Ref. Ares(2020)1269945 - 28/02/2020

HORIZON 2020
TOPIC FETHPC-02-2017
Transition to Exascale Computing

Exascale Programming Models for Heterogeneous Systems
801039

D5.3
Best practices document for porting applications to large-scale heterogeneous systems

WP5: Pilot applications and integration

Date of preparation (latest version): 29/02/2020
Copyright© 2018 – 2021 The EPiGRAM-HS Consortium

The opinions of the authors expressed in this document do not necessarily reflect the official opinion of the EPiGRAM-HS partners nor of the European Commission.

DOCUMENT INFORMATION

Deliverable Number	D5.3
Deliverable Name	Best practices document for porting applications to large-scale heterogeneous systems
Due Date	29/02/2020 (PM18)
Deliverable lead	KTH
Authors	Olivier Marsden (ECMWF) Ioan Hadade (ECMWF) Niclas Jansson (KTH) Steven W. D. Chien (KTH) Stefano Markidis (KTH) Martin Kuehn (Fraunhofer) Valeria Bartsch (Fraunhofer)
Responsible Author	Stefano Markidis (KTH) e-mail: `markidis@kth.se`
Keywords	Best Practices for Porting, Application Refactoring
WP/Task	WP5 /Task 5.2 and 5.3
Nature	R
Dissemination Level	PU
Planned Date	29/02/2020
Final Version Date	29/02/2020
Reviewed by	Luis Cebamanos (UEDIN), Dipti Shankar (Fraunhofer)
MGT Board Approval	YES

DOCUMENT HISTORY

Partner	Date	Comment	Version
ECMWF	15.11.2019	Draft version of the deliverable	0.1
ECMWF	13.01.2020	Exec. summary, introduction and gen. recomm.	0.2
ECMWF	21.01.2020	Merged ECMWF contributions with KTH	0.3
ECMWF	22.01.2020	Conclusions and other tweaks	0.4
Fraunhofer	27.01.2020	AI contribution	0.4
ECMWF	02.02.2020	Merge of AI, and general curation	0.5
ECMWF	24.02.2020	Minor amendments according to internal review	1.0

Executive Summary

This report collects experiences about application porting to large-scale heterogeneous systems, describing the difficulties encountered and how these were addressed, and provides best-practice suggestions for those embarking on similar endeavours.

The central focus for high-performance computing application porting, around which this report is built, is code refactoring. Important aspects to consider regarding refactoring are presented and organised by themes consistent with the Work Packages in EPiGRAM-HS, namely “Network”, “Memory”, and “Compute”.

Contents

1 Introduction 6

2 General recommendations 6

3 Network 7
3.1 Abstractions to the communication library 7
3.2 From coarse grain to fine grain parallelism 8
3.3 Hybrid implementations 8
3.4 Effect of parallelization strategies in AI applications 11
3.4.1 Data parallelism 11
3.4.2 Pipelining 12
3.4.3 Full model parallelism 13

4 Memory 15
4.1 Abstractions to the memory layout and access semantics 16
4.2 Memory allocation and alignment 17
4.3 Data structures and containers 17
4.4 Data migration across memory hierarchies and address spaces 18
4.5 Memory capacity 19

5 Compute 20
5.1 Optimised libraries 20
5.2 Heterogeneity 20
5.3 Load Balancing between compute nodes 22
5.4 Task-based programming 22

6 Conclusion 22

1 Introduction

Scientific disciplines are among the largest consumers of generally available high-performance computing resources. Indeed in Europe many large high-performance computing sites are dedicated solely to running scientific software from various disciplines. These centres have reached sizes and electrical power levels that mean that continued growth based on traditional homogeneous CPU-based cluster architectures is increasingly difficult.

Hardware heterogeneity is seen by many as an avenue to allow continued growth in computational capabilities at constant total power. This includes the currently popular GPU[1] accelerators, emerging classes of accelerators such as FPGAs[2], and hybrid chips combining strongly different architectures, such as the future EPI[3] chip.

Scientists developing and maintaining existing software and intending to run on the largest high-performance computing systems will be confronted with porting their software to such future heterogeneous systems and their software stack.

A number of aspects contribute to this porting exercise being fraught. Large scientific applications tend to be developed over long periods of time, by scientists whose background is often not computational science. As a result, the appetite for frequent large-scale changes to the software infrastructure and style is moderate, at best. Moreover, a majority of these applications are written in programming languages and models that do not currently allow native targeting of accelerators.

Porting strategies for such scientific applications therefore require careful thought and consideration. This report is an attempt at providing some input to those beginning this thought process in the future.

2 General recommendations

Porting a large and complex code to a new machine, and even more so a new architecture, can be a daunting prospect. Faced with the magnitude of the task and the difficulty of some of the problems which can be met along the way, it is easy to get lost or stuck before completion of the project. Scientists and programmers embarking on such an endeavour should consider establishing a plan of points to work on during the port.

Wherever possible, starting work on extracts from the main code, be they mini-apps or small kernels, is a useful approach. For any large application, a number of these kernels should be envisaged, in order to separate and isolate the algorithmic or technical building blocks of the full code [1][2]. Although the preparation of such simplified kernels might appear to be extra work, the advantages, such as reduced number of lines of code and ability to evaluate different approaches and implementations, should be large enough to compensate the additional effort.

[1] Graphical Processing Unit
[2] Field Programmable Gate Array
[3] European Processor Initiative

A similar approach was followed at ECMWF[4] through ESCAPE[5][3], a Horizon 2020 project funded under the FET-HPC[6] initiative. The work in ESCAPE led to the identification and extraction of so called "Weather and Climate Dwarfs", in reference to the prior work of Colella *et al*[1] but targeting key algorithmic motives in the weather and climate domain that exhibited distinct computational patterns such as spectral transforms, cloud microphysics, radiation scheme etc. These "dwarfs" were then ported on novel high-performance computing architectures such as GPUs and even optical processors with the aim to feed the experiences and best practices learned from the porting exercise back into operational weather codes such as the ECMWF IFS[7]. These ESCAPE dwarfs are also used as pilot applications in a number of other Horizon 2020 projects such as EuroEXA[8] as well as this particular project, EPiGRAM-HS which highlights their significant return on investment with regards to the time taken for their initial development.

The exercise of separating the code into its fundamental operations can also be a good opportunity to consider whether the existing implementation of algorithms in the full code contains spurious dependencies. Such dependencies might include unnecessary/redundant data movement, and synchronisations, *e.g.,* parallel barriers. Both of these are costly operations on large systems, and their removal should improve application performance and scalability. Once an adequate approach is found for each building block, these ported solutions can be reintegrated in the full code.

Finally, any plan for porting an application should at least consider the three main families of performance-sensitive operations in a code: network, memory, and compute operations. These will be discussed in more details in the following sections.

3 Network

Any application targeting large-scale HPC platforms requires distribution - of compute, or memory, or most often both. Distribution is carried out via communication over the HPC network, using any of a number of APIs (MPI, GPI, OpenSHMEM), which are either vendor-supported or open-source. Large-scale communication comes at a cost, both in terms of implementation effort and in terms of performance. Both of these aspects should be considered when porting software to future Exascale systems.

3.1 Abstractions to the communication library

For applications that aim to be ready for more than one heterogeneous platform, it is important to hide the details of the distributed parallel communications behind an abstraction layer. With a thoughtfully designed communication library, it becomes possible to envisage

[4]European Centre for Medium-Range Weather Forecasts
[5]Energy-efficient Scalable Algorithms for Weather Prediction at Exascale
[6]Future and Emerging Technologies in High Performance Computing
[7]Integrated Forecast System
[8]https://euroexa.eu/

replacing MPI with alternatives (*e.g.,*PGAS[9]) in cases where this would be recommended. Ideally, switching out the underlying communication mechanism should be possible without changes to the application layer.

For example, this is achieved in the IFS via the Message Passing Library (MPL). The use of MPL allows for greater flexibility both with respect to the underlying communication library (i.e., MPI/GASPI/OpenSHMEM) and type of message passing technique (i.e., blocking-standard, blocking-buffered, RMA Put/Get). Consequently, the evaluation of different libraries and communication primitives usually requires at most an entire recompilation of the code-base without any programming effort.

3.2 From coarse grain to fine grain parallelism

Large scientific applications have often grown organically from far smaller initial code-bases, that were first developed and run on single processing units. Implementation of parallelism in such applications might not have been considered by the original authors at the start of the project, and might have been undertaken by entirely different scientists. It is likely that the initial implementation of distributed parallelism was carried using blocking synchronous semantics. Where the algorithmic features allow it, a porting exercise can be the opportunity to move to dependency-driven asynchronous communication (non-blocking / one-sided). Benefits of this approach may be twofold, on one hand a reduction in global synchronisation, whose cost on a large heterogeneous system might be large, and on the other hand, the opportunity for overlapping the communication time with useful computations.

A similar approach has been evaluated in the IFS [4] through the exploration of one-sided communication primitives available in Fortran2008 co-arrays and the GASPI/GPI-2 one-sided API with the aim of addressing two points of contention. Firstly, threads executing in the context of OpenMP parallel regions were allowed to take part in communication, thus allowing for a larger degree of overlap and parallelism compared to the existing approach where all communications were funnelled through the master thread. Secondly, one-sided communication was used to implement on-demand halos for the advection process, leading to smaller amounts of data transferred. Early results on the Cray XK-7 (DoE Titan) machine were promising as illustrated in Figured 1. The use of Fortran2008 co-arrays led to a speed-up of up to 60% at 5 km horizontal resolution and at a scale of more than 220,000 processing cores.

3.3 Hybrid implementations

Given the tremendous amount of high-quality scientific software implemented using MPI over the past decades, it is highly likely that a scientific code has one or several dependencies to libraries whose parallelization is based on a message passing paradigm. Therefore, refactoring a large code base will be a costly and time-consuming process, not only due to the

[9]Partitioned Global Address Space

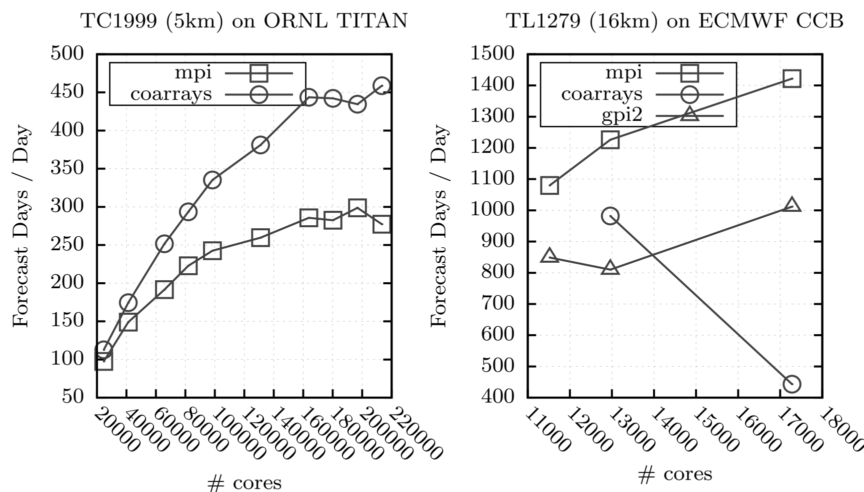


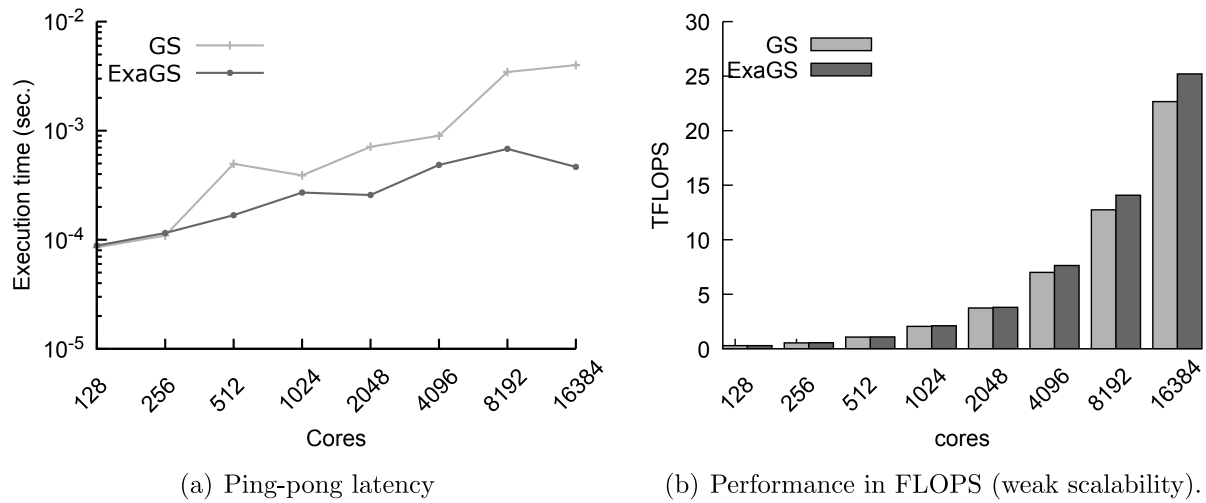
Figure 1: IFS model scaling runs on Cray XK-7 Titan (left) at 5 km horizontal resolution (TC1999L137) comparing baseline (MPI) and Fortran2008 co-arrays (coarrays) implementations and on the ECMWF Cray XC-40 (right) at 16 km horizontal resolution (TL1279L137) comparing baseline (MPI), Fortran2008 co-arrays (coarrays) and GASPI/GPI2 (GPI2) implementations.

complete rewrite of the main application, but also due to the necessary re-implementation of dependencies, and of course verification of correctness of the entire refactored application.

Therefore, a more sustainable way is to replace bits and pieces of key communication kernels with refactored versions, for example allowing for one-sided global view models to be used in certain key kernels, with traditional message passing in a larger bulk of an application. This approach also allows for a less time-consuming incremental refactoring of the code, with an easier adoption of emerging programming models in legacy message passing codes with the potential of good performance gains through the entire refactoring process.

An example of such hybrid parallelization is the refactoring of the gather-scatter kernels in Nek5000, where the original MPI based implementation was replaced with a one-sided PGAS implementation. With the much lower latency of the PGAS abstraction (Figure 2(a), the new communication kernels achieved around 10% better performance at scale (see Figure 2(b)).

Another good illustration of a successful hybrid implementation is the replacement of the linear algebra back-end in FEniCS. Sparse matrix assembly is one of the key bottlenecks, preventing good strong scalability for low order methods. The main issue at scale is that with very few elements per core, the necessary communication steps during assembly will be dominated by synchronisation cost from two-sided hand-shaking in MPI. By replacing (re-linking) the MPI-based linear algebra back-end in FEniCS with a new one written in the PGAS language UPC, performance was improved by almost an order of magnitude, see Figure 3, without any modifications in the main FEniCS code base, that still runs using MPI, and with several MPI dependencies alongside the new UPC based back-end.



(a) Ping-pong latency (b) Performance in FLOPS (weak scalability).

Figure 2: Performance results for Nekbone, using the new experimental communication kernel *ExaGS* compared to the standard MPI based *GS* kernel.

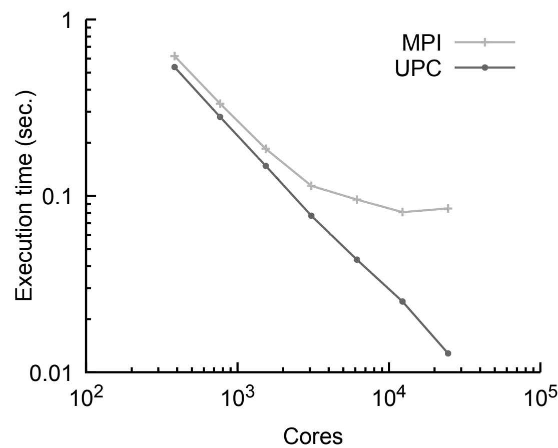


Figure 3: Sparse matrix assembly in FEniCS using one-sided communication (UPC) compared to leading MPI based implementations.

3.4 Effect of parallelization strategies in AI applications

While for traditional applications the parallelization strategies are set and well tested throughout the lifetime of the code, for novel applications such as the ones in the field of AI, special care needs to be taken to estimate which components need to be communicated and how much effort is added to the communication phase. We describe here the case for deep learning applications and provide a number of distinct parallelisation strategies as well as their impact on allowing for the overlap between communication and computation.

The following table gives an overview of the three kinds of important tensors in a DNN[10] which are the activations, the gradients of the activations and the gradients of the weights.

Component	Data parallelism	Pipelining	Model parallelism
activations	none	some, point-to-point	potentially all, all-to-all or allreduce
activation gradients	none	some, point-to-point	potentially all, all-to-all or allrededuce
model gradients	all, allreduce	none	depends

From the above table, it is clearly visible that the communication is centered on the weights in data parallelism while in pipelining and model parallelism, it is more centered on the activations and their gradients.

In pipelining, the communication of the activations takes place only between the edges of the partial DNNs while in model parallelism, it takes place between all the layers.

3.4.1 Data parallelism

The local gradients of the weights have to be synchronized after the backward propagation and before the update of the weights of the model. The reduction is an allreduce scheme. Figure 4 illustrates the overlapping strategy. As the gradients of the weights are calculated one by one during the backward propagation, their calculation can be overlapped with the allreduce of the previously calculated gradients. However, no overlapping can take place during the forward propagation which takes about one third of the total calculation time per iteration. Unfortunately the amount of gradients needs to be reduced often is not created linearly over time. Figure 5 illustrates this for the example of a 3D version similar to [5] of a U-Net [6]. The plot shows the number of bytes created during back-propagation versus the number of calculated FLOPS. If we assume a constant rate of FLOPS per second the abscissae can be understood as a time axis. It is striking that most of the gradients are calculated in the last 25% of the backward propagation which is about the last 15% of the total run-time of one iteration. To exploit the network bandwidth fully, the allreduces of the layers should be executed in parallel.

Data parallelism creates copies of the model on different ranks and has to take care on synchronization. Often in the literature this synchronization is done by a parameter server. This approach can easily lead to communication bottlenecks and should be avoided. A better approach would use an allreduce operation among the ranks. For this reason an implementation of an allreduce algorithm has been carried out in the EPiGRAM-HS project,

[10] Deep Neural Network

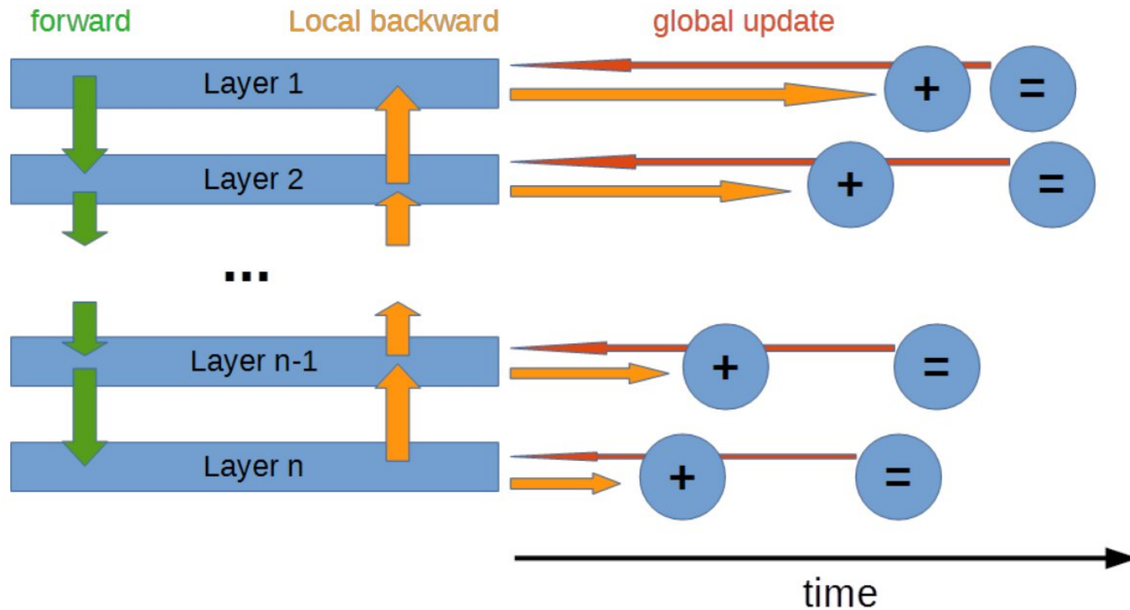


Figure 4: Overlapping strategy for data parallel approaches in DNNs.

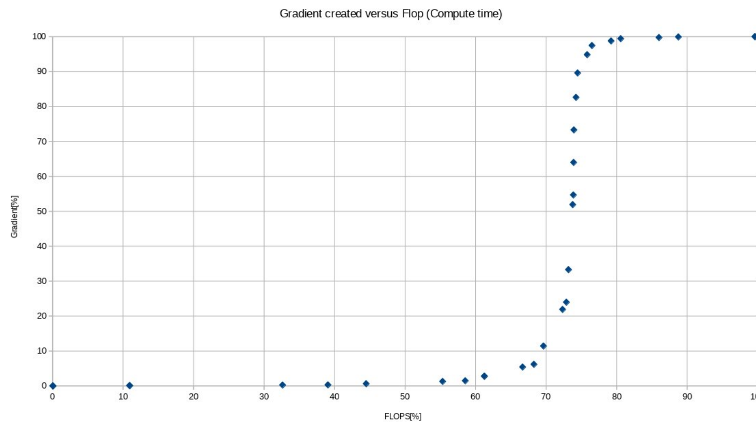


Figure 5: Aggregated sizes of the gradients versus the aggregated FLOPS of the backward propagation. DNN is a 3D version of U-Net similar to [5]. Individual data shown in figure 7.

based on the GPI communication library. For details please check the report for workpackage deliverable D2.3.

3.4.2 Pipelining

Figure 6 illustrates a possible pipelining scheme. Each step in the pipelining is divided into three micro slots. One for forward propagation of a micro batch, one for backward propagation of a gradient of the activations of a micro batch and one for backward propagation of a gradient of the weights. As the forward propagation and the backward propagation work on

different microbatches, the time usable for overlapping is around two thirds of each pipeline step. However, the pipeline steps are shorter than the runtime of a full iteration as each stage in the pipeline has only a part of the model.

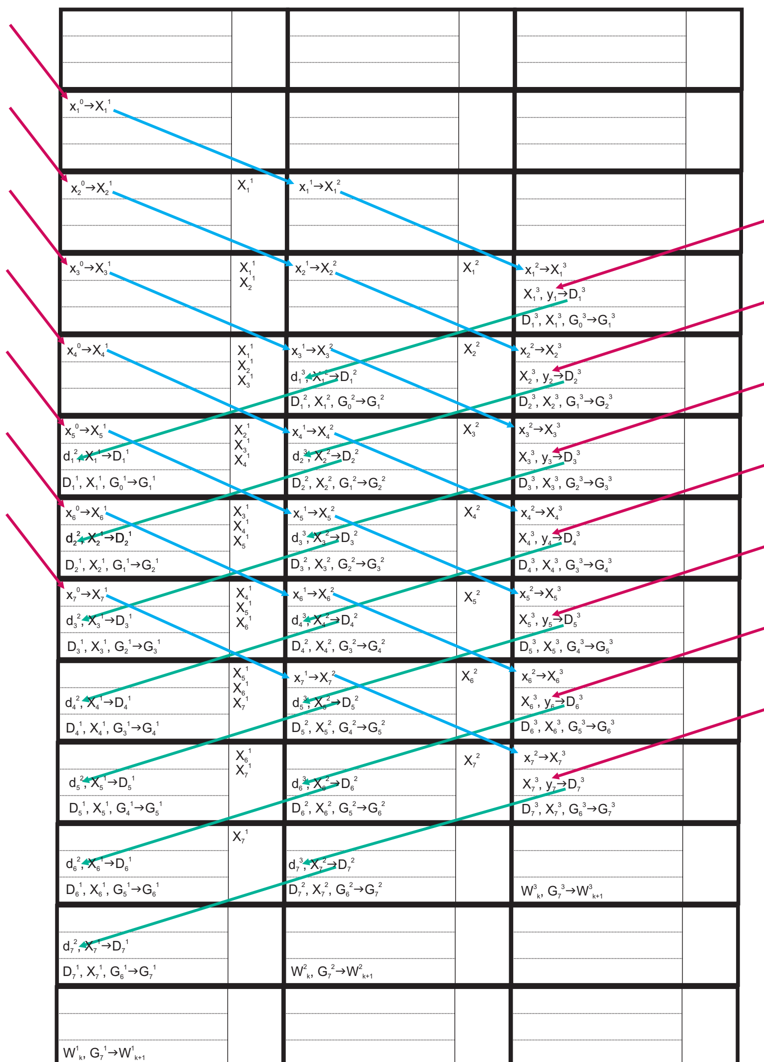


Figure 6: Pipelining scheme. Time axis is shown vertically, processing units and communication between them (e. g. GPUs) are shown horizontally. Parameter: $g = 3$ GPUs, $n_b = 7$ micro batches. Blue arrows mark communication of activations. Green arrows mark communication of gradients. Red arrows mark access to file system.

3.4.3 Full model parallelism

Full model parallelism is the most difficult approach do design and to implement. A careful design is necessary.

Splitting the DNN might be done by splitting the weights, the activations, or both. A good strategy depends on both the model and the goal, e. g. whether the goal is to overcome memory limitations or to speed up computation.

With fully connected layers, a split of weights and tensors at the same time is quite feasible even for higher rank counts. For convolutional layers the potential might be less prominent.

If in doubt the user should stick to a strategy that matches to their main goal. For example the ratio of weights to activations might depend strongly on the DNN at hand. If the goal is to save memory, splitting depends on whether the weights or the activations consume most of the memory.

Many DNNs are based mainly on convolutional layers, so some considerations on them are provided. The input and output tensors have different dimensions. The image dimension $h, w(, d)$, the number of channels c and the minibatch size n. Let h^i, w^i, d^i, c^i and n^i be the input dimensions of layer i and $\check{h}^i, \check{w}^i, \check{d}^i, \check{c}^i$ and $\check{n}^i$ the dimensions of its output. Let p_j be an image of the input with j depicting the channel id (within same minibatch entry) and $\check{p}_k = w_{jk} \star p_j$ the output image of channel k with the weights set w_{jk}. With this operation we can write the whole folding procedure similar to a matrix vector multiplication.

$$(\check{p}_k) = b_k + \sum_j (w_{jk}) \star (p_j). \tag{1}$$

The matrix (w_{jk}) is dense.

Cutting model without overlapping Let's assume the model is distributed without overlapping, i. e. each weight is stored on exactly one rank. In this case the incoming and outgoing tensors can only be split along the channel dimension. This is because any other dimension makes it necessary to duplicate weights and gradients of weights.

The user can choose whether to distribute the output channels without overlap. The output channels can then be assembled in an allgather scheme to all the ranks. Alternatively, the input channels can be split without overlap. The results can then be generated with an allreduce. The disadvantage of both approaches is that a full size input tensor or output tensor is needed on every rank, which might lead to memory problems. Splitting both can reduce the amount of memory needed for these activations.

Overlapping communication and computation might be done by splitting the own channels in chunks that are communicated while the next chunk is calculated. During the backward pass the gradients of the activations can be communicated while the gradients of the weights are computed.

Cutting model with overlap If the weights are duplicated over different ranks, the other dimensions can also be split. However, the drawback here is that the different copies of the weights require synchronization between each iteration, creating an additional communication overhead. The pattern here would be an allreduce as in the case of data parallelism.

The easiest split of the activations would be along the batch size but that delivers data parallelism again. So the only option left is cutting the tensors along the h^i, w^i, d^i dimension. In most cases the stencil of the convolutions is small, e. g. 3×3 or 1×1. So the interaction of voxels in the input with voxels in the output happens only in short distances. In this way, only the boundaries of the local domains have to be communicated between the ranks.

As an example, consider a 3×3 convolutional kernel, with one pixel overlap between parts. The slowest dimension of the images (e. g. h^i) is distributed by cutting them in slices. The overlap of each rank with its two neighbors would be $w^i \times d^i$ for each neighbor. The relative exchange overhead depending on the number of ranks p would be

$$o = 2\frac{p}{h^i}. \tag{2}$$

In the case of a 2 dimensional split (e. g. h^i, w^i) each rank would communicate with up to 8 neighbors. The relative communication overhead o would be

$$o = 2\sqrt{p}\frac{(h^i + w^i)}{h^i w^i}. \tag{3}$$

If $h^i = w^i$ we have

$$o = 4\frac{\sqrt{p}}{h^i}. \tag{4}$$

The advantage of this approach is that the amount of data that has to be communicated is rather small compared to the total size of the computed activations.

One option to reduce network latency would be to communicate every r layers in the DNN instead of every layer. This could be done by increasing the overlap to r. In this case the overhead would scale also approximately with a factor of r. In total this would not reduce the amount of data to be transferred but the frequency to do so. This might be important if network latency is an issue.

The overlapping potential of this approach is high, because the inner part of the activations can be computed while only the boundaries of the previous layer are communicated. During the backward pass the gradients of the activations can be communicated while the gradients of the weights are computed.

4 Memory

Along with communication bottlenecks, memory access costs are a major performance aspect for most HPC applications. Questions of access order and allocation alignment are often crucial to obtaining good performance. On heterogeneous systems, these questions are non-trivial, as different hardware will likely require different memory layouts in order to achieve optimal usage.

4.1 Abstractions to the memory layout and access semantics

High performance on modern processors is usually dependent on the efficient exploitation of the underlying memory hierarchy. On cache-based multicore processors, this requires that access patterns are performed at unit stride and exhibiting some measure of temporal locality via reuse. For streaming accesses, the Structure of Arrays (SoA) layout is favoured since calculations and accesses can be more easily vectorized. However, in the presence of indirection, the Array of Structures (AoS) layout can perform better due to the better exploitation of caches since adjacent elements are loaded together at cache line granularity [7]. In contrast, on massively parallel architectures such as GPUs, the SoA data layout is almost always preferred since the latency resulting from indirection can be hidden through the large number of in-flight threads.

The above might not hold true on future architectures that may implement deeper memory hierarchies or no caches at all. Consequently, abstractions to both memory layout and access semantics are imperative in order to allow for the seamless transition between memory layouts when moving across different architectures such as CPU to GPU. The alternative is to manually change the access semantics and data structure layout for each new architecture which, for an application of significant size, is an extremely time consuming and error prone process.

An example of memory layout abstractions in C++ can be seen in frameworks such as Kokkos[8] and is typically implemented by overloading the '()' operator combined with heavy use of templating constructs. These abstractions facilitate the separation between the access semantics and the actual storage by providing a consistent access API[11].

An important point to consider for the above is that loop nests might need to be swapped in the event that the underlying storage changes so that the fastest moving index is always streaming contiguously through the memory. Frameworks such as Kokkos can do this behind the scenes thanks to C++11 lambdas while other programming languages might require pre-processing machinery.

In the iPIC3D [9] application, arrays that are used in GPU kernels need to be allocated in a contiguous fashion to simplify the data transfer between the host and the device. On the CPU, the memory abstraction of iPIC3D provides a pointer chain (such as `array[x][y][z]`) that eventually de-references to the linear memory address and provides a subscript-based access instead of accessing the data array itself. This avoids the need to compute a flattened index and allows for cleaner code. On the GPU, helper device kernels are provided to compute flattened index inside the GPU compute kernel. This allows easy access to linear data arrays in the absence of multidimensional subscript access.

For applications written in Fortran, abstractions to the memory layout are much more difficult to implement due to the lack of features such as templates and lambdas. One avenue would be via the use of pre-processor directives although this is not standardized across compilers. Another one would be via module type bound procedures. The drawback of this though is that it relies heavily on the compiler to inline the function call that wraps

[11] Application Programming Interface

the index calculation for the memory access. If the compiler fails to inline this, the penalty of the function call for every memory access will lead to significant performance loss.

A workaround for the above is switching to a different memory access model such as one based on tiles where data is organised and consumed in contiguous regions (*i.e.,* tiles). In such model, even though the starting index is calculated using a function call, the penalty is negligible if the tile size is large enough that consuming and streaming through the data takes a significantly larger amount of time. However, a drawback to this approach is that it requires a significant refactoring effort since the entire looping structure of the application needs to change since the outer loop would iterate over the available tiles.

4.2 Memory allocation and alignment

Multidimensional arrays that are transferred to the GPU should be allocated in a linear fashion to allow for a more efficient and easier data transfer between the host and the device. In programming languages such as C, multidimensional arrays are often declared and allocated in a pointer-to-pointer fashion. A direct consequence of this is that the entire data structure is not allocated contiguously in memory. Thus, the array can not be copied directly to the memory of the accelerator via a single stream and may require multiple operations depending on the layout of the data structure (*e.g., for each row of the matrix*). A way to mitigate this is to allocate a single contiguous block of memory and use the memory abstraction layer to reference it accordingly. An example of this can be found in the multidimensional arrays that are used to represent grids in the iPIC3D [9] application. These are declared as one-dimensional arrays to ensure contiguity and therefore allow for more efficient memory copy operations (such as `cudaMemcpy()`).

Alignment restrictions on both CPUs and GPUs should be taken into account when performing allocation, copying and access. One solution is to use architecture specific memory allocators such as `cudaMalloc3D()`, `cudaMemcpy3D()` or `_mm_malloc`. These functions allocate memory in a linear fashion while taking into account hardware alignment which maximizes access speed. For example, to allow better data coalescing on GPUs [10] or more efficient vector load operations on CPUs. However, a potential drawback of these allocators is that they often allocate more memory than necessary to provide padding for alignment. Consequently, the implementation of a high level memory allocator should be followed, where hardware specific complex logic can be abstracted away and where such padding can be catered for. In iPIC3D, this is implemented by using a high level array class which allows the seamless switch between allocators (for example, between `malloc()` or `__mm_malloc()`) via header file definitions.

4.3 Data structures and containers

Complex data containers should be avoided as they may interfere with the underlying data allocations, creating performance issues. One example is the particle data structure in iPIC3D,

which uses an Array of Structure (AoS) layout based on a self-implemented STL[12] vector-like class. The vector class is mainly used to support the `push()` and `pop()` access methods. A feature of the STL vector class is that the underlying memory allocation is dynamically managed through book-keeping data. This means that an initial memory allocation can be re-allocated or resized during run-time. This is undesirable on the majority of systems due to memory allocation being an expensive operation. For example, iPIC3D uses CUDA pinned memory in the experimental GPU port [9]. Consequently, a run-time resize of pinned memory would be unacceptable for performance. A planned refactoring of iPIC3D is to use the refactored array class illustrated in the previous sections to store particle data. This allows a unified memory abstraction and avoids the complexity of dynamic data container.

Furthermore, GPU device kernels have a limited support of objects due to issues relating to deep copying. The issue is the same as why linear allocation should be used when allocating multidimensional arrays. For this reason, complex data containers should be avoided on the device as well. If they are to be used, this should be done by making use of existing architectural specific libraries. One example is the Thrust library for CUDA, which provides an STL-vector like data container which functions both on the host and device [11]. Additionally, the library supports simple and commonly used data operations, such as reduction. A drawback is that memory management is hidden in the library. Similar to STL-vector, the behavior cannot be easily controlled by programmers.

4.4 Data migration across memory hierarchies and address spaces

A single memory address space model can significantly reduce code complexity when managing complicated memory hierarchies between host memory and GPU memory. An example is the CUDA Unified Memory, which provides a single address space between GPU and host memory, and where data movement between the memory systems is managed through a run-time system using a page fault mechanism. With CUDA Unified Memory, it is not necessary for the programmer to perform explicit data copy (`cudaMemcpy()`) and pointer management (`float *grid, *d_grid;`) anymore. Moreover, the framework also allows for the over-subscription of GPU device memory by allowing an application to allocate as much memory as there is available on the host. However, a drawback of using the CUDA Unified Memory system is that run-time data movement due to page faults can severely impact device execution performance.

Another aspect that can affect performance when using the CUDA Unified Memory system is the alignment of data accesses. When a data item is accessed by a CUDA kernel, which does not currently reside in the GPU memory system, a page fault is generated. The fault is resolved by the CUDA runtime through page migration. However, when multiple items in a page are accessed by different threads in a warp, multiple page faults to the same page are generated. This increases the workload of the run-time system as it has to resolve more page faults for a single page. One solution is to perform padded access and process with the warp-per-page approach [12], where instead of one CUDA thread processing one

[12]Standard Template Library

item, one CUDA warp will process data on the entire page. This reduces the number of page faults due to many threads accessing the same page but the obvious drawback is reduced parallelism.

Best performance when using the CUDA Unified Memory run-time is obtained in the presence of regular access patterns which allow the run-time to better predict and optimize the data movement. Furthermore, features such as *advice*[13] can also be used to provide further information to the run-time on locality and access patterns. For example, data structures such as look-up tables, can be marked as *ReadMostly*, which indicates that update to the table is unlikely. Thus, the run-time can create multiple copies of the table on different memory systems to allow concurrent access without page faults, rather than migrating it between systems. In addition, programmers can choose to prefetch pages in bulk if they are not to be used in the future, to avoid page fault generation when a CUDA kernel is executed [13].

For workloads that exhibit a mixture of read and write patterns, run-times such as the CUDA Unified Memory system will unlikely outperform a judicious implementation based on the low-level primitives available on that particular architecture (i.e., cudaMemcpy/cudaMemcpyAsync for NVIDIA GPUs). A compromise, from a performance perspective, are abstractions layers such as the one developed in work package 3 of the EPiGRAM-HS project.

4.5 Memory capacity

For HPC applications, memory consumption is especially critical on GPUs which tend to have one order of magnitude less memory than the typical cluster node. For certain applications, implementing different parallelization strategies can result in better memory consumption due to less replication. For example, in the context of AI[14] applications, a reduction of memory consumption on distributed GPU nodes can be achieved by choosing a suitable parallelization strategy.

The following table depicts how different parallelization strategies for tensors can reduce memory consumption while distributed on p ranks.

Component	Data parallelism	Pipelining	Model parallelism
activations	none	fully	depends
activation gradients	none	fully	depends
weight gradients	none	fully	depends

In the case of deep learning obviously data parallelism doesn't reduce any memory consumption. Pipelining scales perfectly, except in cases of problems with homogeneity (see section 5.2). The model parallelism very much depends on the layers and the applied distribution strategy.

[13]CUDA mechanism for specifying constraints on memory characteristics
[14]Artificial Intelligence

5 Compute

Application developers who have put thought into communication patterns and requirements, and who have made sure that memory is allocated and accessed appropriately, have already got an application structure that should perform well on large-scale heterogeneous systems.

Nevertheless a few more points can be mentioned, that should contribute to efficient use of compute resources at the node level. These include using optimised libraries wherever appropriate, and addressing load balancing, both inter-node, and intra-node for example with a task-based paradigm.

5.1 Optimised libraries

It is highly recommended that applications make use of existing computational libraries (*i.e.,* BLAS, FFTW etc) instead of developing their own in-house implementation. Highly optimized implementations of these primitives can be found on the majority of high-performance computing architectures available today. However, rather than call these libraries directly, further abstractions might be necessary for interfacing in order to allow for further platform specific optimisations such as batching of FFT[15] calls on the GPUs etc.

5.2 Heterogeneity

Most models for application performance assume that the compute can be homogeneously distributed on the compute resources. In practice heterogeneities occur. The situation worsens with an increased heterogeneity. If implementations for multiple accelerators exist the execution time can vary with the type of accelerator used. An example for this behaviour is given for AI applications. Here we study the pipelining parallelization strategy introduced in section 3.4.2.

The following table depicts the imbalance in FLOPS and memory consumption for a 3D version of a U-Net [6] topology which is a popular topology for image segmentation. The topology is similar to [5]. Here 2 to 8 ranks are used and imbalance is defined as the worst case deviation from the ideal distribution in the pipeline. The additional FLOPS more or less directly transfer into higher runtimes.

# GPU	FLOPS	mem
2	10.7 %	3.7%
3	2.1%	44.7%
4	22.3%	89.9%
5	17.5%	120.7%
6	30.0%	158.2%
7	51.7%	201.3%
8	73.4	244.3%

[15] Fast Fourier Transforms

Taking into account that Deep Learning applications often use GPUs with a limited amount of memory as accelerators for their compute kernels, a possible strategy could be to distribute the pipeline equally. Concerning the heterogeneity of computation however, the situation gets worse when we distribute the pipeline for equal memory consumption. The distribution for equal memory consumption hurts the equal FLOPS distribution in this example.

# GPU	FLOPS	mem
2	17.1%	0.2%
3	68.3%	2.4%
4	90.5%	9.5%
5	109.2%	11.9%
6	81.1%	4.1%
7	111.3%	17.5%
8	145.4%	23.1%

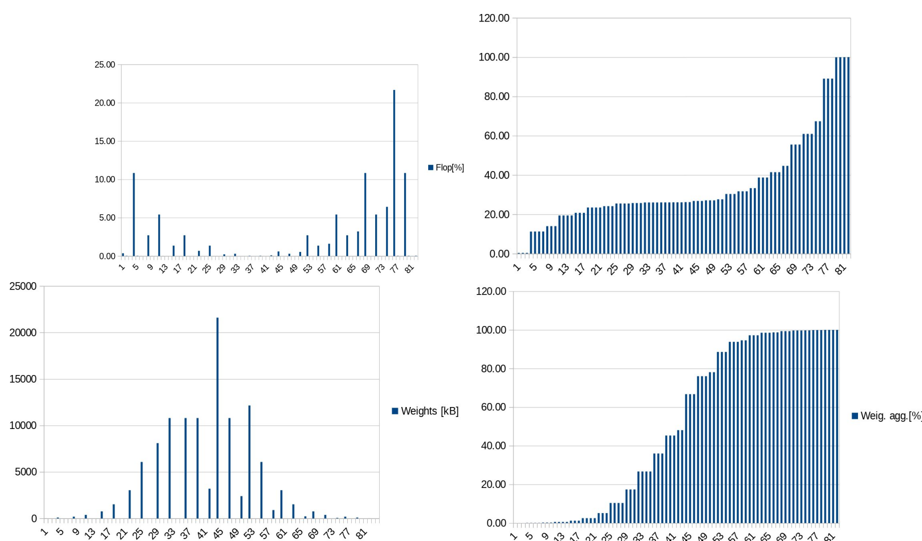


Figure 7: This image shows the distribution of the backward propagation of a 3D version of a U-Net similar to [5]. It is striking that e. g. layer number 77 takes around 22% of the total FLOPS and layer 45 around 20% of the total gradient size.

The reason for this is illustrated by figure 7 which depicts the distribution of FLOPS and calculated gradients of the weights per layer.

Thus it shows how the aspect of homogeneous FLOPS distribution and homogeneous memory distribution are often intertwined and one would need to combine several methods with each other in order to achieve the desired low memory usage and faster time to solution. In this case a hybrid approach when one would split only a few of the biggest layers with model parallelism might alleviate this problem.

5.3 Load Balancing between compute nodes

Load balancing is an important aspect arising from the described heterogeneities. But also frequent communication points between different ranks lead to more or less tight synchronization of the ranks. The more communication points there are the closer the synchronization. Tight synchronization leads to higher overheads in communication and load balancing.

The following table depicts how good the perspective is of a good load balancing with the example of the different parallelization strategies for AI applications.

	Data parallelism	Pipelining	Model parallelism
Synchronization between compute nodes	low	medium	high
Load Balancing	very good	difficult	some restrictions

Here one can see a correlation of the coupling between compute nodes and load balancing.

5.4 Task-based programming

The previous subsections have highlighted the need for load balancing in application. An easy way to guarantee load balancing is task-based programming. Task-based programming models define elementary units of computation, named tasks, and submit them to a runtime system scheduler, which subsequently maps them on the available computing resources, in such a way as to optimize some cost function (usually the overall execution time, but other factors such as energy consumption or memory subscription may also be taken into account).

Task-based approaches aim to reduce sequential regions and synchronization phases guaranteeing load balancing and an easy integration of heterogeneous resources. I.e. if an implementation of the task on several resources exists it can be taken into account by the task-based programming model as described at the example of GPI-Space in D4.2 and D4.3 which report on the design and implementation of a new version of GPI-Space that handles multiple implementations for heterogeneous systems.

6 Conclusion

This report has presented a number of best practices and experiences for porting scientific applications onto large-scale heterogeneous systems. These ranged from general recommendations to ones specific to the main themes of the EPiGRAM-HS project such as network, memory and compute.

With regards to general recommendations, we advocated for the development of mini-applications or the extraction of small kernels from the main application that best characterise its computational and memory access patterns. Thus, initial porting can be conducted using these smaller code-bases rather than the main application therefore preempting potential implementation pitfalls and finding the most optimal implementation early on.

On the network component, the best practices presented revolved around the implementation of suitable abstractions to the communication library which would allow for seamless transition between different programming models such as MPI and PGAS or even the usage of both via a hybrid implementation.

For memory, we advocated for similar abstractions as for the network but targeting the layout in memory of data structures and their associated access patterns. This was stressed to be important due to the fact that for certain access patterns, the optimal memory layout might differ between the CPU and the accelerator. Consequently, being able to switch between different layouts with minimal effort is very much required. Moreover, we presented the advantages and pitfalls of using single address space models, abstractions to memory allocation and alignments and specific advice pertaining to legacy applications.

On the compute theme, we argued for the need to use existing libraries for primitives such as matrix matrix multiply or FFT's since optimised versions of these will most likely exist for both CPUs as well as accelerators such as GPUs etc. As a result, an application using these rather than their own implementation will be much more easily ported onto newer high-performance computing architectures.

References

[1] P. Colella. Defining software requirements for scientific computing. DARPA HPCS Presentation, 2004.

[2] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: a view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[3] Andreas Müller, Willem Deconinck, Christian Kühnlein, Gianmarco Mengaldo, Michael Lange, Nils Wedi, Peter Bauer, Piotr K. Smolarkiewicz, Michail Diamantakis, Sarah-Jane Lock, Mats Hamrud, Sami Saarinen, George Mozdzynski, Daniel Thiemert, Michael Glinton, Pierre Bénard, Fabrice Voitus, Charles Colavolpe, Philippe Marguinaud, Yongjun Zheng, Joris Van Bever, Daan Degrauwe, Geert Smet, Piet Termonia, Kristian P. Nielsen, Bent H. Sass, Jacob W. Poulsen, Per Berg, Carlos Osuna, Oliver Fuhrer, Valentin Clement, Michael Baldauf, Mike Gillard, Joanna Szmelter, Enda O'Brien, Alastair McKinstry, Oisín Robinson, Parijat Shukla, Michael Lysaght, Michał Kulczewski, Milosz Ciznicki, Wojciech Piactek, Sebastian Ciesielski, Marek Błażewicz, Krzysztof Kurowski, Marcin Procyk, Pawel Spychala, Bartosz Bosak, Zbigniew Piotrowski, Andrzej Wyszogrodzki, Erwan Raffin, Cyril Mazauric, David Guibert, Louis Douriez, Xavier Vigouroux, Alan Gray, Peter Messmer, Alexander J. Macfaden, and Nick New. The ESCAPE project: energy-efficient scalable algorithms for weather prediction at exascale. *Geoscientific Model Development Discussions*, pages 1–50, jan 2019.

[4] George Mozdzynski, Mats Hamrud, Nils Wedi, Jens Doleschal, and Harvey Richardson. A PGAS implementation by co-design of the ECMWF Integrated Forecasting System (IFS). In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 652–661. IEEE, 2012.

[5] Fabian Isensee and Klaus H. Maier-Hein. An attempt at beating the 3d u-net, 2019.

[6] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation, 2015.

[7] Ioan Hadade, Feng Wang, Mauro Carnevale, and Luca di Mare. Some useful optimisations for unstructured computational fluid dynamics codes on multicore and manycore architectures. *Computer Physics Communications*, 235:305 – 323, 2019.

[8] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[9] Chaitanya Prasad Sishtla, Steven WD Chien, Vyacheslav Olshevsky, Erwin Laure, and Stefano Markidis. Multi-GPU Acceleration of the iPIC3D Implicit Particle-in-Cell Code. In *International Conference on Computational Science*, pages 612–618. Springer, 2019.

[10] CUDA C++ Programming Guide, Dec 2019. [Online; accessed 20. Jan. 2020].

[11] Thrust - Parallel Algorithms Library, Jul 2019. [Online; accessed 20. Jan. 2020].

[12] Maximizing Unified Memory Performance in CUDA | NVIDIA Developer Blog, Nov 2017. [Online; accessed 20. Jan. 2020].

[13] S.W.D. Chien, I.B. Peng, and S. Markidis. Performance Evaluation of Advanced Features in CUDA Unified Memory. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pages 50–57, Nov 2019.