An Analytical Study of Loop Tiling for a Large-scale Unstructured Mesh Application

M.B. Giles*, G. R. Mudalige*, C. Bertolli†, P.H.J. Kelly†, E. László‡ and I. Reguly*
*Oxford e-Research Centre, University of Oxford, 7, Keble Road, Oxford OX1 3QG
email: mike.giles@maths.ox.ac.uk, {gihan.mudalige, istvan.reguly}@oerc.ox.ac.uk
†Dept. of Computing, Imperial College London, U.K.
email: {c.bertolli,p.kelly}@imperial.ac.uk
‡Pázmány Péter Catholic University, Faculty of Information Technology, Budapest, Hungary
email: lasen@digitus.itk.ppke.hu

Abstract—Increasingly, the main bottleneck limiting performance on emerging multi-core and many-core processors is the movement of data between its different cores and main memory. As the number of cores increases, more and more data needs to be exchanged with memory to keep them fully utilized. This critical bottleneck is already limiting the utility of processors and our ability to leverage increased parallelism to achieve higher performance. On the other hand, considerable computer science research exists on tiling techniques (also known as sparse tiling), for reducing data transfers. Such work demonstrates how the increasing memory bottleneck could be avoided but the difficulty has been in extending these ideas to real-world applications. These algorithms quickly become highly complicated, and it has be very difficult to for a compiler to automatically detect the opportunities and implement the execution strategy. Focusing on the unstructured mesh application class, in this paper, we present a preliminary analytical investigation into the performance benefits of tiling (or loop-blocking) algorithms on a real-world industrial CFD application. We analytically estimate the reductions in communications or memory accesses for the main parallel loops in this application and predict quantitatively the performance benefits that can be gained on modern multi-core and many core hardware. The analysis demonstrates that in general a factor of four reduction in data movement can be achieved by tiling parallel loops. A major part of the savings come from contraction of temporary or transient data arrays that need not be written back to main memory, by holding them in the last level cache (LLC) of modern processors.

1 Introduction

Increasingly, the main bottleneck limiting performance on emerging multi-core and many-core processors is due to the movement of data between its many different cores and memory. Data movement is crucial not only for performance (the latency in fetching data from main memory can be many 100s of clock cycles) but also for energy efficiency (the energy cost of a DRAM read/write is approximately 1000 times greater than the cost of a double precision floating point operation [1]). As the number of cores increases, more data needs to be exchanged to keep them fully utilized. Such a critical bottleneck is already limiting the utility of emerging processors and our ability to leverage increased parallelism to achieve higher performance. Considerable computer science research [2]–[7] on communication-avoiding algorithms such as "tiling" or "loop blocking" has existed for well over a decade. These algorithms aim to increase the reuse of data within L1 and L2 caches to reduce the data transfer to/from main memory. Savings of up to factor 10 have been achieved for simple applications [7]. However, the difficulty has been in extending these ideas to real-world applications, because of the sheer complexity that needs to be overcome by application developers if these are to be implemented manually, or due to the challenges in developing a compiler to automatically detect the opportunities for such tiling and then implement the execution strategy.

In this paper, we present an analytical investigation into the performance benefits of tiling based on a real-world industrial application. The application, Hydra, is a large-scale, unstructured mesh CFD code used for turbomachinery design at Rolls Royce plc. Our aim is to detail key tiling approaches for this application and quantify the performance gains that can be achieved on modern multi-core and many-core processor architectures. The specific contributions of this paper are twofold:

1) We present tiling (or loop blocking) approaches for unstructured mesh applications and analytically derive the reduction in communications or memory accesses when applied to the main parallel loops in the Hydra CFD application.
2) Based on hardware specifications of modern multi-core (e.g. Intel Sandy Bridge and Ivy Bridge) and many core (e.g. Intel MIC) processor architectures, particularly considering the utilization of the last level caches (LLCs), we estimate the performance gains achievable for Hydra on these platforms.

The conclusions from this study are that a conservative estimate of a factor 4 reduction in data transfer is achieved. Our work motivates further research to develop the implementation to achieve these savings. The rest of this paper is organized as follows: in Section 2

```
for (i=0; i<N; i++) res[i] = 0.0;    //loop 1
for (i=0; i<N-1; i++) {              //loop 2
  flux = flux_function(q[i],q[i+1]);
  res[i]   -= flux;
  res[i+1] += flux;
}
for (i=0; i<N; i++)                  //loop 3
  q[i] += dt*res[i];
```

Fig. 1. A trivial 1D application with 3 different loops

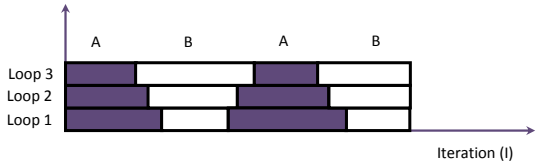
we give an overview of tiling with details of related work in this area; Section 3 presents the performance benefits of tiling when applied to the Hydra CFD application. Finally Section 4 concludes the paper.

2 Background

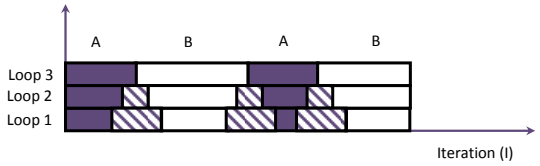
Tiling methods date back to techniques used to optimize dense matrix-matrix multiplication by reusing data within L1 and L2 caches. Related work in this area include [2]–[9]. The objective is to keep the active sections of the data arrays in the caches, as long as possible, thus avoiding the need to read/write to the slower main memory. Structured and unstructured mesh applications are often bandwidth limited, and some research has shown up to a factor 10 improvement in performance for simple structured mesh applications overlapping the execution of multiple iterations of an algorithm [7]. However the key difficulty has been in extending these ideas to real-world applications, because of the challenges of developing a compiler to automatically detect the opportunities for such overlapping and then implement the tiling execution strategy.

The tiling methods presented and analyzed in this paper are aimed at unstructured mesh applications, based on the OP2 [10] abstraction library. Unlike previous work, we examine a real-world application, used in an industrial production chain. Furthermore we analyze our performance based on an extended version of the standard split tiling used in the Pochoir compiler [7] by utilizing redundant computation at the borders. Our motivation is that on modern processor architectures, computation costs significantly less than the cost for data movement. As such carefully trading one for the other will lead to better utilization of the resources on a processor and its memory resulting in increased performance.

To explain the tiling optimizations that we are considering for Hydra, in this paper, it is helpful to review a simple concrete example. Fig. 1 gives some code for a trivial 1D application. Fig. 2 shows how the computation can be tiled. Each of the vertical "towers" is computed as a single mini-task, reusing data held within the cache. Fig. 2(a) illustrates a standard tiling construction (also called split tiling) used in the literature, such as the "hyperzoid" construction used by the MIT Pochoir compiler. The towers labeled *A* can all be computed in parallel, with one thread per tower, then after a global synchronization the towers labeled *B* can be computed.



(a) Parallelized split tiling



(b) Parallelized tiling with redundancy

Fig. 2. Tiling of iteration spaces for the three loops: the usual split tiling in (a), and the tiling parallelization with redundancy in (b)

```
#pragma omp parallel for private(offset, flux)
for (int t=0; t<nt; t++) {
  offset = t*T_SIZE;  flux = 0.0;
  for (int k=0; k<T_SIZE+2; k++){
    work[t*(T_SIZE+2)+k] = 0.0;
  }
  for (int k=0; k<T_SIZE+1; k++) {
    int i = k + offset;
    if (i<N-1) {
      flux = flux_function(q[i],q[i+1])
      work[t*(T_SIZE+2)+k] -= flux;
      work[t*(T_SIZE+2)+k+1] += flux;
    }
  }
  for (int k=0; k<T_SIZE; k++) {
    int i = offset + k;
    if (i<N) {
      q_new[i] = q[i]+dt*work[t*(T_SIZE+2)+k];
    }
  }
}
q = q_new;
```

Fig. 3. 1D application after tiling with redundancy (`T_SIZE` - tile size, `n_tiles` - number of tiles)

At the interface between the towers, the *B* calculations make use of some results previously computed by the *A* towers. Fig. 2(b) illustrates a different tiling approach based on a common technique in distributed-memory parallel computing. Here, each thread performs all of the calculations required to determine the final result for its tile at the end of loop 3. This allows all towers to be calculated in parallel, but at the interface between the towers, there is some redundant computation (indicated by the cross-hatched pattern). The main benefit is not the increased parallelism but reduced communication; the intermediate results (e.g. the array `res`) need never be stored in the main memory, they can just be transient values within the L2/L3 cache. In 1D, the dependency analysis for constructing the towers is quite trivial. The extension of the ideas to unstructured meshes has not been tackled in the literature because of the dependence on run-time data.

The application code after applying redundant parallel tiling is given in Fig. 3. It is apparent from this figure, how even a trivial set of 1D loops can become considerably complicated, when tiling optimizations are applied. The tiling with redundancy works by making use of a private scratch-pad of memory (`work[]`) for each tile and keeping it within the cache. Each thread executing the first loop needs to initialize its portion of the `work` array to zero, which is then used in the second loop to compute the contributions from the flux calculation. The second loop iterates over `T_SIZE+1` such that the final loop, operating over `T_SIZE` elements can access `work[t_ID][k+1]`. In the final loop, the updating of `q` is performed using a different array `q_new` so as to not cause a data conflict at the borders of the tiles. If we directly write to `q`, then multiple tiles could read/write to/from the same location at the border.

Considering the code in Fig. 1, each iteration of `loop 1` performs a write into `res` (i.e. two memory accesses, one to load in the cache line and another to store it later). `loop 2` performs a read from `q` and a write to `res` (i.e. three memory accesses). Finally, `loop 3` reads `res` and then performs a write to `q` (i.e. three memory accesses). This, in total results in 8N memory accesses (where `N` is the set size) or transfers to or from main memory.

In contrast, when using the parallelized tiling with redundancy we get the following access counts: The `res` initialization will be performed in cache (using the `work` array) resulting in no reads or writes to memory, `loop 2` reads `q` and then updates `work` in cache (i.e. 1 memory access) and finally `q` is updated in cache and written back to main memory via `q_new` (i.e. 2 memory accesses). `res` does not have to be stored back in main memory, just need to hold a working set in cache. The parallelized tiling with redundancy gives a total of 3N transfers to or from main memory. This is a factor of 2.67 savings compared to the original version of the code.

3 Tiling for the Hydra CFD Application

Given the tiling with redundancy techniques, our objective is to understand whether opportunities for such tiling optimizations exists for Hydra, a large-scale industrial application. Hydra is a highly complex, configurable CFD application, used as the main production code for validating turbomachinery designs at Rolls Royce plc [11]. Simulations in Hydra vary in size, from small cases with about a million edges which run in a few minutes, up to large ones with over 100 million edges taking days on a cluster of multi-core processors. These features make Hydra an interesting case to explore the benefits of optimizing communications or data accesses.

Hydra is implemented based on the OPlus [12], [13] abstraction layer and is currently being converted to use its successor, OP2 [10], [14], [15]. Both OPlus and OP2 aim to decouple the specification of an unstructured mesh problem from its parallel implementation. OPlus provided an abstraction layer for parallelizing an application on distributed memory systems using MPI, while OP2 develops an "active" library framework for the solution of unstructured mesh applications, enabling the use of both inter- (e.g. distributed memory, MPI) and intra- (e.g. OpenMP, CUDA, OpenCL, AVX) node parallelism. The active library approach uses program transformation tools, so that a single application code written using OP2 is transformed into the appropriate form that can be linked against a target parallel implementation, enabling execution on different back-end hardware platforms.

The OP2 API, code generation framework and its performance have been previously presented in [14]–[17]. The OP2 approach to the solution of unstructured mesh problems involves breaking down the algorithm into four distinct parts: (1) sets (`op_sets`), (2) data on sets (`op_dats`), (3) connectivity (or mapping) between the sets (`op_maps`) and (4) operations over sets. These lead to an API through which any mesh or graph can be completely and abstractly defined. Depending on the application, a set can consist of nodes, edges, triangular faces, quadrilateral faces, or other elements. Associated with these sets are data (e.g. node coordinates, edge weights) and mappings between sets which define how elements of one set connect with the elements of another set. All the numerically intensive computations in the unstructured mesh application can be described as operations over sets. Within an application code, this corresponds to loops (an `op_par_loop` in the OP2 API) over a given set, accessing data through the mappings (i.e. one level of indirection), performing some calculations, then writing back (possibly through the mappings) to the data arrays. The elemental operation over a set element is defined by the user, allowing for any computation to be implemented within an `op_par_loop`. A more complete description of implementing Hydra using OP2 is presented in [18].

The extension of the tiling can be applied quite naturally within the context of OP2, based on the mapping (or connectivity) tables which are a key part of OP2. Additionally, the Access/Execute specification [19] which OP2 instantiates, declares the way in which the user's datasets are utilized within each parallel loop and is vital to computing the dependencies between the mesh elements. The strategy to implement tiling with OP2 will be to first identify the final "results" data arrays (usually arrays with write and read/write access) that needs to be written back to main memory at the end of the block loops that are to be fused with tiling. Then OP2 will need to work backwards (starting from the final loop in the block of loops to be fused) and identify the corresponding elements of all the other arrays that needs to be updated or accessed (if they are read only) in order to compute a "tile" of these final results arrays.

Hydra, consists of several components [20] and in this paper we use the non-linear solver configured to compute in double precision floating point arithmetic. Hydra can also be used to express multi-grid simulations, but for simplicity of performance analysis and re-

Algorithm *Hydra Time marching loop*

```
1.  savold[]:q(1),qo(2)
2.  grad[n,e,n]:ewt(1),vol(1),x(1),xp(1), q(1),
3.  qp(6),ql(6)
4.  vflux[n,e]:ewt(1),dist(1),x(1),xp(1), q(1),
5.  qp(1),ql(1),vres(4)
6.
7.  iflux[n,e]:ewt(1),x(1),q(1),vres(1),res(4)
8.  src[n]:vol(1),x(1),q(1),res(2)
9.  update[n]:jac(1),rhs(1),qo(1),res(1),q(2)
10.
11. iflux[n,e]:ewt(1),x(1),q(1),vres(1),res(4)
12. src[n]:vol(1),x(1),q(1),res(2)
13. update[n]:jac(1),rhs(1),qo(1),res(1),q(2)
14.
15. iflux[n,e]:ewt(1),x(1),q(1),vres(1),res(4)
16. src[n]:vol(1),x(1),q(1),res(2)
17. update[n]:jac(1),rhs(1),qo(1),res(1),q(2)
18.
19. grad[n,e,n]:ewt(1),vol(1),x(1),xp(1), q(1),
20. qp(6),ql(6)
21. vflux[n,e]:ewt(1),dist(1),x(1),xp(1), q(1),
22. qp(1),ql(1),vres(4)
23.
24. iflux[n,e]:ewt(1),x(1),q(1),vres(1),res(4)
25. src[n]:vol(1),x(1),q(1),res(2)
26. update[n]:jac(1),rhs(1),qo(1),res(1),q(2)
27.
28. iflux[n,e]:ewt(1),x(1),q(1),vres(1),res(4)
29. src[n]:vol(1),x(1),q(1),res(2)
30. update[n]:jac(1),rhs(1),qo(1),res(1),q(2)
```

$$\begin{aligned} TC_{orig} &= 3PN + 2(6E + 16N + 31PN) + \\ &\quad 5(3E + 32N + 14PN) \\ &= 27E + N(192 + 135P) \end{aligned} \quad (1)$$

Fig. 4. Data access counts in one time step of the Hydra time-marching loop

porting we utilize experiments with a single grid (mesh) level. The configuration and input meshes of Hydra in this experiments model a standard application in CFD, called NASA Rotor37 [21], [22] with 2 million edges. It is a transonic axial compressor rotor widely used for validation in CFD. For simplicity in this paper we only direct our efforts in analyzing the performance benefits on a single node, assuming that the tiling algorithms are only implemented using shared memory parallelization. Extending the study to distributed memory parallelization will be presented in future work.

We consider the main/essential loops in the current production implementation of Hydra in this analysis. The loops operate over two `op_sets`, nodes or edges and a number of data arrays (or `op_dats`) are defined on these sets:

`ewt(`$3E$`),vol(`N`),x(`$3N$`),xp(`$4N$`),jac(`$25N$`),`
`dist(`N`),q(`PN`),qo(`PN`),qp(`$3PN$`),ql(`PN`),`
`res(`PN`),rhs(`PN`),vres(`PN`)`

E = number of edges , N = number of nodes

P = dimension of PDE (5 in simplest case)

All the above data arrays holds double-precision floating-point data where for example, the notation `vres(`PN`)` indicates that the `vres()` array is defined on nodes and each node consists of P double-precision

Algorithm *Hydra Time marching loop*

```
1.  savold[]:q(1),qo(2)
2.  grad/vflux[n,e,n,n,e]:ewt(1),vol(1),dist(1),
3.  x(1),xp(1),q(1),vres(2)
4.
5.  iflux/src/update[n,e,n,n]:ewt(1),vol(1),x(1),
6.  jac(1),q(1),q_new(2),qo(1),vres(1),rhs(1)
7.  iflux/src/update[n,e,n,n]:ewt(1),vol(1),x(1),
8.  jac(1),q(1),q_new(2),qo(1),vres(1),rhs(1)
9.  iflux/src/update[n,e,n,n]:ewt(1),vol(1),x(1),
10. jac(1),q(1),q_new(2),qo(1),vres(1),rhs(1)
11.
12. grad/vflux[n,e,n,n,e]:ewt(1),vol(1),dist(1),
13. x(1),xp(1),q(1),vres(2)
14.
15. iflux/src/update[n,e,n,n]:ewt(1),vol(1),x(1),
16. jac(1),q(1),q_new(2),qo(1),vres(1),rhs(1)
17. iflux/src/update[n,e,n,n]:ewt(1),vol(1),x(1),
18. jac(1),q(1),q_new(2),qo(1),vres(1),rhs(1)
```

$$\begin{aligned} TC_{fusion1} &= 3PN + 2(3E + 9N + 3PN) + \\ &\quad 5(3E + 29N + 6PN) \\ &= 21E + N(163 + 39P) \end{aligned} \quad (2)$$

Fig. 5. Hydra data access counts - loop fusion version 1

values. We assume that reading a set element from a data array residing in main memory results in one memory access and a write or an increment of an element results in two memory accesses (load followed by a store). For one time-step of the main time marching loop in Hydra, the memory access counts as listed in Fig. 4 were observed. Specific loops, `savold`, `grad`, `vflux`, `iflux`, `src` and `update` forms a single iteration of the main time-marching loop.

The notation `[e,n]` indicates that there are two loops within the same routine where the first iterates over edges and the second iterates over nodes. The notation `vres(4)`, for example, indicates that four memory accesses are performed per set element (in this case per node, as `vres()` is defined on nodes). Consider counting the number of memory accesses related to individual loops in Fig. 4. For example, there are three `grad` loops, the first iterating over nodes, the second over edges and the final again over nodes. The arrays `ewt`, `vol`, `x`, `xp`, and `q` are all accessed once (i.e. one read operation) per set element. The `qp` and `ql` arrays are read and written to (i.e. 2 memory operations) per set element. For each of the three loops, assuming the values remain in cache while being used during one loop, but are displaced from cache before coming back to it in the next loop, gives us 6 memory operations in total for each array. This assumes a good numbering of the set elements has been used [23]. The total number of memory accesses is given in (2).

We can apply tiling with redundancy, to fuse `grad` and `vflux`. The motivation is to eliminate the need to write back `qp`, and `ql` making these array values exists only as transient values in cache (see Fig. 5). Further more `vres` will only need to be read once and written back once, reducing its access count to two. Similarly, fusing `iflux`, `src` and `update` allows to eliminate the read

```
Algorithm Hydra Time marching loop
1.  savold[]:q(1),qo(2)
2.  grad/vflux[n,e,n,n,e]:
3.  ewt(1),vol(1),dist(1),x(1),xp(1),q(1),vres(2)
4.
5.  3*iflux/src/update[n,e,n,n,n,e,n,n,n,e,n,n]:
6.  ewt(1),vol(1),x(1),jac(1),q(1),
7.  q_new(2),qo(1),vres(1),rhs(1)
8.
9.  grad/vflux [n,e,n,n,e] : ewt(1), vol(1),
    dist(1), x(1), xp(1), q(1), vres(2)
10.
11. 3*iflux/src/update[n,e,n,n,n,e,n,n,n,e,n,n]:
12. ewt(1),vol(1),x(1),jac(1),q(1),
13. q_new(2),qo(1),vres(1),rhs(1)
```

$$\begin{aligned} TC_{fusion2} &= 3PN + 2(3E + 9N + 3PN) \\ &\quad +2(3E + 29N + 6PN) \\ &= 12E + N(76 + 21P) \end{aligned} \quad (3)$$

Fig. 6. Hydra data access counts - loop fusion version 2

```
Algorithm Hydra Time marching loop
1.  savold[]:q(1),qo(2)
2.  all loops:ewt(1),vol(1),dist(1),x(1),
3.  xp(1),jac(1),q(1),q_new(2),vres(2),rhs(1)
```

$$TC_{fusion\_full} = 3E + N(34 + 9P) \quad (4)$$

Fig. 7. Hydra data access counts - full loop fusion

and write back to memory for `res`. Additionally, in this case `ewt, vol, x, q` and `vres` need only be loaded from memory once, minimizing further memory traffic. `q` is written back via `q_new` to avoid data conflicts as in Fig. 3, resulting in 2 memory accesses to read and write `q_new`.

Applying tiling to fuse loops for a second time (see loop fusion version 2 in Fig. 6) enables us to combine each block of `iflux/src/update` into one loop. The number of read accesses to `ewt, vol, x, jac, qo, vres` and `rhs` is now reduced to one read per array for the whole fused loop. Furthermore `q` is read and updated (via `q_new`) once.

A final, fully fused version of the loops is detailed in Fig. 7. In this case, each array that was previously read only, or read/write is accessed only once. As before `q_new` is needed to update `q`. A standard structured grid would have 3 edges per node, while an unstructured, tetrahedral mesh would have as many as 10 edges per node. Thus if we consider a middle case where there are 6 edges per node (i.e. $E = 6N$) and set the dimension of the PDE (P) to be 6, the total transfer cost for each version above can be estimated as in TABLE 1.

From TABLE 1 we see that, up to a factor of 10 reduction in the number of memory accesses could be achieved by progressively fusing the loops in Hydra by implementing tiling. The fusion of all the loops, however will lead to a significant increase in redundant computations (at the borders of the tiles), and may not be viable considering the sizes of the caches of processors. Both the first and second fusion versions appear to be practically

TABLE 1
Memory access cost comparison, $E = 6N, P = 6$

Version	Figure	Cost	$\frac{TC_{orig}}{TC_*}$
Original	$27E + N(192 + 135P)$	1164N	1
Fusion V1	$21E + N(163 + 39P)$	523N	2
Fusion V2	$12E + N(76 + 21P)$	274N	4
Full Fusion	$3E + N(34 + 9P)$	106N	10

TABLE 2
Data requirements for each fused loop, $E = 6N, P = 6$

Loop	data per loop	Bytes/ node
V1.`grad/vflux`	$3E + N(9 + 3P) = 45N$	360
`iflux/src/update`	$3E + N(29 + 6P) = 83N$	664
V2.`grad/vflux`	$3E + N(9 + 3P) = 45N$	360
`iflux/src/update`	$3E + N(29 + 6P) = 83N$	664
full fusion	$3E + N(34 + 9P) = 106N$	848

achievable on current multi-core processor architectures. Thus overall we can speculate that conservatively a factor of 4 reduction in data transfers could be achieved. For each version we can compute the amount of data that is to be held for each of the fused loops (see TABLE 2). Here we assume that each node holds one double precision floating-point value.

Intel's latest high-end Sandy Bridge and Ivy Bridge processors consists of up to 20 MB of L3 cache [24]. With 664 bytes per node (see TABLE 2), 20 MB of last level cache (LLC) capacity corresponds to approximately 32^3 nodes per tile for loop fusion versions 1 and 2. Considering emerging many core architectures such as Intel's Many Integrated Cores (MIC) architecture [25] we see that the LLC cache is the L2 cache, 512 KB per core with 60 cores in total on one chip (32MB in total), all shared between the cores. This will allow us to compute a slightly larger tile (approximately over 36^3 nodes per tile for loop fusion versions 1 and 2).

4 Conclusions

In this paper we presented an analytical study of the performance benefits that can be gained by applying loop tiling algorithms for the industrial CFD application, Hydra. Our analysis was based on the fusion of the main parallel loops to form transient data such that they need not be written back to main memory by holding them in the last level cache of modern processors. We quantitatively presented that, conservatively up to a factor of 4 reduction in data movement could be achieved for Hydra. Thus, the results in this paper provide us with valuable insights into the achievable performance from these optimizations, providing us with clear motivation to now develop and test an implementation of the tiling strategy described here. As such future work will implement the tiling optimizations presented and analyzed in this paper as part of the OP2 framework.

Acknowledgements

Funding for this research has come from the UK Technology Strategy Board and Rolls-Royce plc. through the Siloet project,

and the UK Engineering and Physical Sciences Research Council projects EP/I006079/1, EP/I00677X/1 on Multi-layered Abstractions for PDEs. The funding support from grants TÁMOP-4.2.1.B-11/2/KMR-2011-0002 and TÁMOP-4.2.2/B-10/1-2010-0014 is also gratefully acknowledged. We are thankful to Leigh Lapworth, Yoon Ho and David Radford at Rolls-Royce, for making the Hydra application available for this project.

REFERENCES

[1] W. Dally, "Power and Programmability: the Challenges of Exascale Computing." 2011, http://www.orau.gov/archI2011/presentations/dallyb.pdf.

[2] M. E. Wolf and M. S. Lam, "A Data Locality Optimizing Algorithm," *SIGPLAN Not.*, vol. 26, no. 6, pp. 30–44, May 1991.

[3] Y. Song and Z. Li, "New Tiling Techniques to Improve Cache Temporal Locality," *SIGPLAN Not.*, vol. 34, no. 5, pp. 215–228, May 1999.

[4] M. Kandemir, "Data space oriented tiling," in *Programming Languages and Systems*, ser. Lecture Notes in Computer Science, D. Le Métayer, Ed. Springer Berlin / Heidelberg, 2002, vol. 2305, pp. 269–285.

[5] I. Kadayif and M. Kandemir, "Data space-oriented tiling for enhancing locality," *ACM Trans. Embed. Comput. Syst.*, vol. 4, no. 2, pp. 388–414, May 2005.

[6] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Implicit and explicit optimizations for stencil computations," in *Proceedings of the 2006 workshop on Memory system performance and correctness*, ser. MSPC '06. New York, NY, USA: ACM, 2006, pp. 51–60.

[7] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The Pochoir Stencil Compiler," in *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, ser. SPAA '11. New York, NY, USA: ACM, 2011, pp. 117–128. [Online]. Available: http://doi.acm.org/10.1145/1989493.1989508

[8] M. M. Strout, L. Carter, and J. Ferrante, "Rescheduling for locality in sparse matrix computations," in *Proceedings of the 2001 International Conference on Computational Science*, ser. Lecture Notes in Computer Science, V.N.Alexandrov, J. Dongarra, and C.J.K.Tan, Eds. San Francisco, CA, USA: Springer-Verlag, May 28-30 2001.

[9] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, "Communication-optimal parallel and sequential qr and lu factorizations," *SIAM Journal on Scientific Computing*, vol. 34, no. 1, pp. A206–A239, 2012.

[10] M. B. Giles, "OP2 for Many-Core Platforms," 2011, http://www.oerc.ox.ac.uk/research/op2.

[11] "Rolls royce," http://www.rolls-royce.com/.

[12] P. I. Crumpton and M. B. Giles, "Multigrid Aircraft Computations Using the OPlus Parallel Library," *Parallel Computational Fluid Dynamics: Implementations and Results Using Parallel Computers*, vol. -, pp. 339–346, a. Ecer, J. Periaux, N. Satofuka, and S. Taylor, (eds.), North-Holland, 1996.

[13] D. A. Burgess, P. I. Crumpton, and M. B. Giles, "A Parallel Framework for Unstructured Grid Solvers," in *Computational Fluid Dynamics'94:Proceedings of the Second European Computational Fluid Dynamics Conference*, S. Wagner, E. Hirschel, J. Periaux, and R. Piva, Eds. John Wiley and Sons, 1994, pp. 391–396.

[14] M. B. Giles, G. R. Mudalige, Z. Sharif, G. Markall, and P. H. J. Kelly, "Performance analysis and optimization of the OP2 framework on many-core architectures," *The Computer Journal*, vol. 55, no. 2, pp. 168–180, 2012.

[15] G. Mudalige, I. Reguly, M. Giles, C. Bertolli, and P. Kelly., "OP2: An Active Library Framework for Solving Unstructured Mesh-based Applications on Multi-Core and Many-Core Architectures." in *Proceedings of Innovative Parallel Computing (InPar '12)*. IEEE, May 2012.

[16] C. Bertolli, A. Betts, G. R. Mudalige, M. B. Giles, and P. H. J. Kelly, "Design and Performance of the OP2 Library for Unstructured Mesh Applications," ser. Euro-Par 2011 Parallel Processing Workshops, Lecture Notes in Computer Science, 2011.

[17] M. Giles, G. Mudalige, B. Spencer, C. Bertolli, and I. Reguly, "Designing OP2 for GPU Architectures," *Journal of Parallel and Distributed Computing*, 2012.

[18] C. Bertolli, A. Betts, N. Loriant, G. Mudalige, D. Radford, M. Giles, and P. Kelly, "Compiler optimizations for industrial unstructured mesh cfd applications on gpus," March 2012.

[19] 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. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 168–182.

[20] "Hydra Website at Rolls Royce," http://www.rolls-royce.com/about/technology/systems_tech/design_systems_tools.jsp.

[21] L. Reid and D. Moore, "Design and Overall Performance of Four Highly Loaded, High-speed Inlet Stages for an Advanced High-pressure-ratio Core Compressor," NASA, Tech. Rep., 1978, tP-1337.

[22] J. Dunham and G. Meauze, "An AGARD Working Group Study of 3D Navier-Stokes Codes Applied to Single Turbomachinery Blade Rows," *ASME*, 1998.

[23] D. A. Burgess and M. B. Giles, "Renumbering unstructured grids to improve the performance of codes on hierarchical memory machines," *Adv. Eng. Softw.*, vol. 28, pp. 189–201, April 1997.

[24] "Intel Xeon Processor E7 Family," http://ark.intel.com/products/family/59139.

[25] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey, "Fast sort on cpus, gpus and intel mic architectures," Technical Report, 2010. [Online]. Available: http://techresearch.intel.com/userfiles/en-us/FASTsort_CPUsGPUs_IntelMICarchitectures.pdf