with CUDA Unified Memory. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing* (New York, NY, USA, 2021), HPDC '21, Association for Computing Machinery, p. 107–120.
21. Knap, M., and Czarnul, P. Performance evaluation of Unified Memory with prefetching and oversubscription for selected parallel CUDA applications on NVIDIA Pascal and Volta GPUs. *The Journal of Supercomputing 75*, 11 (2019), 7625–7645.
22. Landaverde, R., Zhang, T., Coskun, A. K., and Herbordt, M. An investigation of Unified Memory Access performance in CUDA. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)* (2014), pp. 1–6.
23. Matsumura, K., Sato, M., Boku, T., Podobas, A., and Matsuoka, S. Macc: An openacc transpiler for automatic multi-gpu use. In *Supercomputing*

Frontiers: 4th Asian Conference, SCFA 2018, Singapore, March 26-29, 2018, Proceedings 4 (2018), Springer International Publishing, pp. 109–127.

24. McDonell, T. L., Chakravarty, M. M., Keller, G., and Lippmeier, B. Optimising purely functional GPU programs. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2013), ICFP '13, Association for Computing Machinery, p. 49–60.
25. Prades, J., Varghese, B., Reaño, C., and Silla, F. Multi-tenant virtual GPUs for optimising performance of a financial risk application. *Journal of Parallel and Distributed Computing 108* (2017), 28–44. Special Issue on Scalable Computing Systems for Big Data Applications.
26. Svensson, B. J., Vollmer, M., Holk, E., McDonell, T. L., and Newton, R. R. Converting data-parallelism to task-parallelism by rewrites: purely functional programs across multiple GPUs. In *Proceedings of the 4th ACM SIGPLAN Workshop on Functional High-Performance Computing* (2015), pp. 12–22.
27. van Beurden, P., Koopman, T., and Scholz, S.-B. Multi Gpu Ifl2024. https://gitlab.sac-home.org/sac-group/artefact-ifl2024-multi-gpu, 2024. Commit 123528eb3080a1e5f244efc43079f7047c8cee77.
28. van Beurden, P., and Scholz, S.-B. On Generating Out-Of-Core GPU Code for Multi-Dimensional Array Operations. In *Proceedings of the 34th Symposium on Implementation and Application of Functional Languages* (New York, NY, USA, 2023), IFL '22, Association for Computing Machinery.
29. Vedaldi, A., and Lenc, K. MatConvNet: Convolutional Neural Networks for MATLAB. In *Proceedings of the 23rd ACM International Conference on Multimedia* (New York, NY, USA, 2015), MM '15, Association for Computing Machinery, p. 689–692.
30. Verloop, M., Koopman, T., and Scholz, S.-B. Modulo in high-performance code: strength reduction for modulo-based array indexing in loops. In *Proceedings of the 35th Symposium on Implementation and Application of Functional Languages* (New York, NY, USA, 2024), IFL '23, Association for Computing Machinery.
31. Viessmann, H.-N., and Scholz, S.-B. Effective Host-GPU Memory Management Through Code Generation. In *Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages* (New York, NY, USA, 2021), IFL '20, Association for Computing Machinery, p. 138–149.
32. Šinkarovs, A., Viessmann, H.-N., and Scholz, S.-B. Array languages make neural networks fast. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming* (New York, NY, USA, 2021), ARRAY 2021, Association for Computing Machinery, p. 39–50.