Heterogeneous CPU-GPU Execution of Stencil Applications

Bálint Siklósi, István Z Reguly
Faculty of Information Technology and Bionics
Pazmany Peter Catholic University
Budapest, Hungary
Email: siklosi.balint@hallgato.ppke.hu, reguly.istvan@itk.ppke.hu

Gihan R Mudalige
Department of Computer Science
University of Warwick
Coventry, United Kingdom
Email: g.mudalige@warwick.ac.uk

***Abstract*—Heterogeneous computer architectures are now ubiquitous in high performance computing; the top 7 supercomputers are all built with CPUs and accelerators. Portability across different CPUs and GPUs is becoming paramount, and heterogeneous scheduling of computations is also of increasing interest to make full use of these systems. In this paper we present research on the hybrid CPU-GPU execution of an important class of applications: structured mesh stencil codes. Our work broadens the performance portability capabilities of the Oxford Parallel library for Structured meshes (OPS), which allows a science code written once at a high level to be automatically parallelised for a range of different architectures. We explore the traditional per-loop load balancing approach used by others, and highlighting its shortcomings, we develop an algorithm that relies on polyhedral analysis and transformations in OPS to allow load balancing on the level of larger computational stages, reducing data transfer requirements and synchronisation points.**

We evaluate our algorithms on a simple heat equation benchmark, as well as a substantially more complex code, the CloverLeaf hydrodynamics mini-app. To demonstrate performance portability, we study Intel and IBM systems equipped with NVIDIA Kepler, Pascal, and Volta GPUs, evaluating CPU-only, GPU-only and hybrid CPU-GPU performance. We demonstrate a $1.05 - 1.2\times$ speedup on CloverLeaf. Our results highlight the ability of the OPS domain specific language to deliver effortless performance portability for its users across a number of platforms.

***Keywords*—Stencil computations, hybrid CPU-GPU, load balancing, polyhedral analysis, tiling, heterogeneous scheduling**

I. Introduction

After the end of Dennard's scaling, hardware architectures have once again diversified, trying to improve performance by means of increased parallelism. In the domain of high performance computing, there are a number widely used architectures - to this day the most common are traditional x86 CPUs, albeit with dozens of cores, long vector units, and large caches. The second most commonly used architecture is the graphical processing unit (GPUs), and NVIDIA's Tesla line of products in particular. At one end, some of the largest supercomputers built rely on GPUs, such as Summit, Titan or Piz Daint, and at the other end GPUs are commonplace in desktops and workstations. Future systems are also likely to be heterogeneous in some form.

Due to the increasing complexity of these high performance architectures, it is becoming more and more difficult and arduous to optimise codes to work well on individual target hardware. Such optimisation requires an in-depth understanding of the target, which is something that many domain scientists, who develop these codes, do not have - and should not be required to obtain. The problem is further exacerbated by the need to achieve "performance portability"; have a code run fast across multiple hardware architectures - for production codes it is not an option to maintain several different versions of the same code. Finally, utilising several architectures *simultaneously* for the solution of the same problem is something even fewer manage to do - arguably because the expected performance improvements are only moderate, given the relative performance difference between CPU and GPU.

Nevertheless, there is a large body of research on optimising applications to utilise both architectures - an excellent overview is given by Mittal and Vetter [1], and in the Related Works section we discuss the relevant literature in detail. Generally speaking, perhaps the most common approach to utilising multiple architectures is through the use of task schedulers [2], [3], [4]. Yet, they have not been successful in the area of stencil computations, because one needs a very fine-grained control over the load balance and to minimise data movement, which would lead to many small tasks, but efficient execution requires a coarse granularity - the stencil sweep has to be scheduled in one or two chunks (due to threading and GPU kernel launch overheads).

Domain Specific Languages (DSLs) have emerged over the past decade trying to address the challenges of performance, portability, and productivity with a limited scope, focusing on a particular domain only, where given enough knowledge about the possible algorithmic patterns, optimisations could be applied automatically.

The Oxford Parallel library for Structured meshes (OPS) [5], [6] is a DSL, embedded in C/C++ and Fortran, which provides an abstraction for multi-block structured mesh computations. The key technique used by OPS is the separation of the description of computations from their parallel implementation - including the management of data. The user will write seemingly single-threaded code using the API of OPS, handing ownership of data to the library, which in turn can facilitate automatic parallelisation for both distributed-memory systems and shared-memory parallel architectures, such as multi-core CPUs and GPUs. Prior work has examined the efficiency of such a DSL approach, contrasting performance to hand-coded

implementations, and in other papers optimisations targeting various architectures were introduced. The most important feature of OPS for the present work is its capability for delayed execution of computations, which allows cross-loop analysis and optimisations.

In this paper, we introduce capability in OPS to utilise CPU and GPU architectures simultaneously, exploring and addressing the challenges that arise, and giving a contrasting performance analysis of the resulting algorithms. Specifically, we make the following contributions:

- We develop an algorithm in OPS that adaptively balances computational load between CPU and GPU on a per-loopnest basis, overlapping the computations with the memory copies between the two.
- We extend and develop the cache-blocking tiling functionality in OPS to enable load-balancing between CPU and GPU across a number of computational loops, thereby reducing the penalty of highly varying balance between subsequent loopnests.
- We carry out detailed performance analysis on a machine with Intel CPUs and NVIDIA GPUs, as well as a IBM Power8 system with NVIDIA GPUs, on a simple heat equation solver benchmark, as well as the significantly more complex CloverLeaf hydrodynamics code.

The rest of the paper is organised as follows: Section II discusses related work, Section III briefly introduces and discusses the OPS library, Section IV describes the heterogeneous scheduling algorithms that we developed, Section V presents the performance results, and finally Section VI draws conclusions.

II. Related Work

Here we give an overview of the general approaches and discuss works that address stencil computations. Perhaps, the most common approach to heterogeneous scheduling is task schedulers - a problem is broken up into tasks, defining their dependencies using a directed acyclic graph (DAG), which is then given to a runtime system which can automatically monitor performance and assign tasks to different processing units. Examples of such systems include StarPU [3], OmpSS [4] and Legion [2] - there are many examples of its uses for such purposes, such as QR factorisation [7] or image processing [8]. Others have developed more generic frameworks to automate the hybrid scheduling process, OpenCL being a prime target; runtimes can automatically intercept and distribute workgroups - examples include Maestro [9] and SnuCL [10].

Hybrid scheduling often relies on sending different kinds of tasks to different hardware; Stefanski [11] sends short length discrete Green's functions to the CPU and long ones to the GPU, GROMACS [12] uses the GPU to calculate non-bonded forces, and the CPU for bonded forces. Ricardo et al [13] keep the broad phase of a CFD problem with rigid body interactions on the CPU, and do the rest of the computations on the GPU.

Stencil computations are particularly challenging given that they already run very efficiently on both CPUs and GPUs, and the disparity between CPU and GPU performance make potential gains fairly small. Additionally, these applications are usually bulk-parallel - there are parallel sweeps over a computational grid separated by synchronisation points (usually for MPI halo exchanges). Nevertheless, there are still a number of papers optimising these computations - Venkatasubramanian and Vuduc [14] take a simple Jacobi iteration, partition it between CPU and GPU, and minimise the synchronisation points between CPU and GPU - achieving 1.2-2.5$\times$ speedups vs. the synchronous GPU implementation. Yang and colleagues [15], [16] work on Tianhe-1 to speed up Linpack as well as atmospheric simulations - in the latter they use the GPU to compute on the middle of each partition, and the CPU around the boundaries - overlapping CPU-GPU transfers. Sourouri et al [17] take a simple 3D 7-point stencil application and statically partition the workload between CPU and GPU - reporting a 1.1-1.2$\times$ speedup. PSkel is a DSL for algorithms using parallel skeletons - their authors demonstrate simple static load balancing between CPU and GPU on single-stencil applications in [18], reporting up to 1.28$\times$ speedups, and later work [19] discusses the use of time-tiling on the same single-stencil applications to reduce the communications requirements between CPU and GPU at the cost of redundant computations.

In summary, prior work on hybrid CPU-GPU stencil computations are limited in several respects; most works use a static load balance and only consider a single (often repeating) stencil. Arguably, these are difficult to generalise to larger applications. The few works that do consider realistic applications also only consider optimising for hybrid CPU-GPU execution on a per-loop basis. This is a crucial limitation; the optimal load balance can be significantly different between different loops, yet if they access some of the same data, the cost of data movement can negate the benefits. Our research differs from existing work and advances the state of the art in several respects - we demonstrate the limits of the per-loop balancing approach, and develop an algorithm that uses polyhedral analysis to partition a set of loops and thereby drastically reduce synchronisation points as well as communication requirements. We demonstrate our techniques on applications substantially more complex than most others.

III. The OPS Domain Specific Language

The Oxford Parallel library for Structured meshes defines an abstraction and an API, embedded in C/C++ and Fortran, for multi-block structured mesh computations. Its design and implementation is based on the philosophy of the separation of concerns; the programmer should focus on what to compute and not on how to compute it. Thus, given a high-level code written once with the OPS API, the library assumes responsibility for the parallelisation of computations as well as all data movement. The user first defines a collection of structured blocks, as well as data defined on them, and a number of access patterns, or stencils. Algorithms are then expressed as a sequence of parallel sweeps over blocks, accessing data with stencils and describing the type of access (read/write), as well as providing a "computational kernel" to be applied at each grid point. An example is shown in Figure 1; the `ops_par_loop` construct captures meta-data about the iteration space, the data accessed, the pattern of access and the mode of access. This description does not express how parallelism is to be orchestrated, and how data is to be stored

```
void copy(double *a, const double *b) {
  a[OPS_ACC0(0,0)] = b[OPS_ACC1(0,0)]; }
void calc(double *b, const double *a) {
  b[OPS_ACC0(0,0)] = a[OPS_ACC1(0,0)]
   + a[OPS_ACC1(0,1)] +a[OPS_ACC1(1,0)]; }
...
int range[4] = {12,50,12,50};
ops_par_loop(copy, block, 2, range,
                ops_arg_dat(a,S2D_0,"double",OPS_WRITE),
                ops_arg_dat(b,S2D_0,"double",OPS_READ));
ops_par_loop(calc, block, 2, range,
                ops_arg_dat(b,S2D_0,"double",OPS_WRITE),
                ops_arg_dat(a,S2D_1,"double",OPS_READ));
```

Fig. 1. An OPS parallel loop

and transferred - the computational kernel simply receives a pointer to the data associated with the current grid point.

During initialisation, the user hands all data to the library, which later can only be accessed through API calls. This allows OPS to automate both shared memory parallelisation, through code generation of CUDA/OpenACC/OpenMP/OpenCL code, as well as data movement - for example MPI decomposition and all communications. Code generation relies on parsing the user code for calls to the OPS API, and generating boilerplate code around the computational kernel provided by the user. Given the user's definition of access patterns and types the appropriate parallel patterns and implementations can be generated. The back-end library keeps track of accesses to data, allowing to manage multiple memory spaces (with GPUs) and MPI halo exchanges.

Prior work [5], [6] has demonstrated that such an approach can match the performance of hand-coded implementations and in some cases outperform it. We have also introduced a number of techniques and optimisations - for the present work the most relevant is the delayed evaluation technique and the polyhedral analysis and optimisation. Delayed evaluation allows OPS to gather information about a sequence of loops and reason about them together, allowing cross-loop optimisation techniques. Such an optimisation is a scheduling of computations that improves memory locality [20] - this depends on polyhedral analysis and transformations. We have demonstrated speedups of 1.5-3× on the CPU, on bandwidth-bound applications.

IV. Hybrid scheduling in OPS

Hybrid scheduling uses a combination of code-generation and backend logic. We rely on the delayed evaluation mechanisms in OPS; for each loop, we generate an OpenMP parallelised CPU code as well as CUDA code. At runtime, when the user code calls these, instead of executing one of them immediately, we send the loop descriptors and function pointers to the back-end. The back-end determines the required partitioning between CPU and GPU, makes the necessary data transfers to satisfy data dependencies, then calls the CPU code and the GPU code with altered iterations ranges.

We present two key algorithms in OPS that perform hybrid scheduling; the first that takes a per-loop view of load balancing, as done in related research [14], [17], and the second which uses polyhedral analysis to come up with an overlapped tiling execution schedule for the CPU and GPU that encompasses a number of loops. For the latter case, we rely on the aforementioned delayed evaluation technique in a more advanced manner: loop descriptors can be queued up until data needs to be returned to the user - this allows OPS to analyse and optimise these loops together.

A. Per-loop load balancing

For this approach, OPS keeps a historical record of relative performance between CPU and GPU for each loop $1..N$, identified by the different computational kernels that are applied. After each run the record is updated, and the ideal *split* (load balance) between CPU and GPU updated with a moving average calculation. At the same time, OPS keeps track of all the datasets' *clean* and *dirty* regions: what part of the computational grid is up-to-date on the CPU and the GPU.

The process for scheduling execution and satisfying data dependencies is outlined in Algorithm 1. When a loop i is scheduled to run next, OPS calculates the optimal $split_i$, and based on the access descriptors (read/write and access pattern), initiates the transfer of data to satisfy data dependencies for the given split. At the same time, it calculates the regions of the iteration space that are dependent on the ongoing data transfer and the regions that are independent (both for CPU and GPU). Therefore, while the copies are taking place, the independent regions can be scheduled by passing a modified iteration range through the function pointers. A synchronisation waits for the copies to finish, and the dependent parts can be scheduled. Finally, historical timing data is updated.

This algorithm has been described in the literature [14], [17], [16], although they were one-off implementations, whereas this is incorporated into OPS, and is applied without any changes to user code. This algorithm has the potential to incur very little overhead - as long as the copies can be completely overlapped with the independent computations. This has been the case for applications studied in the related works; simply because the same stencil was applied repeatedly, the ideal load balance did not change, and the copies only had to move data on the thin boundary region, as wide as the stencil used. However, this is not the case for multi-stencil applications, such as CFD or hydrodynamics solvers (e.g. CloverLeaf), where a large number of different stencil sweeps are required, each with its (often very different) ideal balance. In such cases, the data movement requirements between loops can become prohibitively expensive. This motivates the development of the following algorithm.

B. Tiled scheduling

In our previous work [20], we have developed a run-time algorithm that can analyse several loops together, and using polyhedral dependency analysis and transformations it can create an execution schedule that improves cache locality. This algorithm was also developed to apply *overlapped tiling* [21] across MPI boundaries - the idea is that halo regions are extended and a single MPI communication satisfies all the data dependencies required to execute all the loops that are tiled over - at the expense of redundant computations in these halo areas.

Algorithm 1 PER-LOOP HYBRID SCHEDULING

Input: loop index i, iteration range $[start, end)$, access descriptors
Calculate ideal load balance $split_i$ based on historical data
Calculate data dependencies given $split_i$ and access descriptors:
  For each dataset m = largest stencil offset in negative direction, p = in the positive direction
  For each dataset d, range to be uploaded to GPU $[up_{d,begin}, up_{d,end}) = [split_i - m, end) \cap [dirty_{gpu,begin}, dirty_{gpu,end})$
  For each dataset d, range to be downloaded to CPU $[down_{d,begin}, down_{d,end}) = [start, split_i + p) \cap [dirty_{cpu,begin}, dirty_{cpu,end})$
Calculate dependent regions $dep_{i,CPU}$,$dep_{i,GPU}$ and independent regions $indep_{i,CPU}$,$indep_{i,GPU}$:
  $[dep_{cpu,begin}, dep_{cpu,end}) = (\bigcup_{datasets\ d} [down_{d,begin}, down_{d,end})) \cap [start, split_i)$
  $[indep_{cpu,begin}, indep_{cpu,end}) = [start, split_i) \setminus dep_{cpu,begin}, dep_{cpu,end})$
  $[dep_{gpu,begin}, dep_{gpu,end}) = (\bigcup_{datasets\ d} [up_{d,begin}, uo_{d,end})) \cap [split_i, end)$
  $[indep_{gpu,begin}, indep_{gpu,end}) = [split_i, end) \setminus dep_{gpu,begin}, dep_{gpu,end})$
Upload and download data to satisfy dependencies
Simultaneously launch computations on independent regions $indep_{i,CPU}$,$indep_{i,GPU}$
Wait for uploads and downloads to finish
Launch computations on dependent regions $dep_{i,CPU}$,$dep_{i,GPU}$
Update historical timing data
Update dirty regions:
  For each dataset d written, dirty region on GPU is: $[dirty_{gpu,begin}, dirty_{gpu,end}) = [start, split_i) \cup ([dirty_{gpu,begin}, dirty_{gpu,end}) \setminus [up_{d,begin}, up_{d,end}))$
  For each dataset d written, dirty region on CPU is: $[dirty_{cpu,begin}, dirty_{cpu,end}) = [split_i, end) \cup ([dirty_{cpu,begin}, dirty_{cpu,end}) \setminus [down_{d,begin}, down_{d,end}))$

We have modified and applied the very same algorithms to hybrid CPU-GPU scheduling: we create a tile for the CPU as well as the GPU, do the data transfer between the two before executing the two tiles, and use redundant computations along the boundary. This allows the two tiles to execute asynchronously without any further communications and synchronisation (just as it does over MPI). For efficient data transfers between CPU and GPU we only split in the last dimension, so the data to be copied is contiguous in memory. Aside from reduced communication costs, this approach has a key advantage over per-loop scheduling: the load balance between CPU and GPU no longer has to be set for individual loops, rather for an entire loop chain, which enables averaging out relative performance differences on different loops.

The specifics of the algorithm are described in Algorithm 2, and the subroutine for creating tiles for CPU and GPU in Algorithm 3. Once the execution of a loop chain is triggered, due to some data being returned to the user (e.g. a reduction),

Algorithm 2 TILED HYBRID SCHEDULING

Input: Loop chain $l_1..l_N$, each with iteration range, access descriptors
Calculate ideal load balance $split$ based on historical data
Create CPU and GPU tile, and data dependency ranges:
  Call Algorithm 3, once for CPU, once for GPU
Transfer array chunks with a non-empty intersection of data dependency range and CPU/GPU dirty region (as in Algorithm 1)
Asynchronously launch loops in GPU tile
Launch loops in CPU tile
Reduce partial reduction results
Update historical timing data for this loopchain
Update dirty regions on CPU and GPU (as in Algorithm 1)

Algorithm 3 CREATING TILES

Input: Loop chain $l_1..l_N$, each with iteration range $[begin_{l_i}, end_{l_i})$, access descriptors, required $split$
Output: Iteration ranges for each loop $[tbegin_{l_i}, tend_{l_i})$, data dependency ranges for datasets $\forall d$: $[read\_dep_{d,start}, read\_dep_{d,end})$
for all loops i, backwards **do**
  {Start index for current loop in tile}
  if CPU tile **then**
    $tbegin_{l_i} = begin_{l_i}$
  else if GPU tile **then**
    {Extend tile, satisfying RAW dependencies}
    $tbegin_{l_i} = split$
    for all dataset d written in loop i **do**
      $tbegin_{l_i} = max(begin_{l_i}, min(tbegin_{l_i}, read\_dep_{d,begin}))$
    end for
  end if
  {End index for current loop in tile}
  if CPU tile **then**
    {Extend tile, satisfying RAW dependencies}
    $tend_{l_i} = split$
    for all dataset d written in loop i **do**
      $tend_{l_i} = min(end_{l_i}, max(tendl_i, read\_dep_{d,end}))$
    end for
  else if GPU tile **then**
    $tend_{l_i} = end_{l_i}$
  end if
  {Update read dependencies}
  for all dataset d accessed in loop i **do**
    if dataset d is written **then**
      $[read\_dep_{d,start}, read\_dep_{d,end}) = \emptyset$
    else if dataset d is read **then**
      Let m and p be the largest stencil offset in the negative/positive direction
      $[read\_dep_{d,start}, read\_dep_{d,end}) = [read\_dep_{d,start}, read\_dep_{d,end}) \cup [tbegin_{l_i} - m, tendl_i + p)$
    end if
  end for
end for

based on historical data, OPS calculates the ideal load balance for this particular loop chain. It then constructs a single tile for the GPU, calculating the necessary overlapping region and required data transfers, then does the same for the CPU part. The CPU part however may be further divided into smaller tiles to allow cache blocking tiling - this is not detailed here as the algorithm is quite extensive, the reader is referred to [20]. The halo transfers are then made, all the loops in the GPU tile are launched asynchronously, then the CPU proceeds to execute the CPU tile(s). Finally, dirty/clean regions are updated and partial reduction results are merged.

V. Results

In this section we first introduce the stencil applications being studied and the testing environments, then move on to analyse performance with the previously introduced methods.

A. Experimental setup

For performance measurements, we have tested our algorithms on two different applications. The first is a simple Poisson equations solver with two main functions in each iterations. The first step computes the new value of each cell from its neighbours using a five point stencil, and the second step updates all values regarding to the previous results. This algorithm has been well studied in the literature, and lends itself to good load balancing considering the few and simple computational loops. For testing, we run a 1000 iterations on various mesh sizes.

The second application is the CloverLeaf mini-app which involves the solution of the compressible Euler equations, which form a system of four partial differential equations. The equations are statements of the conservation of energy, density and momentum and are solved using a finite volume method on a structured staggered grid. The cell centres hold internal energy and density while nodes hold velocities. The solution involves an explicit Lagrangian step using a predictor/corrector method to update the hydrodynamics, followed by an advective remap that uses a second order Van Leer up-winding scheme. The advective remap step returns the grid to its original position. The original application [22] is written in Fortran and operates on a 2D structured mesh. It is of fixed size in both x and y dimensions.

The CloverLeaf application consists of 25 datasets defined on the full computational domain (200 bytes per grid point), and 30 different stencils used to access them. There are a total of 83 parallel loops spread across 15 source files, each using different datasets, stencils and "user kernels"; many of these include branching (such as upwind/downwind schemes, dependent on data). The source files that contain `ops_par_loop` calls include branching and end up calling different loops, dependent on e.g. sweep direction, with some code paths shared and some different for different sweeps, and often the pointers used refer to different datasets, depending on the call stack. A single time iteration consists of a chain of 153 subsequent loops. The full size of CloverLeaf is 4800 lines of code. It is a substantially more complex code than what is usually studied in the literature - the only paper we have found that considers the hybrid CPU-GPU scheduling of a larger stencil code is by Yang et. al. [15], however they only consider load balancing on a per-loop basis. For testing in this paper, we run for 50 time iterations.

For testing, we have used three workstations. The first has CentOS Linux 7.4.1708, Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz CPU with 20 cores, 2 threads per core and a NVIDIA Tesla K80 GPU. The second environment has Ubuntu 16.04.3 operating system, two POWER8E IBM CPUs each with 10 cores running at up to 3.69 GHz, 8 threads per core and 2 Tesla K40m GPUs. The third system has Debian 9, Intel(R) Xeon(R) CPU E5-2660 v4 @ 2.00GHz CPUs with 14 cores, 2 threads per core, an NVIDIA P100 and an NVIDIA V100 GPU. For in-depth testing, we only use the first two machines, and a single socket to avoid any NUMA issues. We then briefly discuss results obtained on the faster GPUs.

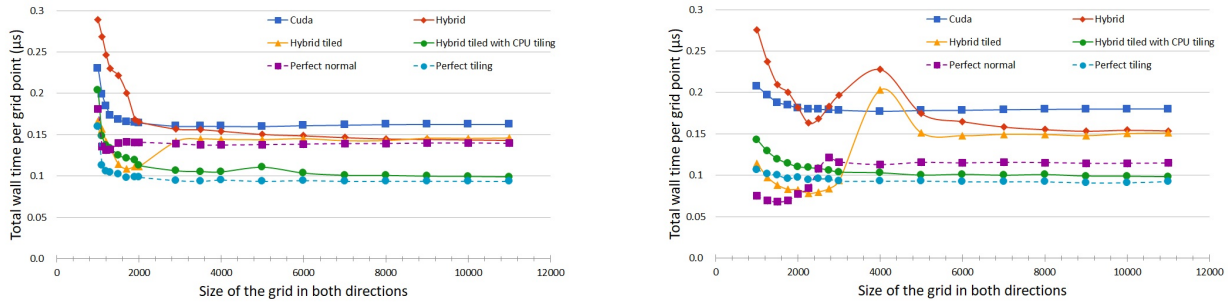
B. Analysis

Our results for the different applications on the different environments can be seen in Figures 2 and 3, and for a representative mesh size, in Table I. We have two performance baselines: the GPU-only version, and the CPU-only version. Since the CPU-only computed version is too slow to be shown in the same figure (on Intel), we show just the original GPU-only computed runtimes (*cuda*) for a measured reference - CPU-only performance is documented in previous work [20]. However, CPU times for a representative problem size are included in Table I.

Subsequently, we employ a trivial performance model to calculate the ideal hybrid CPU-GPU execution time - based on the relative performance difference between the two hardware, the fraction of workload to be assigned to the GPU will be $cpu/(cpu + gpu)$, and to the CPU $gpu/(cpu + gpu)$, where cpu and gpu indicate the CPU-only and GPU-only runtimes respectively. Multiplying this by the execution time of CPU or GPU only versions, we get $cpu * gpu/(cpu + gpu)$ for the theoretical minimum execution time utilising both hardware. Given that OPS has previously introduced support for cache-blocking tiling on the CPU, there are two CPU-only runtimes, with and without tiling; yielding two ideal runtimes, which we denote as Perfect normal and Perfect tiling. These estimates present a theoretical upper bound for performance, because they do not account for data movement between CPU and GPU and any further costs involved with load balancing.

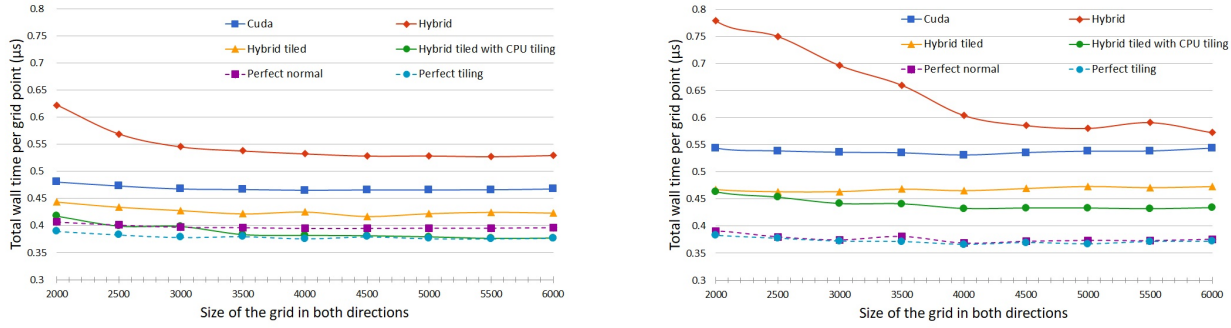
The *hybrid* curves represent the basic per-loop hybrid scheduling. Its extended version with the tiled scheduling is shown with the *Hybrid tiled* curves, and the final version which is combined with the cache blocking tiling scheduling within the CPU is the *Hybrid tiled with CPU tiling*.

We first study the performance on the Poisson code, shown in Figure 2 - with growing mesh sizes, the throughput of the GPU-only version stabilises as expected, and this behaviour is followed by all the other curves as well. Considering that the two repeating loops require similar balance, therefore the re-balancing between loops involves little data movement, the plain hybrid and the hybrid tiled versions perform almost the same at higher mesh sizes. However, at small mesh sizes, the hybrid tiled version is very fast - this is due to the CPU partition still fitting in on-chip cache. Performance is even better than the ideal at certain mesh sizes, which is due to the model not accounting for cache size. This effect is not



(a) (b)

Fig. 2. Simulation results for the Poisson application on the (a) Intel-NVIDIA and the (b) IBM Power8-NVIDIA environments



(a) (b)

Fig. 3. Simulation results for the CloverLeaf application on the (a) Intel-NVIDIA and the (b) IBM Power8-NVIDIA environments

TABLE I. TOTAL WALL TIME PER GRID POINT (μs) FOR LARGER PROBLEMS OF TWO APPLICATIONS ON THE DIFFERENT ENVIRONMENTS

Application	Environment	Grid size	GPU-only	Hybrid	Hybrid tiled	Hybrid tiled with CPU tiling	Perfect normal	Perfect tiling	CPU-only	CPU-only with tiling
Poisson	Intel-K80	8000*8000	0.1623	0.1450	0.1429	0.1007	0.1394	0.0936	0.9823	0.2211
CloverLeaf	Intel-K80	4000*4000	0.4647	0.5324	0.4252	0.3819	0.3949	0.3758	2.6295	1.9653
Poisson	Power8-K40	8000*8000	0.1794	0.1552	0.1492	0.1007	0.1153	0.0922	0.3226	0.1897
CloverLeaf	Power8-K40	4000*4000	0.5308	0.6041	0.4653	0.4322	0.3686	0.3660	1.2056	1.1791
Poisson	Intel-P100	8000*8000	0.0619	0.0659	0.0766	0.0473	0.0579	0.0443	0.8898	0.1561
CloverLeaf	Intel-P100	6000*6000	0.1694	0.2664	0.1798	0.1539	0.1564	0.1421	2.0387	0.8811
Poisson	Intel-V100	8000*8000	0.0434	0.0500	0.0628	0.0366	0.0414	0.0340	0.8898	0.1561
CloverLeaf	Intel-V100	6000*6000	0.1114	0.2110	0.1231	0.1050	0.1056	0.0989	2.0387	0.8811

TABLE II. ACHIEVED BANDWIDTH (GB/S) FOR THE TWO APPLICATIONS

Application	Environment	GPU-only	CPU-only	Best CPU+GPU
Poisson	Intel+K80	183.6	134.8	296.0
CloverLeaf	Intel+K80	167.3	39.3	201.6
Poisson	Power8+K40	166.1	157.1	296.0
CloverLeaf	Power8+K40	145.2	65.3	178.4
Poisson	Intel+P100	479.8	190.9	296.0
CloverLeaf	Intel+P100	458.9	50.68	500.3
Poisson	Intel+V100	684.3	190.9	296.0
CloverLeaf	Intel+V100	697.9	50.68	735.3

observed on the plain hybrid version, due to the much higher number of synchronisations. At larger problems, on the Intel platform the hybrid and hybrid tiled versions perform within 5% of the ideal. On the Power8 platform there is still a 25% gap between the hybrid version and the ideal speedup, which is due to a yet unexplained slowdown of the CPU side of computations. Enabling cache blocking tiling on the CPU side leads to a significant performance improvement - on Intel we see a 29% performance gain and get within 5% of the ideal, and on Power we see a 32% improvement, getting within 8% if the ideal. In total we see a $1.62\times$ speedup over the GPU-only version on the Intel platform, and $1.78\times$ on the Power platform.

Next, we move on to studying the much more complex CloverLeaf application. We can immediately observe in Figure 3 that the hybrid version - which load balances on a per-loop basis, performs very poorly. This is simply because subsequent loops often require a very different load balance between CPU and GPU, because of their different relative performances. Data movement can no longer be overlapped with computations, and we experience a slowdown - on Intel

19%, on Power 15-120%. This is where the communication-avoiding tiled approach can drastically improve the situation: OPS needs to communicate and synchronise between CPU and GPU much less frequently. This also means that the CPU and GPU parts of individual loops may run asynchronously with respect to each other, allowing load balancing on a much coarser level; all the loops that are tiled across, which in case of CloverLeaf is an entire time iteration consisting of 153 subsequent loops. On Intel, the hybrid tiled version is within 8% of the ideal, and on Power it is within 28%. Enabling cache blocking tiling on the CPU, we get within 2% of the ideal on Intel, and on Power within 17%. In total there is a $1.21\times$ speedup over the GPU-only version on the Intel platform, and a $1.25\times$ on the Power platform.

On newer generation GPU hardware, such as the Pascal P100 and the Volta V100, the performance difference between these and the older generation (Broadwell) CPU is much higher. As shown in Table I, compared to the plain CPU version, on the Poisson application the P100 is $14\times$ and the V100 is $20\times$ faster than a single socket of the CPU - although this is significantly improved by enabling the CPU-side cache blocking tiling optimisation - to $2.5\times$ and $3.5\times$ respectively. Compared to the 1.1-1.4$\times$ difference (with tiling enabled) on the Intel-K80 and Power8-K40 platforms, this is a significant jump - and naturally means that there is much less performance improvement to be had from heterogeneous execution. On the Poisson application, heterogeneous execution with the CPU and the P100 GPU yields a $1.3\times$ speedup (within 7% of the ideal), and with the V100 a $1.18\times$ speedup (within 8% of the ideal). On the more complex CloverLeaf application, we have more significant overheads, the P100+CPU achieves a $1.097\times$ speedup over the P100 (8.4% higher than ideal), and with the V100+CPU there is a speedup of factor $1.055\times$ (6.7% higher than ideal).

Absolute performance metrics - achieved bandwidth (GB/s) are presented in Table II, for the best CPU (with tiling), GPU and hybrid CPU-GPU variants. It is clear that on the Intel platform, for both applications, the achieved bandwidth of the hybrid version is close to the sum of the CPU and GPU-only versions. For the Power platform however, there is a loss of CPU performance when both CPU and GPU are used, particularly on CloverLeaf, therefore the improvement is more modest.

A more interesting aspect of the above absolute performance results is how one would quantify performance portability for applications executed in a hybrid manner on multiple processor architectures, such as done in this paper. Framing this question, in terms of the recent performance portability metric by Pennycook et. al. [23] appears to be not very intuitive. Architectural efficiency of the hybrid application could be computed easily by looking at the peak performance of each of the processors (in this case the bandwidth) and using its sum as the peak performance of the hybrid platform. Then performance portability of the hybrid application can be computed using this peak. However, the insights that can be gained appears to be limited. For example, it is obvious from Table II that for the POWER platform, the loss of bandwidth on the CPU when executing both CPU and GPU will result in lower architectural performance portability. The converse insight is when fully hybrid executable improves the bandwidth performance of each platform than the sum of each executed on their own separately. Both cases give insights into exploring how to gain further absolute performance. Performance portability based on application efficiency as detailed in [23] will require a hand tuned implementation of the hybrid application (or indeed the hybrid application implemented with some other portability framework) to ascertain if a high percentage of the best performance can be achieved with OPSs hybrid executable. Currently to our knowledge such a hand-tuned version of these applications does not exist, nor are there any other portability frameworks capable of achieving this.

VI. Conclusion

In this paper we have explored the techniques and performance for the heterogeneous CPU-GPU scheduling of stencil applications using the OPS framework. We have shown that the per-loop load balancing and scheduling approaches used in the similar works do not generalise to complex applications, and we introduced a technique that relies on lazy evaluation and polyhedral analysis/scheduling to address this problem. By being able to asynchronously schedule a large number of loops, it is possible to load balance for complete computational phases, that include many loops, instead of individual loops. This approach also significantly reduces the amount of data that needs to be communicated between CPU and GPU. The development of such algorithms is highly non-trivial - arguably it is not reasonable to enmesh such constructs in science code, as it would highly degrade its maintainability and readability. Productivity is therefore an important concern when considering such optimisations; there is a definite need for higher-level approaches to implementing science code which will later are capable of supporting these transformations.

We have evaluated our algorithms on a simple Poisson example as well as the substantially more complex CloverLeaf hydro mini-app, and tested them in Intel and IBM systems equipped with NVIDIA GPUs. We have demonstrated a 1.2-1.6$\times$ speedup over GPU-only execution by adaptively load balancing between CPU and Kepler-generation GPUs, and relying on CPU cache blocking tiling techniques already built into OPS. Testing with newer generation GPUs shows diminishing returns - the relative performance difference between GPU and CPU are much higher, and therefore the achieved speedups are lower.

Looking ahead, there are two key trends to consider. It is clear, that while there may be a performance advantage when one GPU is paired with one CPU socket, with higher compute density (more GPUs, same number of CPUs), the expected improvements all but disappear - for all intents and purposes the CPU just becomes something that feeds the GPUs. Second, the relative performance difference between the GPU and the CPU will always fluctuate - with the introduction of Skylake, Intel CPUs got a significant bump in bandwidth as well as compute - but at the same time, the memory bandwidth of Power8+ and Power9 CPUs is actually less than the first-generation Power8 CPUs which used the Centaur memory modules. It is therefore very much system-, and somewhat problem-dependent whether heterogeneous scheduling can deliver worthwhile performance improvements. Given that this research was done in the context of OPS, the user can simply select a heterogeneous version,

and test whether it can improve upon the performance of the GPU-only version.

In summary we have demonstrated the ability of OPS to take a user code written once using its high level API, and automatically apply such complex optimisations as heterogeneous scheduling. This once again underlines the utility of DSLs to deliver performance portability across a diverse set of architectures without user intervention.

ACKNOWLEDGMENTS

István Reguly was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences. Project no. PD 124905 has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the PD_17 funding scheme. The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013). OPS was developed under the UK EPSRC project entitled "Future-proof massively-parallel execution of multi-block applications" (EP/K038567/1). The authors would like to acknowledge the use of the University of Oxford Advanced Research Computing (ARC) facility in carrying out this work. http://dx.doi.org/10.5281/zenodo.22558

REFERENCES

[1] S. Mittal and J. S. Vetter, "A survey of cpu-gpu heterogeneous computing techniques," *ACM Comput. Surv.*, vol. 47, no. 4, pp. 69:1–69:35, Jul. 2015. [Online]. Available: http://doi.acm.org/10.1145/2788396

[2] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 66:1–66:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389086

[3] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, vol. 23, pp. 187–198, Feb. 2011. [Online]. Available: http://hal.inria.fr/inria-00550877

[4] J. Planas, R. M. Badia, E. Ayguad, and J. Labarta, "Self-adaptive ompss tasks in heterogeneous environments," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, May 2013, pp. 138–149.

[5] I. Z. Reguly, G. R. Mudalige, M. B. Giles, D. Curran, and S. McIntosh-Smith, "The ops domain specific abstraction for multi-block structured grid computations," in *2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, Nov 2014, pp. 58–67.

[6] G. R. Mudalige, I. Z. Reguly, M. B. Giles, A. C. Mallinson, W. P. Gaudin, and J. A. Herdman, "Performance analysis of a high-level abstractions-based hydrocode on future computing systems," in *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, S. A. Jarvis, S. A. Wright, and S. D. Hammond, Eds. Cham: Springer International Publishing, 2015, pp. 85–104.

[7] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov, "Qr factorization on a multicore node enhanced with multiple gpu accelerators," in *2011 IEEE International Parallel Distributed Processing Symposium*, May 2011, pp. 932–943.

[8] F. Lecron, S. A. Mahmoudi, M. Benjelloun, S. Mahmoudi, and P. Manneback, "Heterogeneous computing for vertebra detection and segmentation in x-ray images," *Journal of Biomedical Imaging*, vol. 2011, pp. 5:1–5:12, Jan. 2011. [Online]. Available: http://dx.doi.org/10.1155/2011/640208

[9] K. Spafford, J. Meredith, and J. Vetter, "Maestro: Data orchestration and tuning for opencl devices," in *Euro-Par 2010 - Parallel Processing*, P. D'Ambra, M. Guarracino, and D. Talia, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 275–286.

[10] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "Snucl: An opencl framework for heterogeneous cpu/gpu clusters," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12. New York, NY, USA: ACM, 2012, pp. 341–352. [Online]. Available: http://doi.acm.org/10.1145/2304576.2304623

[11] T. Stefanski, "Implementation of fdtd-compatible greens function on heterogeneous cpu-gpu parallel processing system," *Progress In Electromagnetics Research*, vol. 135, p. 297316, 2013.

[12] H. Berendsen, D. van der Spoel, and R. van Drunen, "Gromacs: A message-passing parallel molecular dynamics implementation," *Computer Physics Communications*, vol. 91, no. 1, pp. 43 – 56, 1995. [Online]. Available: http://www.sciencedirect.com/science/article/pii/001046559500042E

[13] J. R. d. S. Junior, E. W. Clua, A. Montenegro, and P. A. Pagliosa, "Fluid simulation with two-way interaction rigid body using a heterogeneous gpu and cpu environment," in *2010 Brazilian Symposium on Games and Digital Entertainment*, Nov 2010, pp. 156–164.

[14] S. Venkatasubramanian and R. W. Vuduc, "Tuned and wildly asynchronous stencil kernels for hybrid cpu/gpu systems," in *Proceedings of the 23rd International Conference on Supercomputing*, ser. ICS '09. New York, NY, USA: ACM, 2009, pp. 244–255. [Online]. Available: http://doi.acm.org/10.1145/1542275.1542312

[15] C. Yang, W. Xue, H. Fu, L. Gan, L. Li, Y. Xu, Y. Lu, J. Sun, G. Yang, and W. Zheng, "A peta-scalable cpu-gpu algorithm for global atmospheric simulations," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13. New York, NY, USA: ACM, 2013, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/2442516.2442518

[16] C. Yang, F. Wang, Y. Du, J. Chen, J. Liu, H. Yi, and K. Lu, "Adaptive optimization for petascale heterogeneous CPU/GPU computing," in *Proceedings of the 2010 IEEE International Conference on Cluster Computing, Heraklion, Crete, Greece, 20-24 September, 2010*, 2010, pp. 19–28. [Online]. Available: https://doi.org/10.1109/CLUSTER.2010.12

[17] M. Sourouri, J. Langguth, F. Spiga, S. B. Baden, and X. Cai, "Cpu+gpu programming of stencil computations for resource-efficient use of gpu clusters," in *2015 IEEE 18th International Conference on Computational Science and Engineering*, Oct 2015, pp. 17–26.

[18] A. D. Pereira, L. Ramos, and L. F. W. Ges, "Pskel: A stencil programming framework for cpu-gpu systems," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 17, pp. 4938–4953. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3479

[19] A. D. Pereira, R. C. O. Rocha, L. Ramos, M. Castro, and L. F. W. Ges, "Automatic partitioning of stencil computations on heterogeneous systems," in *2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, Oct 2017, pp. 43–48.

[20] I. Z. Reguly, G. R. Mudalige, and M. B. Giles, "Loop tiling in large-scale stencil codes at run-time with ops," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, pp. 873–886, 2018.

[21] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, "Effective automatic parallelization of stencil computations," *SIGPLAN Not.*, vol. 42, no. 6, pp. 235–244, Jun. 2007. [Online]. Available: http://doi.acm.org/10.1145/1273442.1250761

[22] S. McIntosh-Smith, M. Boulton, D. Curran, and J. Price, "On the performance portability of structured grid codes on many-core computer architectures," in *Supercomputing*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2014, vol. 8488.

[23] S. Pennycook, J. Sewall, and V. Lee, "Implications of a metric for performance portability," *Future Generation Computer Systems*, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X17300559

APPENDIX

A. Abstract

This artifact comprises the source code, datasets, and build instructions on GitHub that can be used to reproduce our results presented in our paper.

B. Description

1) Check-list (artifact meta information):

- **Algorithm: Heterogeneous scheduling for stencil computations**
- **Program: C/C++ code in OPS backend and CUDA/OpenMP code generated by OPS.**
- **Compilation: Intel ICC/ICPC 17.0.0. and IBM XLC 13.1.7**
- **Transformations: OPS uses a source-to-source translator (see documentation), however all files required have already been generated.**
- **Binary: hybrid executables built with `make` (`*_hyb_cuda`)**
- **Data set: for CloverLeaf, the included clover.in files in the application directories. Can be edited to change problem size.**
- **Run-time environment: CentOS 7.2 with Intel Parallel Studio XE 2017 (Intel) and Ubuntu 16.04.3 with XLC 13.1.7**
- **Hardware: tested on Xeon E5-2650 v3 @ 2.3 GHz, with Hyper-Threading enabled and Power8 (10 core) @ 3.69 GHz**
- **Execution: `OMP_NUM_THREADS=20 numactl --cpunodebind=0 ./executable_name OPS_HYBRID OPS_TILING`.**
- **Output: Elapsed times, GB/s and GFLOPS/s breakdowns.**
- **Experiment workflow: clone sources from GitHub, build OPS backend, build poisson/CloverLeaf 2D hybrid tiled executables, compare to plain CUDA results.**
- **Experiment customization: Plain hybrid execution can be enabled with the `OPS_HYBRID` flag, tiled version can be enabled by adding the `OPS_TILING` flag, and tile sizes to improve CPU-side cache blocking tiling may be set using the T1,T2 environment variables, the problem size in clover.in. For the Jacobi problem, using the -sizex, -sizey, -iters, and the tile height using -itert, switching between copy and non-copy versions with -non-copy.**
- **Publicly available?: Yes**

2) How software can be obtained (if available): The OPS library can be downloaded from `https://github.com/OP-DSL/OPS`.

3) Hardware dependencies: Any Intel CPU, or any IBM Power8 CPU.

4) Software dependencies: The library can be compiled with several compilers (intel, gnu, pgi, clang, XL), but to reproduce the results, we suggest using Intel Parallel Studio XE 2017 on Intel, and XLC 13.1.7 on Power.

5) Datasets: Data is implicitly generated by the applications themselves. To specify the grid dimensions in the Jacobi benchmark, one has to add the -sizex and -sizey arguments, and -iters for the number of iterations. -itert can be used to set the tile height, and -non-copy to switch from the copy variant to the non-copy variant. For CloverLeaf 2D/3D, modify the clover.in files to specify the dimensions and the number of iterations/convergence criteria.

C. Installation

Clone the OPS repository and checkout the feature/hybrid branch:

```
git clone https://github.com/OP-DSL/OPS
cd OPS
git checkout feature/hybrid
```

Set up environment variables, as specified in README.md, to specify the compiler used and to point to the locations of the OPS library, MPI and HDF5, then build the OPS C backend:

```
cd ops/c
make
```

Build the poisson and CloverLeaf variants

```
cd apps/c/poisson
make poisson_cuda poisson_hyb_cuda
cd apps/c/CloverLeaf
make cloverleaf_cuda cloverleaf_hyb_cuda
```

D. Experiment workflow

For testing the OPS version of the Jacobi problem with various parameters:

```
T1=tilesize_x T2=tilesize_y OMP_NUM_THREADS=20 \
  numactl --cpunodebind=0 ./poisson_hyb_cuda \
  -sizex=8192 -sizey=8192 -iters=1000 \
  -itert=20
```

For testing the CloverLeaf 2D with various parameters:

```
T1=tilesize_x T2=tilesize_y OMP_NUM_THREADS=20 \
  numactl --cpunodebind=0 ./cloverleaf_hyb_cuda \
  -DOPS_DIAGS=0 OPS_HYBRID OPS_TILING
```

E. Evaluation and expected result

All benchmarks print timing breakdowns as well as achieved bandwidth figures, which can be directly compared to those in figures and tables in the paper. CloverLeaf's GFLOPS/s results are extrapolated based on the CUDA version (build cloverleaf_cuda) and the GFLOPS/s values reported by `nvprof`.

F. Experiment customization

The Jacobi benchmark can be customised through runtime flags, and the CloverLeaf runs through changes to the clover.in input file, that are in the same directory as the executable.

G. Notes

You may have to modify the compilation flags to use `-fp-model fast -fma`, as the defaults may be set to IEEE compliant optimisation flags.