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 been 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

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 op-
timize 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 automati-
cally detect the opportunities for such overlapping and
then implement the tiling execution strategy.

The tiling methods presented and analyzed in this pa-
er 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 con-
sidering 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.

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

\[ TC_{\text{orig}} = 3PN + 2(3E + 16N + 31PN) + \frac{5(3E + 32N + 14PN)}{2} \]
\[ = 2E + N(192 + 135P) \]  

\[ TC_{\text{fusion\_1}} = 3PN + 2(3E + 9N + 3P) + 5(3E + 29N + 6PN) \]
\[ = 2E + N(163 + 39P) \]  

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

---

**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:

- \(ewt(3E), vol(N), x(3N), xp(4N), jac(25N), dist(N), q(PN), qo(PN), qo(3PN), q(PN), res(PN), rhs(PN), vres(PN)\)
- \(E = \text{number of edges}, N = \text{number of nodes}\)
- \(P = \text{dimension of PDE (5 in simplest case)}\)

All the above data arrays hold 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 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 floating-point data.
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), x(1), xp(1), q(1), vres(2)
3. 3*iflux/src/update[n,e,n,n,e,n,e,n,n,e,n,n]: ewt(1), vol(1), x(1), jac(1), q(1),
4. q_new(2), qo(1), vres(1), rhs(1)
5. grad/vflux [n,e,n,e,n,e,n,e,n,e,n,e,n]: ewt(1), vol(1), dist(1), x(1), xp(1), q(1), vres(2)
6. 3*iflux/src/update[n,e,n,n,e,n,e,n,n,e,n,n]: 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),
9. dist(1), x(1), xp(1), q(1), vres(2)
10. 11. 3*iflux/src/update[n,e,n,n,e,n,n,e,n,n,e,n,n]: ewt(1), vol(1), x(1), jac(1), q(1),
12. q_new(2), qo(1), vres(1), rhs(1)

\[ TC_{fusion2} = 3PN + 2(3E + 9N + 3PN) + 6E + 29N + 6PN = 12E + N(76 + 21P) \] (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), xp(1), jac(1), q(1), q_new(2), vres(2), rhs(1)
3. \[ TC_{fusion\_full} = 3E + N(34 + 9P) \] (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 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 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,

\[ \begin{align*}
\text{TABLE 1} & \\
\text{Memory access cost comparison, } E = 6N, P = 6 & \\
\text{Version} & \text{Figure} & \text{Cost} & T_{Core}\,\text{TC} \\
\text{Original} & 27E + N(192 + 135P) & 1164N & 1 \\
\text{Fusion V1} & 21E + N(163 + 39P) & 523N & 2 \\
\text{Fusion V2} & 12E + N(78 + 21P) & 274N & 4 \\
\text{Full Fusion} & 3E + N(34 + 9P) & 106N & 10 \\
\end{align*} \]

\[ \begin{align*}
\text{TABLE 2} & \\
\text{Data requirements for each fused loop, } E = 6N, P = 6 & \\
\text{Loop} & \text{data per loop} & \text{Bytes/node} \\
\text{V1.grad/vflux} & 3E + N(9 + 3P) & 45N & 360 \\
\text{iflux/src/update} & 3E + N(29 + 6P) & 83N & 664 \\
\text{V2.grad/vflux} & 3E + N(9 + 3P) & 45N & 360 \\
\text{iflux/src/update} & 3E + N(29 + 6P) & 83N & 664 \\
\text{full fusion} & 3E + N(34 + 9P) & 106N & 848 \\
\end{align*} \]
and the UK Engineering and Physical Sciences Research Council projects EP/I006079/1, EP/I006777X/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


