Loop Tiling in Large-Scale Stencil Codes at Run-time with OPS

István Z. Reguly, *Member, IEEE,* Gihan R. Mudalige and Michael B. Giles

Abstract—The key common bottleneck in most stencil codes is data movement, and prior research has shown that improving data locality through optimisations that optimise across loops do particularly well. However, in many large PDE applications it is not possible to apply such optimisations through compilers because there are many options, execution paths and data per grid point, many dependent on run-time parameters, and the code is distributed across different compilation units. In this paper, we adapt the data locality improving optimisation called tiling for use in large OPS applications both in shared-memory and distributed-memory systems, relying on run-time analysis and delayed execution. We evaluate our approach on a number of applications, observing speedups of 2× on the Cloverleaf 2D/3D proxy applications, which contain 83(2D)/141(3D) loops, 3.5× on the linear solver TeaLeaf, and 1.7× on the compressible Navier-Stokes solver OpenSBLI. We demonstrate strong and weak scalability on up to 4608 cores of CINECA's Marconi supercomputer. We also evaluate our algorithms on Intel's Knights Landing, demonstrating maintained throughput as the problem size grows beyond 16GB, and we do scaling studies up to 8704 cores. The approach is generally applicable to any stencil DSL that provides per loop nest data access information.

Index Terms—DSL, Tiling, Cache Blocking, Memory Locality, OPS, Stencil, Structured Mesh

1 Introduction

Modern architectures now include ever-larger on-chip caches to help exploit spatial and temporal locality in memory accesses: latency and energy benefits of accessing data from cache can be up to 10x compared to accessing it with a load from off-chip memory. Unfortunately, most scientific simulations are structured in a way that limits locality: the code is structured as a sequence of computations, each streaming a number of data arrays from memory, performing a number of operations on each data element, then streaming the resulting arrays back to memory.

Improving memory locality is an area of intense research, and stencil codes have long been a target, given their regular memory access patterns and (mostly) affine loop structures. In stencil codes, we iterate through a 1/2/3 (or higher) dimensional grid, and perform computations given data on the current and adjacent grid points - the adjacency pattern is called the stencil. In a single loop nest (one sweep over the domain) there is already potential for data reuse, given the stencils used (e.g. along the contiguous dimension), which can be further improved using loop blocking [1] - this is standard practice in modern compilers.

Loop fusion [2] merges multiple subsequent loop nests into a single loop nest, making data reuse possible on a larger scale - across loop nests. Fusion is easy to do when loop bounds of subsequent loop nests align, and data dependencies are trivial. There are many examples of loop fusion, demonstrating its importance [3]. Loop fusion in the presence of non-trivial stencils (loop nest reading data generated by a previous loop nest with a multi-point stencil) is much more difficult because loops have to be shifted depending on the stencil pattern, leading to wavefront schemes.

There is a large body of research on the combination of fusion and loop schedule optimisations [4], [5], [6]: techniques that extend loop blocking to work across subsequent loop nests, generally called *tiling*. Tiling carries out dependency analysis similar to what is required for loop fusion, but instead of fusing the bodies of subsequent loops, it forms small blocks in each loop nest (fitting in the cache). Tiling achieves memory locality by executing the same set of blocks in subsequent loops, formed to satisfy data dependencies, then moves on to another set of blocks, etc. There is a well-established framework for loop scheduling transformations: the polyhedral framework.

Research into polyhedral compilers has laid a strong theoretical and practical foundation for cache blocking tiling, yet their use is limited by the fact that they apply compile-time optimisations. These compilers struggle with dynamic execution paths, where it is not known in what exact order loops follow one another, and they cannot manage analysis across multiple compilation units. Furthermore, many cannot handle branching that would lead to different access patterns within a single loop nest. Commonly used benchmarks such as SPEC OMP [7] and PolyBench [8] do not include such use cases. In summary, these compilers have primarily been shown to give excellent performance when a small number of loops repeat a large number of times in a predictable manner - which we do *not* consider large-scale codes for the purposes of this paper.

The OPS (Oxford Parallel library for Structured meshes) DSL (Domain Specific Language) [9], [10] is a C/C++/Fortran domain-specific API that uses source-to-source translation and various back-end libraries to automatically parallelise applications. Any code written using its API can utilise MPI, use multi-core CPUs with OpenMP, as well as GPUs with CUDA, OpenACC, or OpenCL. OPS is being used in a number of PDE applications [10], [11], [12], and indeed the common bottleneck in all of these ap-

- *I.Z. Reguly is with PPCU ITK, Budapest, Hungary. Email: reguly.istvan@itk.ppke.hu*
- *G.R. Mudalige is with Department of Computer Science, University of Warwick, UK. Email: g.mudalige@warwick.ac.uk*
- *M.B. Giles is with the Maths Institute, University of Oxford, UK. Email: mike.giles@maths.ox.ac.uk*

Manuscript received XXX

plications is data movement. The aforementioned challenges combined with the complexity of these applications prohibit the use of traditional stencil compilers - therefore we adopt an iteration space tiling algorithm in OPS, that we apply at run-time using delayed execution of computations.

We choose the CloverLeaf 2D/3D code (part of the Mantevo suite) to demonstrate our results in detail, as it is a larger code that has been intensively studied by various research groups [10], [13], [14]; it is a proxy code for industrial hydrodynamics codes. It has 30 datasets (30 data values, or variables, per grid point), it consists of 83(2D)/141(3D) different loops across 15 source files. During the simulation, a single time iteration consists of the execution of 150/600 loops, where often the same loop nest is executed on different datasets. Furthermore, some stencils are data-dependent, and there is considerable logic that determines the exact sequence of loops. To demonstrate the generality of our approach, we evaluate performance on two more applications using OPS: the matrix-free sparse linear solver proxy code TeaLeaf [15] (also part of the Mantevo suite), and the compressible Navier-Stokes solver OpenSBLI [16].

In this paper, we present research into how, through a combination of delayed execution and dependency analysis, OPS is capable of addressing the aforementioned challenges, *without modifications to the high-level OPS user code*. This is then evaluated through a series of benchmarks and analysed in detail. Specifically, we make the following contributions:

1) We introduce the delayed execution scheme in OPS and describe the dependency analysis algorithm that enables tiled execution on a single block.
2) We extend our algorithms to analyse dependencies and perform scheduling and communications in a distributed memory environment
3) We validate and evaluate the proposed algorithm on a 2D Jacobi iteration example, comparing it to prior research (Pluto and Pochoir).
4) We deploy the tiling algorithm on a number of larger-scale applications, such as the CloverLeaf 2D/3D hydrocode, TeaLeaf, and OpenSBLI. We explore relative and absolute performance metrics, including speedup, achieved bandwidth and computational throughput on Xeon server processors, scaling up to 4608 cores on CINECA's Marconi.
5) We evaluate tiling on the Intel Knights Landing platform, scaling up to 128 nodes (or 8704 cores).

The rest of the paper is organised as follows: Section 2 discusses related work, Section 3 summarises the design and implementation of OPS, Section 4 presents the tiling algorithm integrated into OPS, Section 5 introduces the applications we evaluate in this work and Section 6 carries out the in-depth performance analysis. Section 7 evaluates strong and weak scaling on a CPU cluster, and finally Section 9 draws conclusions.

2 Related work

Manipulating loop schedules to improve parallelism or data locality has long been studied and built into compilers [1], [17], [18], [19], [20]. The mathematics and techniques involved in such loop transformations have been described in the polyhedral framework [21], [22], [23], and since then, a tremendous amount of research has studied transformations of affine loop structures in this framework, and extended it to work on many non-affine cases as well.

Tiling by manually modifying code has been demonstrated on smaller codes [24], [25] where one or two loops repeat a large number of times (typically a time iteration); it is a particularly good example of utilising the large caches on CPUs, and they have been studied in detail.

There are a number of compilers focused on applying tiling to stencil computations such as Pochoir [26], image processing workflows such as Polymage and Halide [27], [28], and more generally to computations covered by the polyhedral model: Pluto [29], [30], R-STREAM [31] - these have shown significant improvements in performance by exploiting data locality by manipulating loop schedules. There are examples of tiling in distributed memory systems as well: R-STREAM [31], Pluto [32], Classen and Griebl [33], and Distributed Halide [34]. In comparison to our work, polyhedral frameworks primarily target use cases where a few loops repeat a large number of times (e.g. in a time iteration), accessing just a few datasets. PolyMage and Halide on the other hand do handle long multi-stage image processing flows, though typically access only a few datasets - since they target a different application domain, they are not applicable to the kinds of codes studied here, but some of the techniques used are shared with our work.

The kinds of transformations applied are also wide-ranging, starting at the simplest skewed tiling methods across time iterations [17], [25], wavefront methods [17], [35], and their combinations with various tile shapes such as diamond and hexagonal tiling [36], [37]. We use a skewed tiling scheme in a slightly different way: while skewing in time means that iteration ranges of the same loop nest repeated in a time-loop are skewed, in our case the iteration ranges of subsequent (different) loop nests are skewed relative to each other to account for data dependencies.

The only work that we are aware of that has applied similar transformations to large-scale scientific problems is the Formura DSL [6], which is in full control of the code that is being generated from high-level mathematical expressions - therefore it avoids the issue of various execution paths and multiple compilation units to tile across.

A common point in all of the above research is that the transformations are applied at compile-time (or before), and therefore they are inherently limited by what is known at compile time, and the scope of the analysis. This in turn makes their application to large-scale codes distributed across many compilation units, that have configurable, complex execution flows and call stacks, exceedingly difficult.

Identifying the sequence of loops to tile across and to carry out dependency analysis is a lot easier at run-time, particularly with the help of delayed evaluation or lazy execution [38], [39], which is a well-known technique used particularly in functional languages that allows expressions to be evaluated only when their results are required. Lazy execution is also used in other fields, such as Apache Spark to plan out the sequence of computations and to skip unnecessary steps. We apply the lazy execution idea to figure out dependencies and compute loops schedules at runtime - to our knowledge these two have not been used together in scientific computing.

```
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

3 The OPS Embedded DSL

The Oxford Parallel library for Structured meshes (OPS) is a Domain Specific Language embedded into C/C++/Fortran, defining an API for expressing computations on multi-block structured meshes. It can be used to express algorithms at a higher level, without having to worry about the intricacies of parallel programming and data movement on various computer architectures. By separating the high-level code from the low-level implementation, OPS lets domain scientists write a single high-level source code and the library developers to automate the generation of low-level implementations given the knowledge of the domain and the target architectures.

OPS defines the following abstraction [9]: the computational domain consists of a number of N dimensional *blocks*, with a number of *datasets* defined on each. Then, computations are expressed as a sequence of parallel loops applying given "user-kernels" over given iteration ranges and a number of datasets defined on the same block, specifying how each dataset is accessed: whether it is read, written, or incremented and what exact stencil is used for the access - this follows the access-execute model [40]. The abstraction requires the parallel operation to be insensitive to the order of execution on individual grid points (within machine precision).

An example of an OPS parallel loop is shown in Figure 1; the `ops_par_loop` API call takes as arguments a function pointer to be applied to each grid point, a block, a dimensionality, an iteration range and a number of data arguments. A data argument encapsulates the dataset handle, the stencil, the underlying primitive datatype and the type of access.

Given this abstraction, OPS is free to parallelise both over parallel loops over different blocks, as well as over individual grid points within a single parallel loop: indeed the library assumes responsibility for correctly parallelising in distributed-memory as well as shared-memory environments, and on different architectures, using different parallel programming models. With a user code written once using the C/C++ or Fortran API of OPS, a source-to-source translator generates code for sequential, OpenMP, OpenACC, OpenCL and CUDA execution, which is then compiled with a traditional compiler and linked against one of the OPS back-end libraries that supports MPI parallelisation and data management. Because ownership of data is handed to the library, and access only happens through OPS APIs, the library can keep track of what data changed and when it is necessary to update it: halos for MPI or the separate address spaces of CPUs and GPUs.

OPS can dramatically improve productivity, particularly when a code is deployed to different architectures, and therefore it needs to support multiple parallelisations. There is a one-off cost of conversion to OPS - with a difficulty somewhere between applying a pragma-based parallelisation (OpenMP) and adopting CUDA: loop bodies have to be outlined and the parallel loop calls written. Once converted, the wide range of parallelisations and optimisations are applied automatically, and performance is no worse than one-off hand-written conversions, as demonstrated in our previous work [10].

4 Skewed tiling in OPS

As described in the previous section, at runtime the `ops_par_loop` construct includes all necessary information about a computational loop nest that is required to execute it: the computational kernel, the iteration range, and a list of datasets, plus how they are accessed - the stencil and whether read or written. This enables OPS to store this information for delayed execution, and reason about multiple loops: an instance of the loop chaining abstraction [41].

4.1 Delayed execution

With all pertinent information about a loop, we create a C struct at runtime, which includes a function pointer to a C++ function that, given the loop nest ranges and the argument list stored in the struct, can execute the computational loop. When the `ops_par_loop` is called from user code, this struct is passed to the back-end, and stored in an array for later execution. Parallel loops can be queued up until the point when the user code needs some data to be returned: such as getting the result of a reduction, based on which a control decision has to be made. At this point, OPS triggers the execution of all loops in the queue.

4.2 Dependency analysis and tile construction

Having queued up a number of computational loops, it is now possible to carry out dependency analysis: this enables us to reason about loop scheduling not only in individual loops but across a number of loops as well. The ultimate goal is to come up with execution schedules that improve data locality by way of cross-loop blocking. Therefore, the dependency analysis carried out by OPS takes into consideration the sequence of loops, the datasets accessed by each loop, the stencils used and whether the data is read, written, or both. Given the restrictions of the OPS abstraction (only trivially parallel loops permitted), the runtime information about datasets and stencils used to access them, the dependency analysis is based on the well-known polyhedral model (though not any specific framework).

The theory of transformations to polyhedral models is well documented [42], [43]. Here we focus only on the overall algorithmic description and some practicalities. Unlike most classical tiling algorithms, we do not assume nor

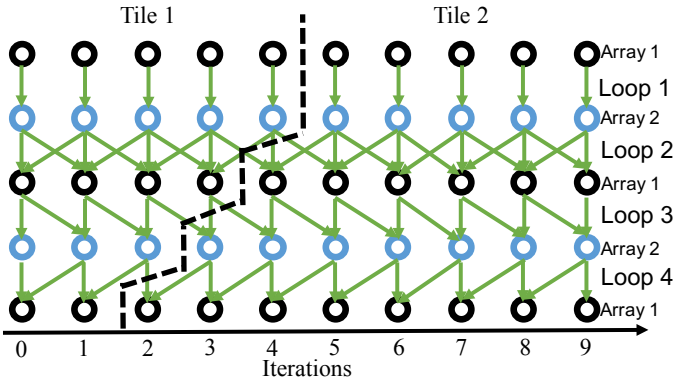

Fig. 2. Illustrative example of 1D dependency analysis and tiling

exploit any recurrence, the time dimension can be thought of as being replaced by a sequence of loops that may all have different iteration ranges and stencils. In this paper, we use dependency analysis to implement a skewed tiling scheme, where subsequent loop nests are skewed (unlike in case of skewing in time), with a sequential dependency (and scheduling) between subsequent tiles and intra-tile parallelism.

To understand our optimisation approach, consider a simple 1D example with four loops, with access patterns and dependencies that are illustrated in Figure 2. This is the conventional way of formulating computational loops; the issue is that these datasets are too large to fit in the on-chip cache, they will be streamed to and from the CPU between loops. This implies that data reuse only happens within a single loop nest, and not across subsequent loop nests. Data reuse in this case is 1-3x depending on the stencil used (to have computations and data movement in balance on modern CPUs, one would need a factor of $\sim 40\times$).

In order to improve locality and reuse data across different loop nests, we need to take multiple loop nests and reason about them together; we would like to apply loop blocking within and across the four loops (tile across four loops). To achieve this, we need to block the iteration ranges of loops and re-organise them so that data accessed by a given block in the first loop nest stays in cache and gets accessed by blocks of subsequent loop nests. Then, we execute the next block on the first loop nest, then on the second loop nest, etc., keeping data in cache in-between. Thus, we can achieve cross-loop data reuse. The caveat is that in constructing these blocks, we need to make sure all data dependencies are satisfied; only those iterations can be included in the block for the subsequent loop nest, for which the block of the previous loops computed the required inputs. Additionally, we wish to parallelise within tiles - threads will execute different iterations of a block of a loop, synchronise, then move on to the block of the next loop.

One way to partition the index spaces of the loops, is to split loop 1 into two equal parts: $[0-4]$ for Tile 1 and $[5-9]$ for Tile 2. Execution in loop 2 uses a 3-point stencil - there is a data dependency on adjacent points computed by loop 1; a Read-after-Write (RAW) dependency. Thus in Tile 1, the iteration range of loop 2 has to be restricted to $[0-3]$, and in Tile 2, it is extended to $[4-9]$. In loop 3, Tile 1 has all the data dependencies satisfied to execute iterations $[0-3]$, however this would overwrite the value at index 3 in Array 2, which would lead to incorrect results when Tile 2 is executing iteration 4 or loop 2; a Write-after-Read (WAR) dependency. Therefore loop 3 in Tile 1 is restricted to $[0-2]$, and extended to $[3-9]$ in Tile 2. Finally, loop 4 is once again restricted to $[0-1]$ due to a RAW dependency in Tile 1, and extended to $[2-9]$ in Tile 2. We call this method of restricting and extending iteration ranges of subsequent loop nests skewing, and the resulting execution scheme skewed tiling.

This method can be generalised to work in arbitrary dimensions and with a much larger number of loops. The algorithm is given a set of loops $loop_l$, with iteration ranges in each dimension $loop_l.start_d$, $loop_l.end_d$, a number of arguments and stencils. Our goal is to construct a number of tiles, each containing iteration ranges for each loop nest $tile_{t_d}.loop_l.start_d, tile_{t_d}.loop_l.end_d$, in a way that allows for the re-organisation of execution in a tiled way, as shown in Algorithm 1.

The algorithm produces as its output the iteration ranges for each loop nest in each tile - this is then cached as a "tiling plan" and reused when the same sequence of loops is encountered. The structured nature of the problem and the fixed size of tiles does not necessitate computing and storing all this information for each and every tile and loop, since it could be computed on the fly - however in our applications the number of corner cases (boundary loops, sub-domain loops) made this approach to analysis and execution overtly complex, therefore we decided to do it on a per-tile basis. This means that the complexity of the analysis is higher: O(#loops*#tiles) - but as we show later, this cost is still negligible.

4.3 Tiled execution

Given a tiling plan, the execution of the tiled loop and iteration schedule is described by Algorithm 2: we iterate through every tile, and subsequent loops, replacing the original iteration range with the range specific to the current tile (loops with empty index sets are skipped), then start execution through the function pointer. Parallelisation happens within the tiles, leading to a synchronisation point after each loop, in each tile.

The construction of the tiles and the logic for their execution is entirely contained in the backend code of OPS, and it is independent of the application. The code generated for the application itself only facilitates the construction of loop descriptors and the execution of single loop nests with loop bounds that may have been altered by the tiling algorithm.

For this work, we chose the skewed tiling algorithm instead of more advanced algorithms such as diamond tiling [44] or its combination with wavefront scheduling [36], because it is both simpler to implement and verify, and it results in large tiles, helping to diminish the overhead of launching the execution of computational loops through function pointers (detailed in section 6.2). OPS however captures all the information necessary to apply more complex loop scheduling, which will be the target of future research.

4.4 Tile size selection

Considering we perform intra-tile parallelisation, we need tiles with a memory footprint that is about the size of the

Algorithm 1 CONSTRUCTION OF TILING PLAN

Input: $loop_l$ structures $l = 1..L$
Output: $tile_t$ structures
{ compute union of index sets}
for all l in loops, d in dimensions **do**
  $start_d = min(start_d, loop_l.start_d)$
  $end_d = max(end_d, loop_l.end_d)$
  $num\_tiles_d = (end_d - start_d - 1)/tilesize_d + 1$
end for
for all l in loops backward, d in dimensions **do**
  for all t in tiles **do**
    { start index for current loop, dimension and tile}
    if t first in d **then**
      $tile_{t_d}.loop_l.start_d = loop_l.start_d$
    else
      $tile_{t_d}.loop_l.start_d = tile_{t_d-1}.loop_l.end_d$
    end if
    { end index for current loop, dimension and tile}
    $tile_{t_d}.loop_l.end_d = -\infty$
    if t last non-empty in d **then**
      $tile_{t_d}.loop_l.end_d = loop_l.end_d$
    else
      { satisfy RAW dependencies }
      for all a in arguments of loop written **do**
        $tile_{t_d}.loop_l.end_d = min(loop_l.end_d,$
          $max(tile_{t_d}.loop_l.end_d, read\_dep_a.tile_t.end_d))$
      end for
    end if
  end for{ over tiles }
  for all t in tiles **do**
    if t not last **then**
      {satisfy WAR and WAW dependencies }
      for all a in arguments of loo **do**
        $m =$ largest negative stencil point in d
        $tile_{t_d}.loop_l.end_d = min(loop_l.end_d, max($
        $tile_{t_d}.loop_l.end_d, write\_dep_a.tile_t.end_d - m))$
      end for
    end if
    { default to end index at tile size }
    if $tile_{t_d}.loop_l.end_d$ still $-\infty$ **then**
      $tile_{t_d}.loop_l.end_d = min(loop_l.end_d,$
        $start_d + t_d * tilesize_d)$
    end if
    { update read dependencies }
    for all a in arguments of loop read **do**
      $p =$ largest positive stencil point in d
      $read\_dep_a.tile_t.end_d = max($
        $read\_dep_a.tile_t.end_d, tile_{t_d}.loop_l.end_d + p)$
      $p =$ largest negative stencil point in d
      $read\_dep_a.tile_t.start_d = min($
        $read\_dep_a.tile_t.start_d, tile_{t_d}.loop_l.start_d + p)$
    end for
    { update write dependencies }
    for all a in arguments of loop written **do**
      $write\_dep_a.tile_l.end_d =$
        $max(write\_dep_a.tile_t.end_d, tile_{t_d}.loop_l.end_d)$
      $write\_dep_a.tile_l.start_d =$
        $min(write\_dep_a.tile_t.start_d, tile_{t_d}.loop_l.start_d)$
    end for
  end for
end for

Algorithm 2 EXECUTION OF A TILING PLAN

for all tiles t=1..T **do**
  for all loops l=1..L **do**
    for all dimensions d=1..D **do**
      $bounds\_start_d = tile_{t_d}.loop_l.start_d$
      $bounds\_end_d = tile_{t_d}.loop_l.start_d$
    end for
    call $loop_l$ with bounds:
      $bounds\_start_d, bounds\_end_d$
  end for
end for

last level cache, which is shared by all the threads. Thus we designed an algorithm in OPS to automatically choose a tile size based on the number of datasets accessed and the size of the last-level cache (LLC): this is described in Algorithm 3, omitting the handling of corner cases. Based on experiences with tile sizes found by exhaustive search, the general principle is that the size in the contiguous direction (X) should always be large to allow efficient vectorisation, and that the sizes in the non-contiguous directions should allow sufficient parallelisation and a good load balance between threads. Thus in 2D the X size is 3 times the Y size, and in 3D sizes are so that each thread has at least 10 iterations. An exhaustive search of possible tile sizes on different applications is deferred to the Supplementary Material.

Algorithm 3 TILE SIZE SELECTION ALGORITHM

$owned\_size =$ #of grid points owned by process
for all d in datasets **do**
  if d accessed by any loop in the chain **then**
    $footprint+ = d.size$ (bytes)
  end if
end for
$bytes\_per\_point = footprint/owned\_size$
$points\_per\_tile = LLC\_size/bytes\_per\_point$
if 2 dimensions **then**
  $M = \sqrt{points\_per\_tile/(3 * num\_threads^2)}$
  $tilesize_0 = 3 * M * num\_threads$
  $tilesize_1 = 1 * M * num\_threads$
end if
if 3 dimensions **then**
  $tilesize_0 =$owned size in X direction
  while $points\_per\_tile/tilesize_0 < 10 * num\_threads$ **do**
    $tilesize_0/ = 2$
  end while
  $tilesize_1 = \sqrt{points\_per\_tile/tilesize_0}$
  $tilesize_2 = points\_per\_tile/(tilesize_0 * tilesize_1)$
end if

4.5 Extension to distributed memory systems

The parallel loop abstraction of OPS lets it deploy different kinds of parallelisation approaches, including support for distributed memory systems through the Message Passing Interface (MPI). Without any additional user code, OPS will automatically perform a domain decomposition, create halo

regions for all datasets, and given the stencil access patterns in `ops_par_loop` constructs, automatically keep the values of the halo up-to-date. The performance and scalability of OPS has been presented and analysed in previous work [10].

In a distributed memory system, the two key issues in deploying tiling are the construction and scheduling of tiles, and the communication of data required for executing the tiles. Our skewed tiling approach in shared memory systems introduces a sequential dependency across tiles, which prohibits parallelisation between tiles (there we only parallelise within tiles) - this is obviously not a viable approach over MPI. Instead, we apply an overlapped tiling approach [34], [45], where points in the iteration space along the boundaries of the domain decomposition that are required for the execution of tiles on the other side of the boundary are replicated there. Thus, these iterations will be computed redundantly on an MPI process that does not own them, requiring communicating the needed data on those points. This approach is largely the same as what is described in Distributed Halide [34], except that it is all done at run-time.

In terms of the construction of tiles this means that some will extend beyond the original boundaries of domain decomposition, but in terms of scheduling it also means that MPI processes can execute their tiles independently (and in parallel) of one another. In terms of communications, the halo regions are extended to accommodate data required for the computation of these iterations, and before executing the tiles, MPI processes exchange a wider halo of datasets that are read before they are written. During the execution of the tiles however there is no need for further communication, because for each tile all data required is in local memory. This is in contrast to the existing MPI communications scheme in OPS, which is on-demand: halos are updated immediately before loops where they will be accessed, and there are no redundant computations.

There are two key changes to the construction of tiles and the execution schedule, as illustrated on Figure 3. First, the calculation of $tile_{t_d}.loop_l.start_d$ and $tile_{t_d}.loop_l.end_d$ has to account for domain decomposition boundaries: to compute the first tile's start and the last tile's end index, we need to extend beyond the boundaries, but we only have to look at read dependencies (no need to look at write dependencies in overlapped tiling). Second, we have to compute the required halo depth to communicate.

Before either of these, a minor modification to compute the number of tiles on each process: lines 5-6 of the original algorithm change to calculate with the bounds given by the domain decomposition as shown in Algorithm 4.

Algorithm 4 PER-PROCESS GRID SIZE

{line 5-6}
$start_d = min(start_d, loop_l.start\_thisprocess_d)$
$end_d = min(end_d, loop_l.end\_thisprocess_d)$

4.5.1 Tile shapes at MPI boundaries

When calculating the starting index for the first tile on the process, we have two separate cases: (1) when the process has no "left" neighbour in the current dimension (Figure 3, $tile_0$ of Process 0) , then the tile's start index is just the start

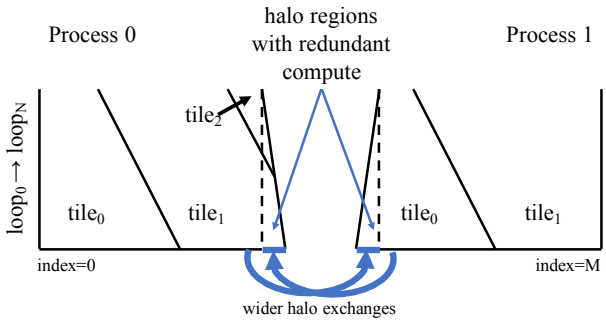

Fig. 3. A 1D illustrative example of tiling across MPI partitions

index of the original loop bounds as previously (line 13, Algorithm 1). (2) when the process has a "left" neighbour: then the iteration range has to be further extended to the left to account for read dependencies (Figure 3, $tile_0$ of Process 1). This is described by Algorithm 5, which replaces line 13 of the original.

Algorithm 5 TILE START INDEX OVER MPI

{line 13}
if process is first in d **then**
  $tile_{t_d}.loop_l.start_d = loop_l.start_d$
else
  { satisfy read-after-write dependencies }
  for all a in arguments of loop written **do**
    $tile_{t_d}.loop_l.start_d = max(loop_l.start_d,$
      $min(tile_{t_d}.loop_l.start_d, read\_dep_a.tile_t.start_d))$
  end for
end if

When calculating the end index of the last tile, we only account for read-after-write dependencies, which now will extend beyond the MPI partition boundary, due to the logic in line 24 (Figure 3, $tile_1$ and $tile_2$ of Process 0). Consider, that the slope of the two sides of the last tile is asymmetric: the left edge (which is the right edge of the previous tile) has to consider both read and write dependencies, but the right edge only considers read dependencies. In particularly long but slim tiles this may lead to the left edge reaching the right edge, and the tile having no iteration space in early loops, as illustrated by $tile_2$ on Figure 3. In accounting for this possibility, we have to check for this "overshoot" and if it does happen, then the previous tile's end index will have to be adjusted. In our algorithm, we have to include two checks, thus lines 30-36 are changed as shown in Algorithm 6.

4.5.2 Halo exchanges

Determining which datasets require a halo exchange and what depth is very simple given the correctly populated $read\_dep_a.tile_t.start_d, read\_dep_a.tile_t.end_d$ arrays; for each dataset we look for the first loop nest that accesses it, if it is a write, then no halo exchange is necessary, if it is a read, then the depth to exchange is the difference between the domain decomposition boundary and the read dependency index of the first/last tile, as illustrated on Figure 3. Since exchange depth may not be symmetrical, each process keeps track of the read dependencies of its previous neighbour's last tile and its next neighbour's first tile.

Algorithm 6 TILE END INDEX OVER MPI

{line 30-36}
if t not last and $tile_{t_d}.loop_l.end_d > tile_{t_d}.loop_l.start_d$ **then**
  { satisfy write-after-read/write dependencies }
  for all a in arguments of loop **do**
    m = largest negative stencil point in d
    $tile_{t_d}.loop_l.end_d = min(loop_l.end_d, max($
    $tile_{t_d}.loop_l.end_d, write\_dep_a.tile_t.end_d - m))$
  end for
  { if we overshoot the next tile }
  if $tile_{t_d}.loop_l.end_d > tile_{t_d+1}.loop_l.end_d$ **then**
    $tile_{t_d}.loop_l.end_d = tile_{t_d+1}.loop_l.end_d$
  end if
end if

This information is saved alongside the tiling plan, and each time we encounter the same sequence of loops, we first carry out these halo exchanges, then execute the tiling plan itself, which then does not need any further MPI communications until its completion.

5 STENCIL APPLICATIONS

In order to evaluate the efficiency of cache-blocking tiling in OPS, we first carry out an in-depth study using one of the most commonly used benchmarks in tiling research: a Jacobi iteration solving the heat equation. This benchmark problem is implemented in both Pluto and Pochoir, and comes as part of the distributed source package, and we use it as a basis for comparison. For both, a double-buffer approach is used where the stencil is applied to a first, putting the result in b, then the other way around, which is then repeated for the desired number of iterations.

Our main benchmark application is CloverLeaf [13]; to our knowledge, successful cache-blocking tiling has not been previously reported on an application of this size. CloverLeaf is a mini-application that solves the compressible Euler equations on a 2D or 3D Cartesian grid, using an explicit second-order method. It uses a Lagrangian-Eulerian scheme to solve Euler's equations for the conservation of mass, energy and momentum, supplemented by an ideal gas equation. CloverLeaf uses an explicit time-marching scheme, computing energy, density, pressure and velocity on a staggered grid, using the finite volume discretisation. One timestep involves two main computational stages: a Lagrangian step with a predictor-corrector scheme, advancing time, and an advection step - with separate sweeps in the horizontal/vertical/depth dimensions. The full source of the original is available at [46].

CloverLeaf was an ideal candidate for porting to use the OPS library - indeed it is the first application that was developed for OPS [10]. The 2D/3D application consists of 25/30 datasets defined on the full computational domain (200/240 bytes per grid point), and 30/46 different stencils used to access them. There are a total of 83/141 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 153/603 subsequent loops. The full size of CloverLeaf is 4800/6000 lines of code.

These properties of CloverLeaf make it virtually impossible to apply stencil compilers as they are limited by what is known at compile-time - which is indeed very little for larger-scale codes. While some portions of the code (blocks of 4-5 consecutive loops) are amenable to compile-time tiling approaches, there is little data reuse for tiling to show any performance benefit (as experiments with Pluto showed). This motivates our research into tiling with OPS that is capable of constructing and executing tiles at run-time.

To show that our results generalise to other applications using OPS, we also briefly evaluate performance on two more applications. The first is TeaLeaf 2D [15], also part of the Mantevo suite, which is a matrix-free sparse linear solver code for hydrodynamics applications. TeaLeaf has 98 nested loops over the 2D grid, 31 datasets defined on the grid, and the code is spread across 12 source files. It has a wide range of configuration parameters that control its execution: it supports various algorithms including Conjugate Gradient (CG), Chebyshev, and Preconditioned Polynomial CG (PPCG). At runtime, depending on the level of convergence and various problem-specific parameters, it will perform different numbers of preconditioning iterations, has early exits and other control structures that make it particularly unsuitable for polyhedral compilers.

The fourth key OPS application is OpenSBLI [16], a large-scale academic research code being developed at the University of Southampton, focusing on the solution of the compressible Navier-Stokes equations with application to shock-boundary layer interactions (SBLI). Here we are evaluating a 3D Taylor-Green vortex testcase, which consists of 27 nested loops over the computational grid, using 9 different stencils and accessing 29 datasets defined on the 3D grid.

6 BENCHMARKING AND PERFORMANCE ANALYSIS

6.1 Experimental set-up

We evaluate all algorithms and codes on a dual-socket Intel Xeon E5-2650 v3 (Haswell) machine, that has 10 physical cores per socket and 20 MB of L3 cache per socket. Hyper-Threading is enabled. For all tests, we run on a single CPU socket using `numactl` in order to avoid any NUMA effects, and parallelise with OpenMP within tiles (20 threads, pinned to cores). The latest version of OPS is available at [47]. We use Pluto 0.11.4 (dated Oct 28, 2015) and a Pochoir version dated Apr 15, 2015, available from GitHub. All codes are compiled with the Intel compilers version 17.0.3, with `-fp-model fast` and fused multiply-adds enabled.

For a simple roofline model, we benchmark a single socket of the system. Achieved bandwidth to DDR4 memory is 49 GB/s using the STREAM benchmark (Triad, 50M array, repeated 100 times), and 227 GB/s bandwidth to L3 cache (Triad, 900K array, repeated 1000 times). The double precision general matrix-matrix multiply test, using MKL, shows an achieved peak computational throughput of 270

GFLOPS/s. We use these figures for later analysis: with the balance point between computations and communication at 44 Flop/DWord - below the performance is bound by bandwidth, above it is bound by compute throughput.

Bandwidth figures shown in the following are based on back of the envelope calculations, ignoring data reuse within a single loop nest due to multi-point stencils; these values stay in cache. In the case of CloverLeaf we use the automated reporting system in OPS that estimates bandwidth based on the iteration range and the type of access (R/W) to data - for this calculation the data reuse due to multi-point stencils is ignored. As the Haswell microarchitecture does not have counters for floating point operations, we use counters in NVIDIA GPUs - we run OPS CUDA variants of our applications (the computational kernels are identical to the CPU implementation) through the `nvprof` profiler collecting double-precision flop counters using `--metrics flop_count_dp`. These flops counts are then used as they are to estimate GFLOPS/s throughput figures on the CPU.

6.2 Heat equation

As one of the most studied examples for tiling, we carry out the analysis on a Jacobi iteration solving the 2D heat equation. We solve on a 8192^2 mesh, with one extra layer for a Dirichlet boundary condition on all sides, for 250 time iterations. All data and computations are in double precision, and the total memory footprint is 1 GB.

6.2.1 State of the Art

First, we evaluate Pluto, compiling with the recommended flags: `./polycc test/jacobi-2d-imper.c --tile --parallel --partlbtile --pet`, and running a series of tests at different tile sizes. Without tiling (with OpenMP), the runtime is 8.42 seconds, achieving 29.69 GB/s and 14.02 GFLOPS/s. Pluto constructs a number of diamond tiles in sets that are inter-dependent, and parallelises over different tiles in the same set. The best performance is achieved at an X tile size of 160, a Y tile size of 32, tiled over 32 time iterations. The test completes in 1.98 seconds, achieving 126.23 GB/s. In terms of computational throughput, this corresponds to 59.64 GFLOPS/s.

Second, we evaluate Pochoir - as previously described, its implementation avoids straight copies from one array to another, therefore it is slightly faster. Without tiling, the reference implementation runs in 9.4 seconds, achieving 26.6 GB/s and 12.56 GFLOPS/s. Pochoir, similar to Pluto, parallelises over different tiles, the `heat_2D_NP` version runs in 3.26 seconds and achieves 76.7 GB/s and 36 GFLOPS/s, and the zero-padded version `heat_2D_NP_zero` runs in 2.92 seconds, and achieves 85.6 GB/s and 40.5 GFLOPS/s.

Third, we implement small experimental codes that solve the heat equation. In a similar way to Pluto, the tile sizes are known at compile time, however, (unlike Pluto) the mesh size and the number of iterations are not. The key difference between this code and Pluto/Pochoir, is that it uses the same sort of skewed tiles that OPS does, and parallelises within the tile. For this benchmark, we only tile in time and the Y dimension, and not in the X dimension (which is equivalent to choosing an X tile size of 8192). The baseline performance without tiling is 8.31 seconds, or 30.1 GB/s and 14.21 GFLOPS/s. With tiling, the best performance is achieved at a Y tile size of 120, tiling over 50 time iterations 2.43 seconds or 101 GB/s and 48 GFLOPS/s - memory used is 15MB (vs. 20MB of L3) per tile.

TABLE 1
Performance summary of the Jacobi iterations on 10 cores: timing is seconds (Base and Tiled), computational throughput in GFLOPS/s, and achieved bandwidth in GB/s

Test	Base	BW	Comp	Tiled	BW	Comp
Pluto	8.42	14	29.69	1.98	126.2	59.6
Pochoir	9.4	12.6	26.6	2.92	76.7	40.5
handcoded	8.31	14.2	30.1	2.43	101	48
OPS	8.58	13.6	29.4	3.69	67.3	32

Next, we outline the computational loop in our hand-coded benchmark into a separate source file (accepting data pointers and the X,Y iteration ranges as arguments), so as to simulate the way OPS calls the computational code. We compile with inter-procedural optimisations turned off to make sure the function does not get inlined - this reduces performance by 30-45% across the board, bringing the best performance from 2.43 to 3.54 seconds, or 70.1 GB/s and 33.35 GFLOPS/s.

We draw two key conclusions from these results: (1) on this example diamond tiling in Pluto performs best, hyperspace cuts in Pochoir also do well, and the simple skewed tiling approach is in-between. (2) this establishes an upper bound for performance that is achievable through OPS, where computational subroutines are outlined and are in separate compilation units, no compile-time tile size or alignment information is available.

6.2.2 OPS

Finally, we evaluate performance in OPS. Note, that OPS has the least amount knowledge of compile-time parameters, and it uses a completely generic dependency analysis at runtime, as opposed to all previous tests which do the dependency analysis at compile-time. Without tiling, the runtime is 8.58 seconds, achieving 29.4 GB/s and 13.6 GFLOPS/s.

Time iterations are simply expressed as a `for` loop calling `ops_par_loop`s repeatedly. Tile height (that is the number of time iterations to tile over) is defined through a runtime parameter; in the time iteration there is a statement that triggers the execution of the queued kernels after the given number of iterations. After switching on tiling and tuning the tile size, best performance is achieved at 8192 X size, 100 Y tile size, tiled over 30 time iterations, with a runtime of 3.69 seconds, achieving 67.3 GB/s and 31.99 GFLOPS/s, 5% slower than the *outlined* hand-coded benchmark. The overhead of computing the tiled execution scheme was 0.0040 seconds - about 0.1% of the total runtime.

It is important to observe that in all of the above benchmarks, the optimal performance was achieved when the X tile size was considerably bigger than the Y tile size - this is due to the higher efficiency of vectorised execution and prefetching: X loops are peeled by the compiler to get up to alignment with non-vectorised iterations, then the bulk of iterations are being vectorised over, and finally there are scalar remainder iterations.

6.2.3 Comparison of tiling implementations

Overall, as summarised in Table 1, the performance of non-tiled versions is quite consistent: they all run in about 8-

9 seconds, achieving close to 30 GB/s of bandwidth. This clearly shows that at this point performance is bound by how much data is moved. When cache-blocking tiling is enabled, performance improves dramatically, up to 4.5×, computations and latency of instructions starts to dominate performance. They achieve over 100 GB/s, which is a very good fraction of peak L3 bandwidth, considering misaligned accesses due to stencils and the cache flushes between tiles. In contrast to all other versions, OPS constructs tiles based on run-time information only, and calls computational kernels through function pointers which adversely affects performance as predicted by the outlined hand-coded benchmark, nevertheless it still achieves a 2.32× speedup over the non-tiled version.

6.3 Tiling CloverLeaf

The previous benchmark showed the superiority of stencil/polyhedral compilers, however they cannot be applied to an application like CloverLeaf at sufficient scope: code analysis and experiments showed that only a handful of code blocks with 3-4 consecutive loop nests can be tiled across, because either the sequence of loop nests cannot be determined at compile time or subsequent loop nests are in different compilation units. Tiling them with Pluto did not result in a measurable performance difference.

In contrast, OPS determines the loops to tile across at runtime. As it does not require any modifications to user code, it is possible to automatically deploy this optimisations to large-scale codes.

Enabling automatic tiling in OPS requires compiling a different executable, but otherwise no action is necessary on the part of the user. At runtime, unless specified otherwise, OPS will tile over all loops up to the point where a global reduction is reached, to make sure control decisions are executed correctly in the host code. In CloverLeaf, this means tiling over an entire time iteration, which is a sequence of 153 loops in 2D and 603 loops in 3D - this is also when best performance is achieved. Breaking the time iteration into several shorter chains is discussed in the Supplementary Material.

Aside from the automatic tile size selection algorithm, it is also possible to manually specify tile sizes in each dimension. Since OPS is parallelising within each tile, we size the tiles in the last dimensions to be an integer multiple of the number of threads - in 2D this is achieved by setting the Y tile size to be a multiple of 20, and in 3D we collapse the Y and Z loops (using OpenMP pragmas), and set Y and Z so that their product is a multiple of 20.

Here, we study the performance of both the 2D and the 3D versions of CloverLeaf. In 2D, we use a 6144^2 mesh, and run 10 time iterations, and in 3D, we use a 330^3 mesh, and run 10 time iterations. The total memory footprint in 2D is 7.054 GB and in 3D 8.741 GB. Note that normally CloverLeaf would run for over 10000 time iterations to fully resolve the simulation, but since its execution is following a recurring pattern, here we can restrict it to 10 and still obtain representative performance figures.

6.3.1 Baseline performance

The baseline performance using pure OpenMP restricted to a single socket is established at 18.7 seconds in 2D and 32.5 seconds in 3D. All innermost loops are reported as vectorised by the compiler. We show performance breakdowns in Tables 2 and 3 - note that here parallel loops are grouped by computational phase and averaged (weighted with relative execution time). It is clear that bandwidth is the key bottleneck in both 2D and 3D - especially in 2D where average bandwidth is 30 GB/s, same as on the Heat equation. There are a number of more computationally intensive kernels, where a considerable fraction of the peak computational throughput is achieved: for example Viscosity achieves 58/62 GFLOPS/s - but this is still not high enough to tip the balance from bandwidth-bound to compute-bound. Average bandwidth over the entire application in 3D is reduced to 25 GB/s, and the average computational throughput decreases from 20.7 to 18.9 GFLOPS/s. It is therefore clear that for most of these applications, the code is bound by bandwidth, which could potentially be improved with cache-blocking tiling.

6.3.2 Tiling CloverLeaf 2D

After enabling tiling in OPS, the best performance is achieved at a tile size of 640×160 - 8.73 seconds. At this point the memory footprint of the tile is 20.4 MB, and the speedup over the non-tiled version is 2.13×. Tiled performance is very resilient to changes in the exact tile size: in our experiments, there were 32 tile size combinations out of 144 within 2% of the optimum performance, with up to 60% smaller tiles, or up to 20% bigger tiles. For more details, please see the Supplementary Material. The tile size automatically chosen by OPS for the main time iteration is 600×200, with a runtime of 8.82 seconds, 1% slower that the tile size found by searching. The overhead of computing the tiling plans and loop schedules is 0.016 seconds, or 0.18% of the total runtime, which would be further diminished on longer runs.

Performance breakdowns for the 2D version are shown in Table 2 - in line with the overall reduction in runtime, both average bandwidth and computational throughput have more than doubled. It also shows that the least computationally intensive loops have improved the most; most of these are straightforward loops with few stencils, such as Revert and Reset which copy from one dataset to another: here we see a 3-4.5× improvement in runtime compared to non-tiled versions. The single most expensive phase is Momentum advection, with several fairly simple kernels, it gains a 3.4× speedup. In contrast, loops in Timestep or PdV gain only 1.1-1.5x due to reductions and higher computational intensity. Notably, boundary loops in Update Halo slow down by a factor of up to 3.7 - this is due to small loops being subdivided even further, worsening their overheads. Viscosity now achieves 143 GFLOPS/s, a considerable fraction of the measured peak. The second most expensive computational phase is cell advection, it gains only a 1.8× speedup, which is in part due to the large number of branches within the computational kernels.

6.3.3 Tiling CloverLeaf 3D

For CloverLeaf 3D, the best performance is achieved when the X dimension is not tiled, and the Y and Z tile sizes are both 20; runtime is 16.2 seconds, which is 2× faster than the baseline. Once again, we see the performance

TABLE 2
Peformance of CloverLeaf 2D baseline and tiled

	CloverLeaf 2D OpenMP				CloverLeaf 2D Tiled				
Phase	Time(sec)	%	GB/s	GFLOPS/s	Time(sec)	%	GB/s	GFLOPS/s	Speedup
Timestep	0.71	3.79	39.67	58.57	0.69	7.33	40.90	60.40	1.03
Ideal Gas	0.89	4.78	30.21	30.41	0.65	6.97	41.27	41.54	1.37
Viscosity	0.89	4.75	15.86	58.76	0.36	3.86	38.85	143.92	2.45
PdV	2.36	12.64	30.97	30.37	1.54	16.46	47.36	46.45	1.53
Revert	0.37	1.97	30.63	0.00	0.08	0.83	143.86	0.00	4.70
Acceleration	0.92	4.90	33.80	21.44	0.40	4.28	77.10	48.92	2.28
Fluxes	0.62	3.32	36.24	7.30	0.34	3.67	65.37	13.16	1.80
Cell Advection	4.01	21.46	30.16	18.92	2.23	23.73	54.34	34.09	1.80
Momentum Advection	6.68	35.77	32.84	12.89	1.95	20.77	112.66	44.21	3.43
Reset	0.74	3.95	30.46	0.00	0.25	2.62	91.51	0.00	3.00
Update Halo	0.07	0.39	2.66	0.00	0.26	2.80	0.73	0.00	0.28
Field Summary	0.11	0.57	47.82	37.44	0.05	0.56	96.42	75.50	2.02
The Rest	0.31	1.68	6.26	13.09	0.52	5.53	3.80	7.94	0.61
Total	18.68	100.00	30.89	20.71	8.73	100.0	66.12	44.31	2.14

TABLE 3
Performance of CloverLeaf 3D baseline and tiled

	CloverLeaf 3D OpenMP				CloverLeaf 3D Tiled				
Phase	Time(sec)	%	GB/s	GFLOPS/s	Time(sec)	%	GB/s	GFLOPS/s	Speedup
Timestep	1.77	5.27	18.14	31.64	0.85	5.05	38.02	66.34	2.10
Ideal Gas	0.89	2.64	28.99	29.18	0.64	3.80	40.36	40.63	1.39
Viscosity	1.71	5.10	14.06	62.39	0.63	3.74	38.54	171.01	2.74
PdV	3.88	11.54	21.41	24.84	2.47	14.77	33.56	38.94	1.57
Revert	0.37	1.09	29.22	0.00	0.12	0.72	88.73	0.00	3.04
Acceleration	2.05	6.11	18.42	19.43	1.08	6.47	34.92	36.83	1.90
Fluxes	1.13	3.36	28.58	9.59	0.48	2.84	67.70	22.72	2.37
Cell Advection	6.46	19.22	27.65	17.18	2.97	17.74	60.14	37.36	2.17
Momentum Advection	12.82	38.16	29.40	13.50	4.50	26.91	83.67	38.41	2.85
Reset	0.91	2.71	29.60	0.00	0.68	4.06	39.63	0.00	1.34
Update Halo	0.99	2.94	12.65	0.00	1.55	9.29	8.04	0.00	0.64
Field Summary	0.17	0.50	33.45	45.77	0.07	0.44	75.53	103.34	2.26
The Rest	0.45	1.34	5.56	13.92	0.52	3.12	4.79	11.99	0.86
Total	32.5	100.00	25.27	18.87	16.56	100.00	51.24	38.26	1.96

being resilient to the exact tile size, although the number of possible tile size combinations is significantly less than in 2D; out of 80 combinations tested, there were 6 other tile size combinations within 2% of the best performance, and 18 within 10%. The tile size automatically chosen by OPS is $330 \times 17 \times 17$, with a runtime of 17.02 seconds, or 5% slower that the tile size found by search. The overhead of computing the tiling plans and loop schedules is 0.01 seconds.

Performance breakdowns for 3D (Table 3) show a more consistent speedup over the baseline version compared to the 2D version: in 2D the standard deviation of speedups is 1.2, whereas in 3D it is only 0.75. The slowdown on Update halo is only 1.5×, and the best speedup (Revert) is only 3×. Both overall bandwidth and computational throughput have increased by a factor of two - Viscosity now achieves 171 GFLOPS/s, or 63% of peak, and the highest achieved bandwidth is 88 GB/s on Revert.

Enabling reporting from the dependency analysis in OPS shows the amount of skewing between the "bottom" and the "top" of the tiles; due to the large number of temporary datasets and the way loops are organised in directional sweeps, the total skewing is only 12 grid points in each direction in 2D and 14 grid points in the Y and Z directions in 3D (note that in this benchmark we do not tile in the X direction). In 2D, the additional data needed due to this skew is only a small fraction of the total tile size of 640×160 (0.2%). However, in 3D where the Y and Z tile sizes are 20, a skew of 14 points is 49%, meaning that much of the data is replaced between the execution of loops at the "bottom" of the tile and the "top" of the tile, nevertheless data reuse is still very high, the replacement happens gradually and it can be served fast enough from off-chip memory.

6.4 Tiling TeaLeaf and OpenSBLI

To demonstrate and underline the applicability of tiling in OPS to different applications as well, we deploy and evaluate this optimisations to two more applications, the TeaLeaf iterative solver code and the OpenSBLI compressible fluids solver. For TeaLeaf, we choose a 2000^2 mesh, and the PPCG solver, configured so that most time is spent in the preconditioning phase which does not require reductions (which limits tiling and is generally a bottleneck over MPI particularly), for the full list of configuration parameters, see [47] and `tea_tiling.in`. Keeping the benchmark still representative, we restrict the execution to 4 time iterations. The total memory footprint is 628 MB, however, the bulk of the runtime is spent executing the preconditioner, which

accesses only 5 datasets - a total of 160 MB. For OpenSBLI, we use a 257^3 grid and the RS variant [16], solving the Taylor-Green vortex testcase, and limiting the number of time iterations to 10; running to completion - 500 iterations - follows the same execution patterns, thus our 10 iterations are representative. The total memory footprint is 3.84 GB.

The results are presented in Table 4: runtime without tiling, with tiling and the automatic tile size selection algorithm and with using the best tile size found by exhaustive search (1010×400). The performance gain on TeaLeaf by enabling tiling is up to $3.5\times$, with a 5% difference between the automatic and manual tile size selection. The large speedup on TeaLeaf is largely due to the bulk of the runtime being spent in just two key loops doing preconditioning, without the need for reductions in-between. There are just 5 datasets being accessed in that part of the algorithm, and the computations themselves are fairly simple - only multiplications and additions. The achieved bandwidth for the baseline version is 46.27 GB/s (95% of peak), and the computational throughput is 13.3 GFLOPS/s. With tiling enabled, achieved bandwidth increases to 164.7 GB/s and computational throughput to 47.35 GFLOPS/s.

TeaLeaf accesses just a small number of all datasets during the bulk of its execution, thus tiling performance is very sensitive to the problem size: if we reduce it to 700^2 (20MB memory footprint), the speedup is reduced to just $1.07\times$. Performance is also very sensitive to the solver selection - if we use the Conjugate Gradient solver, without preconditioning, there is very little speedup at any problem size ($1.05\times$ at 4000^2), due to the frequent reductions, which prohibit tiling across more than 2 loops.

TeaLeaf, for most of its execution, repeats two key loops for a large number of times for this testcase (PPCG). Therefore, we manually modified the relevant code section in the non-tiled OPS version to enable tiling with Pluto - this involved multiversioning at a fairly high level in the call stack (something that we argue also makes the code more difficult to read and maintain). We then compiled and ran this modified version: with the default tile size selection algorithm in Pluto, performance actually decreased by $1.9\times$ to 25.66 seconds. After an exhaustive search over potential tiles sizes we have only been able to improve upon the non-tiled performance by $2.44\times$ to 5.49 seconds (tile size of 192×20 and 5 tile height).

Performance results from OpenSBLI are also presented in Table 4: here speedup from tiling is lower than on other applications: $1.7\times$. The performance difference between automatic and manual tile selection here is 9%. In this application, 60% of the runtime is spent in an extremely complicated computational loop nest containing arithmetic expressions that are tens of lines long (5 expressions in 151 lines), uses a large number of sqrt operations, and accesses 167 double precision values per grid point. As such it is not limited by either bandwidth (7.16 GB/s) or computational throughput (27.7 GFLOPS/s), rather by latency, register pressure, and other factors. Nevertheless, it still gains a $1.57\times$ speedup from tiling, achieving 11.24 GB/s and 43.49 GFLOPS/s. Overall, the entire application gains a $1.71\times$ improvement from tiling.

TABLE 4
Performance summary of TeaLeaf and OpenSBLI

Test	Baseline	Tiled	Tiled Best	Speedup
TeaLeaf	13.438s	3.9339s	3.7666s	3.56×
OpenSBLI	20.807s	13.221s	12.1385s	1.71×

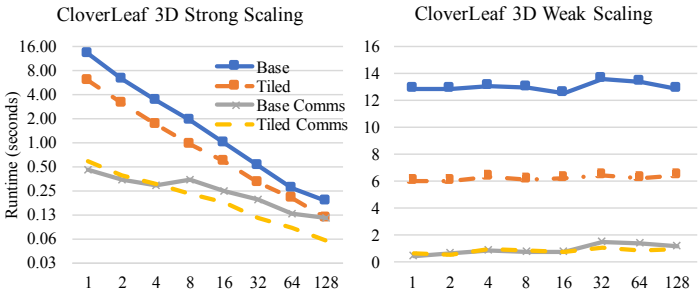

Fig. 4. Scaling CloverLeaf 3D to multiple nodes on Marconi

7 Tiling in Distributed-Memory Systems

To evaluate the efficiency of our tiling approach, we deploy the codebase on CINECA's Marconi supercomputer, and benchmark both strong scaling and weak scaling of the four large-scale codes. Marconi's A1 phase consists of 1512 nodes, each with dual-socket 18-core Broadwell Xeon E5-2697 v4 CPUs, running at 2.3 GHz (Hyper-Threading is off). The nodes are interconnected with Intel's 100Gb/s OmniPath fabric. The scheduling system currently limits job sizes to 160 nodes, therefore in our power-of-two scaling studies, we evaluate performance on up to 128 nodes, or 4608 cores.

For strong scaling CloverLeaf, we use the same mesh sizes as on the single socket tests. For strong scaling TeaLeaf and OpenSBLI we take the same problem sizes as described previously, and double their size in the x direction (due to the particularly small problem sizes in TeaLeaf). To evaluate scalability, we then keep this problem size and run it on an increasing number of nodes. Thus for CloverLeaf 2D, we strong scale a 6144^2 problem for 10 time iterations, for CloverLeaf 3D, a 330^3 problem for 10 time iterations, for TeaLeaf, a 4000×2000 problem for 2 solver iterations, and for OpenSBLI, a $514 \times 257 \times 257$ problem, for 10 time iterations.

For weak scaling, we take the problem sizes described for strong scaling, then scale it with the number of nodes, keeping the per-node size constant. For TeaLeaf, due to its convergence-dependent control flow that changes as we increase the problem size, we also alter convergence criteria to keep the number of solver iterations and the preconditioning iterations approximately the same - as it is not feasible to control this so that the number of iterations matches exactly, we report performance as time per 100 preconditioning iterations.

Figures 4-5 show the results of these scaling tests for CloverLeaf 3D and TeaLeaf - CloverLeaf 2D and OpenSBLI look very similar to CloverLeaf 3D, they are deferred to the Supplementary Material. For CloverLeaf 3D, we show total runtime as well as the time spent in MPI communications. For TeaLeaf, we show the time spent computing and the time spent in MPI communications separately (total time not shown), due to the much larger relative cost of

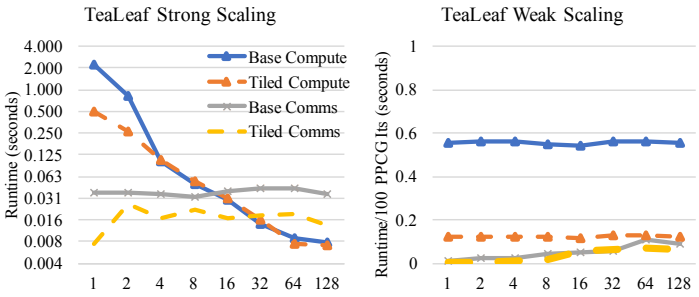

Fig. 5. Scaling TeaLeaf to multiple nodes on Marconi

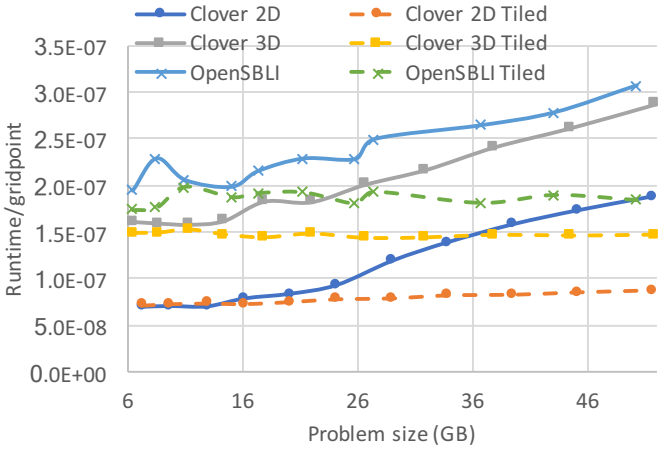

Fig. 6. Problem size scaling on KNL

communications. Tiling over MPI scales well, keeping the speedup ratio between tiled and non-tiled versions at higher node counts as well, except for the strong scaled TeaLeaf test, where after 4 nodes the two computational times are nearly identical: a quick calculation shows that at 4 nodes (8 sockets) the preconditioner's memory footprint is 40MB (versus the 35 MB L3 cache size), therefore even without tiling, the datasets stay in cache. Time spent in MPI communications, particularly when strong scaling, shows the advantage of the communications scheme used when tiling, which effectively results in fewer but larger messages.

8 Tiling on Intel Knights Landing

The second-generation Intel Xeon Phi platform, also called Knights Landing (KNL) has a 16GB on-chip stacked memory, with a $4-5\times$ higher bandwidth than that of DDR4. This memory can serve either as a cache to off-chip DDR4 memory, or as a separately managed memory space, or a combination of the two with a pre-defined split. This high bandwidth stacked memory is a great benefit to applications bound by memory bandwidth, such as our stencil codes. However, if the problem size has a larger memory footprint than 16GB, the user either has to manually allocate different datasets to different memory spaces, or has to rely on good enough caching behaviour. Here, we study the latter case and demonstrate how our tiling approach can maintain high performance and cache efficiency even when the full problem size is much larger than 16GB. We did not experience any benefit from trying to size tiles so they stay in L2 cache, therefore we do not report on those experiments.

Figure 6 shows the performance of the tiled and non-tiled implementations of CloverLeaf 2D/3D and OpenSBLI when the problem size increases; here we normalised

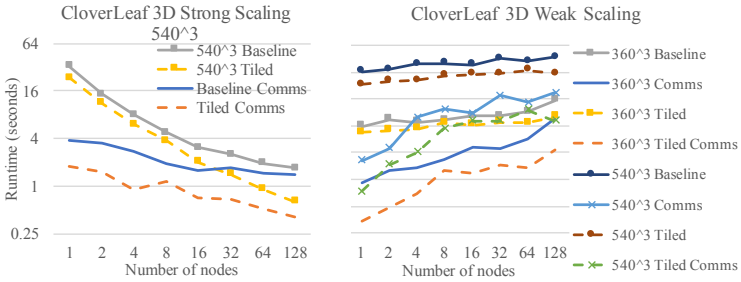

Fig. 7. Scaling CloverLeaf 3D on Marconi-A2 (KNL)

runtimes by dividing them with the number of gridpoints. On our x200 7210 chip (64 cores), we run with 4 MPI processes and 64 OpenMP threads each, and a cache/quadrant memory configuration. As the figure shows, up to a size of 16GB, all data fits in cache, and there is no or very little benefit (due to improved communications) from tiling, and the runtime per gridpoint remains constant. Beyond 16GB however, the non-tiled versions gradually slow down as less and less of the data being worked on stays resident in cache. Tiled versions on the other hand maintain their runtime per gridpoint, demonstrating the utility of our tiling approach even on a machine with orders of magnitude larger cache. At the larger problem sizes where the memory footprint is approximately 50GB, there is a $2.17\times$ speedup on CloverLeaf 2D, a $1.95\times$ speedup on CloverLeaf 3D, and a $1.67\times$ speedup on OpenSBLI. TeaLeaf was omitted due to its small memory footprint at reasonable problem sizes.

We also evaluate scaling up to 128 nodes on Marconi-A2, which has Intel Xeon Phi x200 7250 nodes (68 cores each). We weak scale two cases; one where all data fits in the 16GB cache and one where it does not. Here we report on CloverLeaf 3D, where we scaled a 360^3 (11.1 GB) and a 540^3 (37.8 GB) mesh - results are shown in Figure 7. For both strong scaling and weak scaling the smaller mesh, the baseline and the tiled versions perform very similar since all data can stay resident in cache. The tiled version performs slightly better due to its improved communications scheme; the cost of MPI communications is on average $2\times$ less. When weak scaling the larger mesh though, we can consistently see a 25% improvement over the baseline version. Scaling on CloverLeaf 2D and OpenSBLI show the same behaviour, their performance figures can be found in the Supplementary Material.

9 Conclusions

In this paper, we explored the challenges in achieving cache-blocking tiled execution on large PDE codes implemented using the OPS library; codes that are significantly larger than the traditional benchmarks studied in the literature: execution spans several compilation units, and the order of loops cannot be determined at compilation time. The key issues included how to handle dynamic execution paths within and across loop nests, and across a number of source files - something state-of-the-art polyhedral and stencil compilers (Pluto, Pochoir) cannot do. To tackle this, we adopt the locality improving optimisations called tiling, and instead of trying to tile at compile-time, we develop a run-time capability that relies on building a chain of loops through a delayed execution scheme.

To study the proposed approach, we established a baseline with a comparative study of the finite-difference heat equation Jacobi solver, comparing against Pluto and Pochoir. These libraries use different tiling and parallelisation strategies, and rely on compile-time analysis of the stencil code, but performance is closer than expected - most of the overhead is due to going through function pointers at run-time. Overall, the OPS version achieves a $3.1\times$ speedup over the non-tiled version and a bandwidth of 97.8 GB/s.

Thanks to the run-time analysis, the proposed approach can be trivially applied to larger-scale applications as well; we study the 2D and 3D versions of the CloverLeaf application in detail. Establishing a baseline shows that both versions are bound by bandwidth to off-chip memory. Enabling tiling shows a speedup of up to $2.1\times$ in 2D and $2\times$ in 3D. Detailed performance analysis shows that the simplest loops gain the most performance improvement, some computationally intensive loops become limited by compute instead of bandwidth, and that thin boundary loops slow down. We demonstrate that our results are immediately applicable to applications that use OPS, including TeaLeaf (achieving up to $3.56\times$) and OpenSBLI (achieving up to $1.71\times$).

Our algorithms and testing are also extended to distributed memory systems, demonstrating excellent scalability, maintaining speedup over the non-tiled versions as long as the problem size per socket is reasonably larger than the cache size. Performance is further improved by the communications scheme, particularly when strong scaling, due to using fewer but larger messages. Our work is also evaluated on Intel's Knights Landing platform, showing that even on an architecture with a much larger cache (16GB) our algorithms provide significant performance improvements when the full problem size grows beyond the capacity of the cache, and this improvement is maintained when weak scaled up to 128 nodes.

The fact that cache-blocking tiling can be applied with such ease to larger, non-trivial applications once again underlines the utility of domain specific languages, and their main premise: once an application is implemented using a high-level abstraction, it is possible to transform the code to near-optimal implementations for a variety of target architectures and programming models, without any modifications to the original source code. The algorithms presented in this paper are generally applicable to any stencil DSL that provides per-loop data access information.

ACKNOWLEDGMENTS

The authors would like to thank Michelle Strout at the University of Arizona for her invaluable suggestions and insight, as well as Fabio Luporini at Imperial College London. This paper was supported by the János Bólyai Research Scholarship of the Hungarian Academy of Sciences.

The OPS project is funded by the UK Engineering and Physical Sciences Research Council projects EP/K038494/1, EP/K038486/1, EP/K038451/1 and EP/K038567/1 on "Future-proof massively-parallel execution of multi-block applications" project. This research was also funded by the Hungarian Human Resources Development Operational Programme (EFOP-3.6.2-16-2017-00013). We acknowledge PRACE for awarding us access to resource Marconi based in Italy at Cineca.

REFERENCES

[1] M. Wolfe, "More iteration space tiling," in *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '89. New York, NY, USA: ACM, 1989, pp. 655–664.

[2] A. Darte, "On the complexity of loop fusion," in *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 149–.

[3] J. M. Levesque, R. Sankaran, and R. Grout, "Hybridizing s3d into an exascale application using openacc: An approach for moving to multi-petaflops and beyond," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, Nov 2012, pp. 1–11.

[4] W. Pugh and E. Rosser, *Iteration Space Slicing for Locality*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 164–184.

[5] I. J. Bertolacci, M. M. Strout, S. Guzik, J. Riley, and C. Olschanowsky, "Identifying and scheduling loop chains using directives," in *Proceedings of the Third International Workshop on Accelerator Programming Using Directives*, ser. WACCPD '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 57–67.

[6] T. Muranushi, H. Hotta, J. Makino, S. Nishizawa, H. Tomita, K. Nitadori, M. Iwasawa, N. Hosono, Y. Maruyama, H. Inoue, H. Yashiro, and Y. Nakamura, "Simulations of below-ground dynamics of fungi: 1.184 pflops attained by automated generation and autotuning of temporal blocking codes," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 3:1–3:11.

[7] B. Mohr, A. Malony, and R. Eigenmann, "On the integration and use of OpenMP performance tools in the SPEC OMP 2001 benchmarks," in *Proceedings of 2002 Workshop on OpenMP Applications and Tools (WOMPAT'02), Fairbanks, Alaska, August 2002*, 2002, record converted from VDB: 12.11.2012.

[8] L. N. Pouchet, "PolyBench: The Polyhedral Benchmark suite." [Online]. Available: http://web.cs.ucla.edu/~pouchet/software/polybench/

[9] 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 *Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, ser. WOLFHPC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 58–67.

[10] 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*. Cham: Springer International Publishing, 2015, pp. 85–104.

[11] S. P. Jammy, G. R. Mudalige, I. Z. Reguly, N. D. Sandham, and M. Giles, "Block-structured compressible navier–stokes solution using the ops high-level abstraction," *International Journal of Computational Fluid Dynamics*, vol. 30, no. 6, pp. 450–454, 2016.

[12] J. Meng, X.-J. Gu, D. R. Emerson, G. R. Mudalige, I. Z. Reguly, and M. B. Giles, "High-level abstraction for block structured application: A lattice boltzmann exploration," in *Proceedings of the EMerging Technology (EMiT) Conference*, 2016.

[13] A. Mallinson, D. A. Beckingsale, W. Gaudin, J. Herdman, J. Levesque, and S. A. Jarvis, "Cloverleaf: Preparing hydrodynamics codes for exascale," in *The Cray User Group 2013*, 2013.

[14] J. A. Herdman, W. P. Gaudin, S. McIntosh-Smith, M. Boulton, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis, "Accelerating hydrocodes with openacc, opencl and cuda," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, Nov 2012, pp. 465–471.

[15] "TeaLeaf: Uk mini-app consortium," 2015, https://github.com/UK-MAC/TeaLeaf.

[16] C. T. Jacobs, S. P. Jammy, and N. D. Sandham, "Opensbli: A framework for the automated derivation and parallel execution of finite difference solvers on a range of computer architectures," *Journal of Computational Science*, vol. 18, pp. 12 – 23, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S187775031630299X

[17] M. Wolfe, "Loops skewing: The wavefront method revisited," *International Journal of Parallel Programming*, vol. 15, no. 4, pp. 279–293, 1986. [Online]. Available: http://dx.doi.org/10.1007/BF01407876

[18] M. J. Wolfe, "Iteration space tiling for memory hierarchies," in *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*. Philadelphia, PA, USA: Society for

Industrial and Applied Mathematics, 1989, pp. 357–361. [Online]. Available: http://dl.acm.org/citation.cfm?id=645818.669220
[19] M. J. Wolfe, "Optimizing supercompilers for supercomputers," Ph.D. dissertation, Champaign, IL, USA, 1982, aAI8303027.
[20] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly — performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 04, p. 1250010, 2012.
[21] C. Ancourt and F. Irigoin, "Scanning polyhedra with do loops," in *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '91. New York, NY, USA: ACM, 1991, pp. 39–50. [Online]. Available: http://doi.acm.org/10.1145/109625.109631
[22] D. K. Wilde, "A library for doing polyhedral operations," IRISA, Tech. Rep. 785, December 1993.
[23] M. Geigl, "Parallelization of loop nests with general bounds in the polyhedron model," *Master's thesis, Universit at Passau*, 1997.
[24] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan, "A stencil compiler for short-vector simd architectures," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13. New York, NY, USA: ACM, 2013, pp. 13–24.
[25] R. Strzodka, M. Shaheen, and D. Pajak, "Time skewing made simple," *SIGPLAN Not.*, vol. 46, no. 8, pp. 295–296, Feb. 2011.
[26] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The pochoir stencil compiler," in *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '11. New York, NY, USA: ACM, 2011, pp. 117–128.
[27] R. T. Mullapudi, V. Vasista, and U. Bondhugula, "Polymage: Automatic optimization for image processing pipelines," *SIGARCH Comput. Archit. News*, vol. 43, no. 1, pp. 429–443, Mar. 2015. [Online]. Available: http://doi.acm.org/10.1145/2786763.2694364
[28] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: ACM, 2013, pp. 519–530. [Online]. Available: http://doi.acm.org/10.1145/2491956.2462176
[29] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model," in *International Conference on Compiler Construction (ETAPS CC)*, Apr. 2008.
[30] M. M. Baskaran, N. Vydyanathan, U. K. R. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, "Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors," *SIGPLAN Notices*, vol. 44, no. 4, pp. 219–228, February 2009. [Online]. Available: http://doi.acm.org/10.1145/1594835.1504209
[31] E. Schweitz, R. Lethin, A. Leung, and B. Meister, "R-stream: A parametric high level compiler," *Proceedings of HPEC*, 2006. [Online]. Available: http://llwww.ll.mit.edu/HPEC/agendas/proc06/Day2/21_Schweitz_Pres.pdf
[32] U. Bondhugula, "Compiling affine loop nests for distributed-memory parallel architectures," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 33:1–33:12. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503289
[33] M. Classen and M. Griebl, "Automatic code generation for distributed memory architectures in the polytope model," in *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, April 2006, pp. 1–7.
[34] T. Denniston, S. Kamil, and S. Amarasinghe, "Distributed halide," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '16. New York, NY, USA: ACM, 2016, pp. 5:1–5:12. [Online]. Available: http://doi.acm.org/10.1145/2851141.2851157
[35] "Using time skewing to eliminate idle time due to memory bandwidth and network limitations," in *Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 171–182. [Online]. Available: http://dl.acm.org/citation.cfm?id=846234.849346
[36] T. Malas, G. Hager, H. Ltaief, H. Stengel, G. Wellein, and D. Keyes, "Multicore-optimized wavefront diamond blocking for optimizing stencil updates," *SIAM Journal on Scientific Computing*, vol. 37, no. 4, pp. C439–C464, 2015.
[37] T. Grosser, S. Verdoolaege, A. Cohen, and P. Sadayappan, "The relation between diamond tiling and hexagonal tiling," *Parallel Processing Letters*, vol. 24, no. 03, pp. 1–20, 2014. [Online]. Available: http://www.worldscientific.com/doi/abs/10.1142/S0129626414410023
[38] P. Henderson and J. H. Morris, Jr., "A lazy evaluator," in *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages*, ser. POPL '76. New York, NY, USA: ACM, 1976, pp. 95–103. [Online]. Available: http://doi.acm.org/10.1145/800168.811543
[39] A. Bloss, P. Hudak, and J. Young, "Code optimizations for lazy evaluation," *Lisp and Symbolic Computation*, vol. 1, no. 2, pp. 147–164, 1988.
[40] L. W. Howes, A. Lokhmotov, A. F. Donaldson, and P. H. J. Kelly, "Deriving efficient data movement from decoupled access/execute specifications," in *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, ser. HiPEAC '09. Springer-Verlag, 2009, pp. 168–182.
[41] M. M. Strout, F. Luporini, C. D. Krieger, C. Bertolli, G. T. Bercea, C. Olschanowsky, J. Ramanujam, and P. H. J. Kelly, "Generalizing run-time tiling with the loop chain abstraction," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, May 2014, pp. 1136–1145.
[42] J. Xue, *Loop tiling for parallelism*. Springer Science & Business Media, 2012, vol. 575.
[43] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," *SIGPLAN Not.*, vol. 43, no. 6, pp. 101–113, Jun. 2008.
[44] V. Bandishti, I. Pananilath, and U. Bondhugula, "Tiling stencil computations to maximize parallelism," 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. 40:1–40:11.
[45] 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
[46] "CloverLeaf Reference github repository," 2013, https://github.com/UK-MAC/CloverLeaf_ref.
[47] "OPS github repository," 2013, https://github.com/gihanmudalige/OPS.

István Z. Reguly is a lecturer at PPCU ITK, Hungary. He holds an MSc and a PhD in computer science from the PPCU, Hungary. His research interests include high performance scientific computing on many-core hardware and domain specific active libraries for structured and unstructured meshes.

Gihan R. Mudalige is an assistant professor at the Department of Computer Science, University of Warwick, UK. His research interests are in performance analysis/optimization of scientific applications on high-performance systems. Previously he has worked as a research fellow at the High Performance Systems Group at Warwick and a research intern at the University of Wisconsin—-Madison, US. Dr. Mudalige holds a PhD. in Computer Science from the University of Warwick and is a member of the ACM.

Michael B. Giles is Professor of Scientific Computing in Oxford University's Mathematical Institute where he carries out research into the development and analysis of more efficient Monte Carlo methods for computational finance and engineering uncertainty quantification. He leads research into the use of GPUs for a variety of applications.