Advance Access publication on 12 July 2011
doi:10.1093/comjnl/bxr062

Performance Analysis and Optimization of the OP2 Framework on Many-Core Architectures

M.B. GILES[1], G.R. MUDALIGE[1], Z. SHARIF[2], G. MARKALL[2] AND P.H.J. KELLY[2]

[1]*Oxford e-Research Centre, University of Oxford, Oxford, UK*
[2]*Department of Computing, Imperial College London, London, UK*
**Corresponding author: gihan.mudalige@oerc.ox.ac.uk*

This paper presents a benchmarking, performance analysis and optimization study of the OP2 'active' library, which provides an abstraction framework for the parallel execution of unstructured mesh applications. OP2 aims to decouple the scientific specification of the application from its parallel implementation, and thereby achieve code longevity and near-optimal performance through re-targeting the application to execute on different multi-core/many-core hardware. Runtime performance results are presented for a representative unstructured mesh application on a variety of many-core processor systems, including traditional X86 architectures from Intel (Xeon based on the older Penryn and current Nehalem micro-architectures) and GPU offerings from NVIDIA (GTX260, Tesla C2050). Our analysis demonstrates the contrasting performance between the use of CPU (OpenMP) and GPU (CUDA) parallel implementations for the solution of an industrial-sized unstructured mesh consisting of about 1.5 million edges. Results show the significance of choosing the correct partition and thread-block configuration, the factors limiting the GPU performance and insights into optimizations for improved performance.

Keywords: CFD; unstructured mesh applications; performance; GPU; OpenMP; CUDA

Received 20 December 2010; revised 15 June 2011
Handling editor: Stephen Jarvis

1. INTRODUCTION

Most scientific parallel programs have been and continue to be written by exclusively targeting a parallel programming model or a parallel architecture using extensions to traditional sequential languages such as Fortran, C or C++. This approach increases the cognitive load on programmers and has persisted primarily due to its good performance and the existence of legacy application software that remains critical to the production workloads of scientific organizations. However, the future of such a programming model's use in an environment of increasingly complex hardware architecture is unsustainable. This problem is becoming compounded as the High Performance Computing (HPC) industry begins to focus on delivering exascale systems in the next decade. Thus the current situation is both distracting scientists from investing their full intellectual capacity in understanding the physical systems they model, while also hindering their exploitation of the full capacity of available hardware. It is therefore clear that a level of abstraction must be achieved so that computational scientists can increase productivity without having to learn the intricate details of new architectures.

Such an abstraction enables users to focus on solving problems at a higher level and not worry about architecture-specific optimizations. This splits the problem space into (i) a higher application level, where scientists and engineers concentrate on solving domain-specific problems and write efficient code that remains unchanged for different underlying hardware architectures and (ii) a lower implementation level, which focuses on how a computation can effectively be made faster on a given architecture by carefully analysing the data access patterns. This paves the way for easily integrating support for any future heterogeneous hardware.

OPlus (Oxford Parallel Library for Unstructured Solvers) [1], a research project that had its origins in 1993 at the University of Oxford, provided such an abstraction framework for performing unstructured mesh-based computations across a distributed-memory cluster of processors [2]. OPlus is used as the underlying parallelization library for Hydra [3–5]—a production-grade CFD application used in turbomachinery design at Rolls-Royce plc. OP2 is the second iteration of OPlus and builds upon the features provided by its predecessor but develops an 'active' library approach with code generation to exploit parallelism on heterogeneous many-core architectures.

The 'active' library approach uses programme transformation tools, so that the user code is transformed into the appropriate form to be linked against the required parallel implementation (e.g. MPI, OpenMP, CUDA, OpenCL, AVX etc.) enabling execution on different target back-end hardware platforms [6]. Currently OP2 includes support for developing unstructured mesh applications for execution on multi-core and/or multi-threaded (OpenMP) CPUs and CUDA capable GPUs.

This paper presents a performance evaluation of the current OP2 library. Our objective is to provide a contrasting benchmarking and performance analysis study of a representative unstructured mesh application (Airfoil [7]) written using OP2 on a range of systems. These consist of representative current many-core hardware technologies such as the traditional X86 architecture systems from Intel and GPU offerings from NVIDIA. More specifically this paper make the following contributions:

(i) We present a performance analysis of the Airfoil unstructured mesh application written using OP2 on a number of multi-core CPU systems. OP2's code transformation framework is used to generate back-end code targeting multi-threaded executables based on OpenMP for two current multi-core processor systems: an Intel Xeon E5462 based on the older 'Penryn' micro-architecture and an Intel Xeon E5540 based on the current Intel 'Nehalem' micro-architecture. The end-to-end run times reported in this study are for the execution on an industrial-size problem using an unstructured mesh consisting of about 1.5 million edges.

(ii) The same Airfoil user source code is used by OP2's code generation tools to generate back-end code based on CUDA to be executed on NVIDIA GPUs. The runtime performance of this GPU code is compared against the multi-core, multi-threaded CPU performance. Performance results are presented for two GPUs, a GTX 260 consumer card and a Tesla C2050 based on the new Fermi architecture—NVIDIA's current flagship GPU offering.

(iii) Our analysis demonstrates the performance issues that distinguish the use of CPU and GPU architectures to execute the Airfoil application. These include the significance of choosing the correct partition and thread-block configuration, the factors limiting CPU and GPU performance and insights into optimizations for improved performance. We implement a number of hardware and software configurations and optimizations as part of this exercise for the GPU architecture.

The rest of this paper is organized as follows: Section 2 details related work in developing abstraction frameworks for multi-architecture platforms; Section 3 provides a description of the class of applications supported by OP2 and its API; Section 4 details the OP2 framework and key issues related to parallelizing unstructured mesh applications; Section 5 and Section 6 present performance figures for the execution of Airfoil on CPU and GPU systems, respectively, including comparisons between the two architectures; Section 7 investigates the factors limiting CPU and GPU performance including comparative performance on the GPUs for a number of hardware/software configurations and optimizations. Finally Section 8 concludes the paper.

2. RELATED WORK

Although OPlus pre-dates it, OPlus and OP2 can be viewed as an instantiation of the AEcute (access-execute descriptor) [8] programming model that separates the specification of a computational kernel with its parallel iteration space, from a declarative specification of how each iteration accesses its data. The decoupled Access/Execute specification in turn creates the opportunity to apply powerful optimizations targeting the underlying hardware. A number of related research projects have implemented similar programming frameworks, including LISZT [9], and the Hybrid Multi-core Parallel Programming (HMPP) [10] workbench.

The HMPP workbench allows the user to annotate codelets with HMPP directives that characterize data access in an aggregate (rather than iteration-specific) manner. The programme is then processed through the tool-chain, which uses the hardware vendor specific SDKs to translate it into platform-specific code. The resulting executable is run under the 'HMPP Runtime', which manages the resources and makes it possible to run a single binary on various heterogeneous hardware platforms. HMPP has no specific support for computation on graphs or unstructured meshes.

LISZT is a domain-specific language (embedded in Scala [11]) specifically targeting unstructured mesh codes, and is thus more directly comparable. The aim, as with OP2, is to exploit information about the structure of data and the nature of the algorithms in the code, to apply aggressive and platform-specific optimizations.

To our knowledge, performance figures for the execution of full scale applications, particularly industrial strength codes developed using the HMPP workbench, have not been published. Preliminary performance figures from the LISZT framework have been presented in DeVito *et al.* [12]. The authors report the performance of Joe, a fluid flow unstructured mesh application using a mesh of 750 K cells. Joe is first ported to the LISZT framework and the resulting code compared with the original code running on a cluster of 4-socket 6-core 2.66 GHz Xeon CPUs each with 36 GB RAM per node using MPI. Both codes demonstrate equivalent performance illustrating that no performance loss has resulted due to the use of the LISZT framework. Speed-up figures for the above code running on a Tesla C2050 (implemented using CUDA) against an Intel Core 2 Quad, 2.66 GHz processor are also provided, with results showing a speed-up of about 30× in single precision arithmetic and 28× in double precision relative to a single CPU thread.

Related work in the solution of unstructured mesh applications on GPUs, particularly in the CFD domain have also appeared elsewhere. In Corrigan *et al.* [13], techniques to implement an unstructured mesh solver on GPUs are described. Implementing three-dimensional Euler equations for inviscid, compressible flow are considered. Average speed-ups of about 9.5× are observed during the execution of the GPU implementation on an NVIDIA Tesla 10 series card against an equivalent optimized 4-thread OpenMP implementation on a quad-core Intel Core 2 Q9450.

Similarly [14] reports the GPU performance of a Navier–Stokes solver for steady and unsteady turbulent flows on unstructured/hybrid grids. The computations were carried out on NVIDIA's GeForce GTX 285 graphics cards (in double precision arithmetic) and speed-ups up to 46× (vs. a single core of two Quad Core Intel Xeon CPUs at 2.00 GHz) are reported.

Research in GPU acceleration often cites speed-ups, relative to a hand-coded CPU implementation—sometimes even comparing to a single-core. In this paper, we compare performance on contemporary flagship platforms (NVIDIA C2050, Intel 8-core Penryn and Nehalem). Our goal with OP2 is to generate highly-optimized code for X86 multi-core platforms (via OpenMP and the Intel compiler), as well as for GPUs, from the same code.

3. BACKGROUND

3.1. Unstructured mesh applications

The geometric flexibility of unstructured grids has proved invaluable over a wide area of computational science for solving PDE's (partial differential equations) including: CFD (computational fluid dynamics); CEM (computational electromagnetics); structural mechanics; and general finite element methods. In three dimensions millions of elements are often needed for the required solution accuracy, leading to a large computational cost.

The OPlus approach to the solution of such unstructured mesh problems involves breaking down the unstructured grid algorithms into four distinct parts: (i) sets, (ii) data on sets, (iii) mappings between sets and (iv) operations over sets. These lead to an API through which one can completely and abstractly define any mesh or graph.

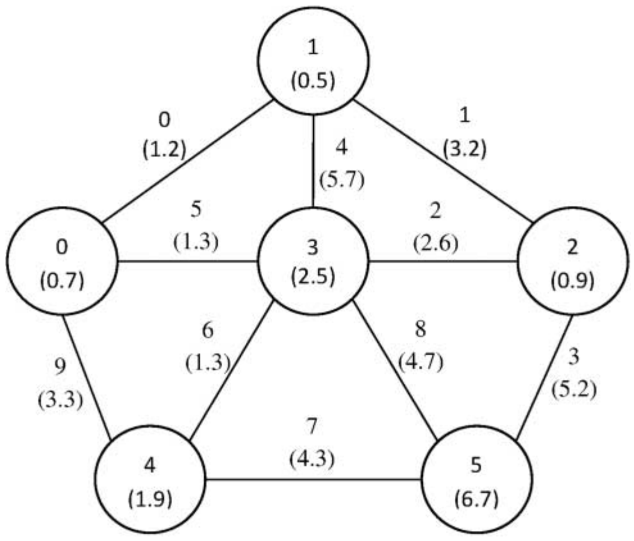


FIGURE 1. An example mesh with node and edge indices, and associated data values in parenthesis.

Unstructured meshes, unlike structured meshes, use connectivity information to specify the mesh topology. Depending on the application, a set can consist of nodes, edges, triangular faces or other elements. Associated with these sets are data (e.g. node coordinates, edge weights) and mappings between sets that define how elements of one set connect with the elements of another set. Figure 1 illustrates a simple triangular mesh that we will use as an example to describe the OP2 API.

The mesh illustrated in Fig. 1 can be defined as two sets, nodes (vertices) and edges, each with their sizes, using the API as follows:

```
op_set nodes;
op_decl_set(6, nodes, "nodes");
op_set edges;
op_decl_set(10, edges, "edges");
```

The connectivity is declared through mappings between the sets. The integer type array `edge_map` can be used to represent how an edge is connected to two different vertices.

```
int edge_map[20] = {0, 1, 1, 2, 2, 3, 2, 5, 1,
                    3, 0, 3, 3, 4, 4, 5, 3, 5,
                    0, 4};
op_map pedge;
op_decl_map(edges,nodes,2,edge_map,pedge,
            "pedge");
```

Each element of `edges` is mapped to two different elements in `nodes`. In our example, an `edge_map` entry has a dimension of 2 and thus for example its index 0 and 1 maps edge 0 to the vertices 0 and 1, index 2 and 3 maps edge 1 to vertices 1 and 2 and so on. Thus the `edge_map` array define the connectivity between the two sets. When declaring a mapping we first pass the source (e.g. edges) then the destination (e.g. nodes). We then pass the dimension of each element (e.g. 2; as edge_map maps each edge to 2 nodes). Once the sets are defined, various data can be associated with them; the following are some arrays that contain data associated with edges and vertices, respectively.

```
float dEdge[10] = {1.2, 3.2, 2.6, 5.2, 5.7,
                   1.3, 1.3, 4.3, 4.7, 3.3};
float dNode[6] = {0.7, 0.5, 0.9, 2.5, 1.9,
                  6.7};
float*dNode_u=(float*)malloc(sizeof(float)*6);
op_dat data_edges;
op_decl_dat(edges, 1, "float", dEdge,
            data_edges, "data_edges");
op_dat data_nodes;
op_decl_set(nodes, 1, "float", dNode,
            data_nodes, "data_nodes");
op_dat data_nodes_u;
op_decl_dat(nodes,1,"float", dNode_u,
            data_nodes_u, "data_nodes_u");
```

Note that here a single float per set element is declared in this example. A vector of a number of values per set element could also be declared (e.g. a vector with three floats per vertex to store the vertex coordinates).

All the numerically intensive parts of an unstructured mesh application can be described as operations over sets. Within a code this corresponds to a loop over a given set, accessing data through the mapping arrays (i.e. one level of indirection) performing some arithmetic, then writing (possibly through the mappings) back to data arrays. If the loop involves indirection through a mapping, we refer to it as an indirect loop; if not, it is called a direct loop. The OP2 library provides a parallel loop declaration syntax that allows the user to declare the computation over the sets in these loops [15]. For the mesh illustrated in Fig. 1 a loop over all the edges (where number of edges, nedge = 10) that updates the nodes can be written in the following sequential execution loop:

```
void seq_loop(int nedge, int *edge_map,
              float *dEdge, float *dNode,
              float *dNode_u)
{
  for (int e=0; e<nedge; e++)
         dNode_u[edge_map[2*e]] +=
                 dEdge[e] * dNode
                 [edge_map[2*e+1]];
}
```

A user declares this loop using the API as follows together with a one-edge kernel function. The library handles the architecture specific parallelization.

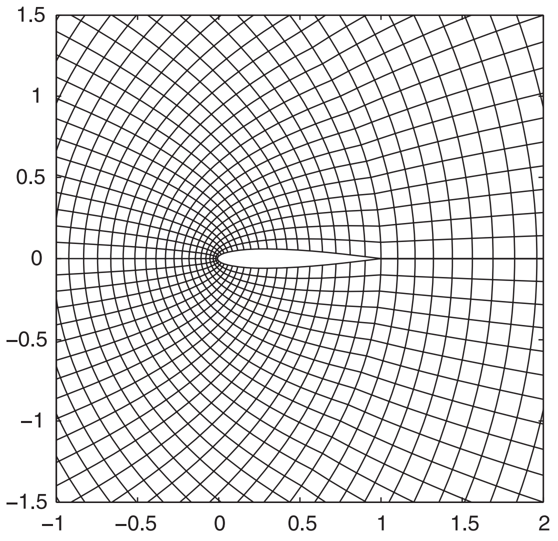


FIGURE 2. Rendering of a simple 120 × 60 mesh used in Airfoil.

```
void kernel(float* e, float* n, float* n_u)
{
  *n_u += e[0] * n[0];
}
op_par_loop_3(kernel,"kernel", edges,
       dEdge,  -1,OP_ID,  1,"float", P_READ,
       dNode,   1,pedge,  1,"float", OP_READ,
       dNode_u, 0,pedge,  1,"float", OP_INC);
```

This general decomposition of unstructured algorithms imposes no restriction on the actual algorithms, it just separates the components of a code. However, the type of calculations that can be done using OP2 is restricted to ones where the order in which elements are processed does not affect the final result, to within the limits of finite precision floating point arithmetic. This constraint allows the program to choose its own order to obtain maximum parallelism. Moreover the sets and mappings between sets must be static and the operands in the set operations cannot be referenced through a double level of mapping indirection (i.e. a mapping to another set, which in turn uses another mapping to data associated with a third set).

Although it might appear that these restrictions are quite severe, the straightforward programming interface and I/O treatment combined with efficient parallel execution makes it an attractive prospect, if the algorithm to be developed falls within the scope of OP2. For example, the API could be used for explicit relaxation methods such as Jacobi iteration; pseudo-time-stepping methods; multi-grid methods that use explicit smoothers; Krylov subspace methods with explicit preconditioning; semi-implicit methods where the implicit solve is performed within a set member—for example, performing block Jacobi where the block is across a number of PDE's at each vertex of a mesh. However, algorithms based on order dependent relaxation methods such as Gauss–Seidel or ILU (incomplete LU decomposition) lie beyond the capabilities of the API. These are not fundamental restrictions, but we believe they help limit the complexity of the API, and encourage programmers to code in a way that can be made efficient.

The example application used in our analysis, Airfoil, is a non-linear 2D inviscid airfoil code that uses an unstructured grid [7]. It is a much simpler application than the Hydra [16] production CFD application used at Rolls-Royce plc. for the simulation of turbomachinery, but acts as a forerunner for testing the OP2 library for many core architectures. A rendering of a smaller (120×60) unstructured mesh similar to the one used in Airfoil is illustrated in Fig. 2. The actual mesh used in our experiments is of size 1200×600, which is too dense to be reproduced here. This consists of over 720 K nodes, 720 K cells and about 1.5 million edges. The code consists of five parallel loops: `save_soln`, `adt_calc`, `res_calc`, `bres_calc`, `update`. The most compute intensive loop `res_calc` has about 100 floating-point operations performed per mesh edge and is called 2000 times during total execution of the application.

4. OP2

The original OPlus library [1] was developed over 10 years ago for MPI/PVM-based distributed memory execution of unstructured mesh algorithms written in Fortran. Its second iteration, OP2, is designed to leverage emerging many-core hardware (GPUs, AVX etc.) on top of distributed memory parallelism, allowing the user to execute on either a single multi-core/many-core node (including large shared memory systems), or a cluster of multi-core/many core nodes. Currently the OP2 library only supports code development in C/C++. A Fortran API will be developed later with similar functionality.

Since the OP2 specification provides no description of the low-level implementation, re-targeting to future architectures only requires the development of a new code-generation back-end. The current implementation focuses on CUDA (using the NVIDIA CUDA programming model [17]) and OpenMP and will later include code generation for OpenCL and Intel AVX, thus supporting a wide range of CPU and GPU hardware. OP2 will also include support for the above to be executed on distributed memory CPU and GPU clusters in conjunction with MPI.

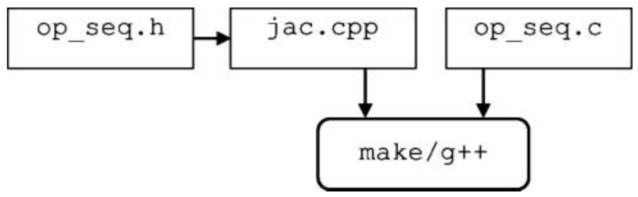


FIGURE 3. Sequential build process.

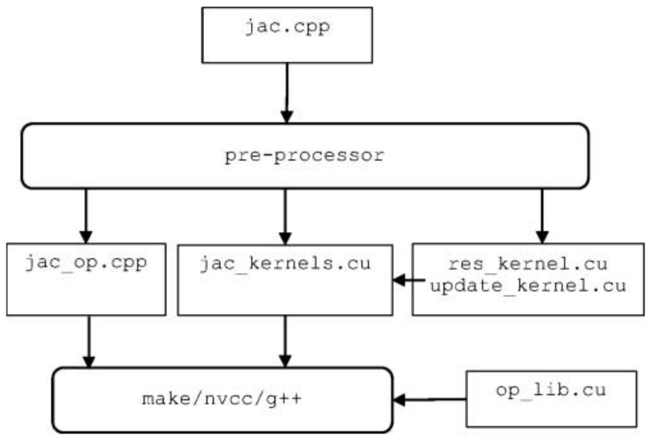


FIGURE 4. CUDA build process.

The OP2 strategy for building executables for different back-end hardware consists of firstly generating the architecture-specific code by parsing the user code (which is written using the OP2 API) through a pre-processor and then secondly linking the generated code with the appropriate parallel implementation library. Figures 3 and 4 illustrate and contrast the build process for generating a single-threaded CPU executable and a CUDA-based executable, respectively. For the single thread CPU executable, the user's main program (in this case `jac.cpp`) uses the OP header file `op_seq.h` and is linked to the OP routines in `op_seq.c` using g++, controlled by a Makefile. For the corresponding CUDA executable, the preprocessor parses the user's main program and produces a modified main program and a CUDA file, which includes a separate file for each of the kernel functions. These are then compiled and linked to the OP routines in `op_lib.cu` using g++ and the NVIDIA CUDA compiler nvcc, again controlled by a Makefile. The results presented in this paper come from code produced by a pre-processor written in MATLAB, which only parses the individual OP2 routine calls. A new pre-processor is being developed using the ROSE compiler framework [18]; this will parse the entire user code, allowing simplification of the API.

4.1. Data dependencies

A key technical difficulty in the parallel execution of unstructured mesh applications is the data dependency issue encountered when incrementing indirectly referenced arrays. Thus, for example, a potential problem arises when two edges update the same node. A solution at a coarse grained level would be to partition the nodes such that the *owner* of the nodal data would carry out the computation. The drawback in this case is redundant computation when the two nodes for a particular edge have different *owners*. At the finer grained level, we could assign a 'colour' for the edges so that no two edges of the same colour update the same node. This allows for parallel execution for each colour followed by a synchronization. The disadvantage in this case is a possible loss of data reuse and loss of some parallelism. A third method would be to use *atomic* instructions, which combine read/add/write into a single operation. However, atomic operations (especially on doubles) are not present on all the hardware platforms we are interested in (atomics are an optional extension in OpenCL), and performance varies between platforms.

The OP2 design addresses the data dependency problems using the already-mentioned three methods, at three levels of parallelism. Method 1 will be used in the future at the MPI level. Given a global mesh with sets and data, OP2 will partition the data so that the partition within each MPI process owns some of the set elements, i.e. some of the nodes and edges. These partitions only perform the calculations required to update their own elements. However, it is possible that one partition may need to access data that belong to another partition; in that case, a copy of the required data are provided by the other partition. This follows the standard 'halo' exchange mechanism used in distributed memory message passing parallel implementations. As the partition size becomes larger, the proportion of 'halo' data becomes very small.

For distributed memory architectures, the partition size is large. However, within a CPU or a GPU, operations are to be performed on a finer granularity on each processing unit. For a multi-core CPU, the processing units are processor cores (each based on a traditional heavy-weight core architecture such as X86 or IBM POWER). For NVIDIA GPUs the processing units are a number of relatively lightweight stream multiprocessors (SMs), each consisting of a number of stream processors (SPs) that share control logic, an instruction cache and a block of shared memory [17]. The new NVIDIA Fermi architecture also provides an L1 cache per SM.

To solve the data dependency issue on GPUs, indirect data from GPU global memory are loaded into each SM's shared memory space forming a local mini-partition. Each mini-partition is assigned to an SM and are executed in parallel. The SPs within an SM execute the mini-partition by utilizing a number of threads (called a thread block). Within a thread block, 32 threads at a time are executed in parallel (32 threads is called a *warp* in CUDA).

During the execution of an indirect loop, the loop receives data from shared memory instead of global memory, maximizing data reuse and minimizing the traffic between global memory and shared memory. Thus on the GPU, updating the same node could occur either (i) by multiple threads within a single processing unit (an SM) updating data held in its shared memory (i.e. mini-partition) or (ii) when the result of the shared memory are written back to the GPU global memory that is used by other processing units. In OP2 thread, colouring is used for the former and a block colouring is used for the latter.

Edges are coloured so that two edges with the same colour never update the same node (see Fig. 5). As a result, the edges with the same colour can be processed in parallel by different threads. The colouring is performed very efficiently in a runtime initialization using a bitwise operation on a 32 bit integer for each of the edges [15]. Similarly, a block colouring scheme is used so that results from shared memory, after processing a mini-partition, are not used by any other mini-partition being processed simultaneously. On a production grade CFD application such as Hydra, a single run would consist of over 100 K blocks, each needing to fit into the shared-memory of a GPU. Ten colours might be needed to avoid data conflicts, suggesting up to 10 K blocks per colour.

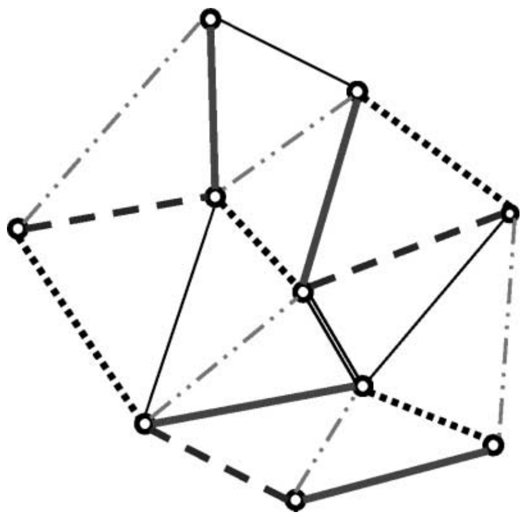


FIGURE 5. An example colouring of edges.

A similar technique is used for multi-core processors. The difference is that now, each mini-partition is executed by a single OpenMP thread. The mini-partitions are coloured to stop multiple blocks trying to update the same data in the main memory simultaneously. This technique is simpler than the GPU version as there is no need for global-local renumbering (for GPU global memory to shared memory transfer) and no need for low-level thread colouring.

4.2. Data layout in memory

Another important implementation decision is the data layout in memory when there are multiple components for each set element. For example, in our airfoil test-case, each cell has four flow variables; should these four components be stored contiguously for each cell (a layout that is sometimes referred to as an array-of-structs, AoS) or should all of the first components be stored contiguously, then all of the second components, and so on (a layout which is sometimes referred to as a struct-of-arrays, SoA)?

The SoA layout was natural in the past for vector supercomputers that streamed data to vector processors, but the AoS layout is natural for conventional cache-based CPU architectures. This is because an entire cache line must be transferred from the memory to the CPU even if only one word is actually used. This is a particular problem for unstructured grids with indirect addressing; even with renumbering of the elements to try to ensure that neighbouring elements in the grid have similar indices, it is often the case that only a small fraction of the cache line is used. This problem is worse for SoA compared with AoS, because each cache line in the SoA layout contains many more elements than in the AoS layout. In an extreme case, with the AoS layout each cache line may contain all of the components for just one element, ensuring perfect cache efficiency, assuming that all of the components are required for the computation being performed.

Until recently, NVIDIA GPUs did not have caches and most applications have used structured grids. Therefore, most researchers have preferred the SoA data layout, which leads to natural memory transfer 'coalescence' giving high performance. However, the latest NVIDIA GPUs based on the Fermi architecture have L1/L2 caches with a cache line size of 128 bytes, twice as large as used by Intel in its latest CPUs. This leads to significant problems with cache efficiency, especially since there is only 48 kB of local shared memory and so not many elements are worked on at the same time. Consequently, we have chosen to use the AoS layout. Our measurements indicate that this reduces the data transfer between global memory and the GPU by over 50%. Ensuring good coalescence in the data transfers requires some more complicated programming, but that is again a benefit of a library; it takes care of the complexity rather than burdening the application programmer with it.

5. MULTI-CORE CPU PERFORMANCE

Our first set of experiments is directed at comparing the performance of Airfoil using OpenMP on a single node comprising multi core, multi-threaded CPUs. This section presents the results for single precision performance on the CPUs. Double precision performance is reported in Section 6. Table 1 details briefly the specifications of each CPU system node. The Intel Xeon E5462 (based on the older Intel Penryn micro-architecture) node consists of two Intel Xeon E5462 quad-core (total of 8 cores) processors operating at 2.8 GHz clock rate per core and has access to 16 GB of main memory. The Intel Xeon E5540 processor based node, consist of two Intel Xeon E5540 quad-core (total of 8 cores) processors consisting of 2.5 GHz per core clock rate and access to 24 GB of main memory. These processors are based on Intel's current flagship Nehalem micro-architecture and have simultaneous multi threading (SMT) enabled for the execution of 16 SMT threads. For brevity and to avoid confusion for the rest of this paper, the Xeon E5462 will be referred to as the Penryn and the Xeon E5540 as the Nehalem. The Airfoil code and OP2 was compiled on both systems using the Intel ICC 11.1 compiler. The exact compiler flags used are detailed in Table 2. Both CPUs make use of the latest SSE instruction sets capable of execution on each processor architecture.

TABLE 1. CPU node system specifications.

Processor	Cores/node	Clock rate (GHz)	Memory/node (GB)
Intel Xeon E5462 (Penryn)	8	2.8	16
Intel Xeon E5540 (Nehalem)	8 (16 SMT)	2.5	24

As mentioned previously, OpenMP parallelism is achieved by OP2 on multi-core processors by partitioning the unstructured mesh assigned to the multi-core node and using one OpenMP thread for each mini-partition. Colouring is used to stop multiple mini-partitions interfering with the same data. Thus, a key parameter in our study will be to investigate the mini-partition size (from here on referred simply as partition size) that provides the best runtime for the Airfoil application for a given unstructured mesh.

Figure 6 presents the total runtime of Airfoil on the Penryn and Nehalem based nodes, compiled using the ICC compiler, for a range of partition sizes using up to 16 OpenMP threads. These experiments were executed at least five times, and the observed standard deviation in run times were significantly $<10\%$. The results reported here (and throughout the rest of the paper) are observed minimum run times. There is only a marginal difference between the performance of the two systems. Due to the higher clock rate on the Penryn, it exhibits better single core (single thread) performance. However, when using all 8 cores, we see about 30% better performance from the Nehalem with a best runtime of about 42 s on a partition size of 512 (running on 16 OpenMP threads). Regardless of the partition size, increasing the number of threads from 1 to 8 provides diminishing returns. Sixteen threads provides an even smaller performance benefit (if any) as two threads share a core using SMT. It appears that other factors of multi-core chips may be limiting their scalability. We have observed a bandwidth utilization of over 20 GB/s on the Nehalem system during

TABLE 2. CPU Compiler specifications.

System	Compiler Flags for ICC 11.1
Intel Xeon E5462 (Nehalem)	-parallel -O3 -ipo -vec-report -xSSE2,SSE3,SSE3,SSE4.1 -funroll-loops
Intel Xeon E5540 (Nehalem)	-parallel -O3 -ipo -vec-report -xSSE2,SSE3,SSE3,SSE4.1,SSE4.2 -funroll-loops

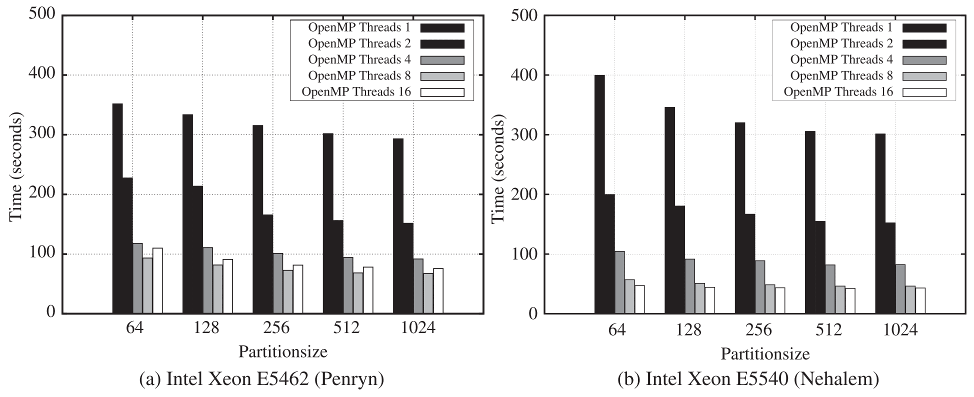


FIGURE 6. Runtime of Airfoil on the Intel processors Compiled with Intel CC 11.1 with up to 16 OpenMP threads—Single Precision (1000 Iterations).

TABLE 3. GPU node system specifications.

GPU	Cores	Clk (GHz)	Glob. Mem. (GB)	Shared Mem./SM (kB)	Driver version (Compute Cap.)
GeForce GTX260	216	1.4	0.8	16	3.2 (1.3)
Tesla C2050	448	1.15	3.0	48	3.2 (2.0)

TABLE 4. GPU Compiler specifications.

System	Compiler Flags for nvcc (nvcc built using gcc 4.4.5)
GTX260	-O3 -arch=sm_13 -Xptxas -Xptxas=-v -dlcm=ca -use_fast_math
Tesla C2050	-O3 -arch=sm_20 -Xptxas -Xptxas=-v -dlcm=ca -use_fast_math

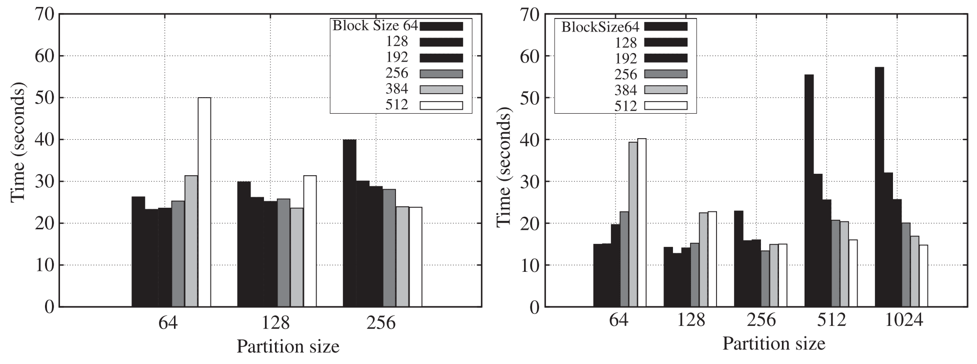
the execution of Airfoil. Given that the maximum available bandwidth of these processors is about 25 GB/s [19], the code appears to be saturating the processor's bandwidth capacity. We investigate memory bandwidth utilization in more detail later in Section 7.

6. GPU PERFORMANCE

Next, we explore the performance of the Airfoil code on two NVIDIA GPUs—the consumer grade GTX260 and the HPC-capable Tesla C2050 based on NVIDIA's current Fermi GPU architecture. The OP2 code transformation framework in this case generates CUDA code to be executed on the GPUs. Table 3 details the specifications of each system. The host CPU for the GTX260 is an AMD Athlon X2 dual core processor at 2 GHz, while for the Tesla the host is a quad-core Intel Xeon E5530 processor operating at 2.4 GHz. In both GPU systems NVIDIA's CUDA/C compiler `nvcc` was built using the GNU C compiler 4.4.5, which in turn is also used for compiling the host (non-gpu) code. The final column of Table 3 details the NVIDIA driver version used for each GPU and the compute capability. The compiler flags used for nvcc are detailed in Table 4.

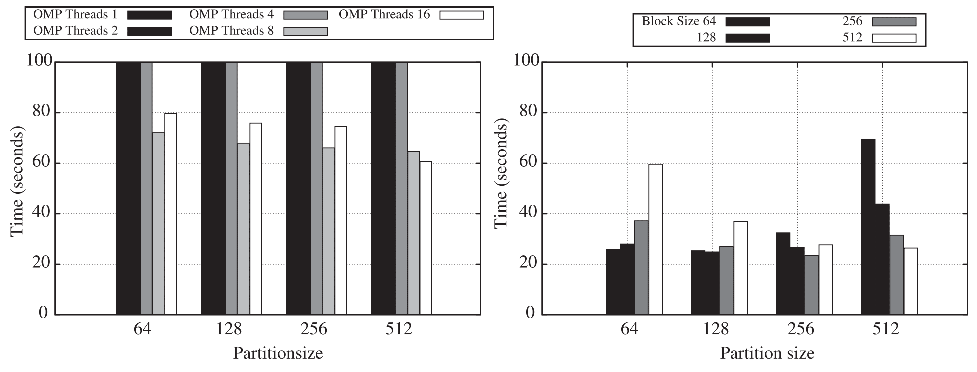
Figure 7 presents the total runtime of Airfoil (for single precision arithmetic) on the two NVIDIA cards. For these runs the number of CUDA threads allocated per mini-partition provides an additional configuration parameter. The GTX260 could only execute (mini-) partition sizes up to 256 due to its limited memory. The GTX260 performs only about two times slower than the Tesla C2050 per core due to their comparable single precision floating-point performance. The best performance—about 12 s—on the C2050 is achieved at a partition size of 256 running a thread-block size of 256. This is a speed-up of just over 3.5× compared with the Intel Nehalem processor system's performance on 16 OpenMP threads.

Given the mesh size, we can approximately compute the single precision floating point performance achieved on both the Nehalem and the C2050 during the most compute intensive loop, `res_calc`. The mesh consists of ∼1.5 million edges each responsible for 100 floating-point operations in `res_calc`. This routine is in turn called 2000 times giving 30×10^{10} floating-point operations in total. The best total time spent in the `res_calc` loop on the CPU (8 cores with SMT on) executing single-precision arithmetic is about 21 s. This translates to about 14 GFlops per second on the Intel Nehalem processor based system. The best total time spent in the `res_calc` loop on the Tesla C2050 is about 7.5 s. This is about 40 GFlops per second. Thus we see that only a fraction



(a)GTX260 (b)TeslaC2050

FIGURE 7. Runtime of Airfoil on NVIDIA GPUs on a range of partition sizes and thread-block size configurations—Single Precision (1000 Iterations).



(a) Intel Xeon E5540 (Nehalem) (ICC 11.1) (b) Tesla C2050

FIGURE 8. Runtime of Airfoil on Intel Xeon E5540 (Nehalem) and NVIDIA Tesla C2050—Double Precision (1000 Iterations).

of the peak single-precision floating-point performance on the GPU is achieved [20].

A key concern in determining whether GPUs are suitable for main-stream HPC and production scientific work is how their performance compares against the traditional processors when executing double precision floating-point codes. As this has been an increasing concern for the adoption of GPUs, NVIDIA has invested heavily in improved double-precision floating-point performance on their current Fermi based GPUs. The key benchmarking study for us therefore is to investigate the Airfoil execution in double-precision. Figure 8 details the double precision performance of Airfoil on the Intel Nehalem and Tesla C2050. The results demonstrate a speed-up of just over 2.5× on the Tesla C2050 (23 s) compared with the best runtime on the Intel Nehalem system (60 s). The total best runtimes spent in the `res_calc` loop executing double precision mathematics is 28 and 13 s, respectively, for 8 Nehalem cores and the C2050, respectively. This translates to a double precision floating-point performance of about 11 GFlops/s on the Nehalem and about 23 GFlops/s on the C2050.

Best overall runtimes according to Fig. 8 are achieved when the problem is configured to be executed with a partition size of 512 on the Nehalem and a partition size of 256 using a thread block size of 256 on the C2050. However, breaking down the runtime into the time taken by the five parallel loops reveals that the optimum partition size and the thread block size differs for each loop. Thus for example, the `res_calc` routine runs best when configured with a partition size of 256 and a thread block of 256 on the C2050, while `bres_calc` is optimized at a partition size of 64 and a thread block of 64. Recall that mini-partitions are only used in indirect loops (to avoid conflicts due to data dependencies); the partition size parameter has no meaning in direct loops.

Similar behaviour can be observed for the Nehalem runs. Thus it is apparent that further runtime improvements could be gained by simply configuring each parallel loop to be executed on its optimum partition size and thread block size settings. The ability to infer the optimum configuration could be gained through historical runtime observation, through a performance model or utilizing an auto-tuning mechanism. We are currently investigating the implementation of an auto-tuning mechanism within OP2. At the moment, the overall partition and block size parameters have a default value set during compile time, but these can be overridden at runtime using

TABLE 5. Best runtimes (double precision) for each parallel loop in Airfoil.

Loop	Part.	Time (S)	BW (GB/s)
Nehalem			
`save_soln`		2.62	24.57
`adt_calc`	64	14.80	8.75
`res_calc`	512	28.11	14.52
`bres_calc`	64	0.17	4.35
`update`		10.70	21.07
Total		56.40	

Loop	Part.	Block.	Time (S)	BW (GB/s)
C2050				
`save_soln`		128	0.80	80.68
`adt_calc`	256	256	3.60	45.20
`res_calc`	256	256	13.14	19.20
`bres_calc`	64	64	0.12	6.18
`update`		128	4.41	50.10
Total			22.07	

command-line arguments. For indirect loops both partition size and thread block size can be overridden. For direct loops only the thread block size is overridden, as partition size is not used. However, our experiments with different thread block sizes have demonstrated only a very weak effect on the run times of direct loops in Airfoil.

7. PERFORMANCE ANALYSIS AND OPTIMIZATION

Considering the best execution time for each parallel loop the total runtime for the Airfoil code on the Nehalem is about 56.4 s, while for the C2050 this is about 22.07 s. Table 5 details the time for each loop in conjunction with their observed memory bandwidth utilization for this case. The bandwidth figures were obtained by counting the total amount of data transferred with global memory during the execution of a parallel loop and dividing it by the runtime of the loop. Currently, only the total amount of useful data transferred is counted. In the future, this will be augmented to include bandwidth utilization due to cache line loading for specific CPU/GPU architectures.

As mentioned before, from observing the memory bandwidth figures on the Nehalem, it is apparent that during the most time-consuming parallel loop (`res_calc`) over 50% of the maximum available bandwidth (25.6 GB/s [19]) on the processor is utilized. Other loops such as `update` get much closer to saturating the available memory bandwidth on the CPU. Thus we suspect that the memory bandwidth of the single node system may become the bottleneck limiting future thread scalability of multi-core CPUs. The memory bandwidth utilization on the Tesla C2050 also comes close to the available upper limit of 144 GB/s [20], on `save_sol` but remains relatively low on `res_calc` suggesting that the higher compute intensity of this loop eases bandwidth saturation. Alternatively, there may be poor cache line efficiency; so the amount of data actually transferred is higher than indicated.

Given that each element on the unstructured mesh could be computed independently, it is not surprising that the GPU architecture outperforms the traditional multi-core processors. But it is interesting that only about 3.5× speed-up is achieved in single precision and only about 2.5× speed-up gained in double precision. One potential cause may be due to the cost of integer pointer arithmetic in computing indirect references in unstructured mesh computations. As there is no separate integer pipeline on the simple GPU cores, an integer operation costs as much as a floating-point operation (at least in single precision).

The remainder of this paper investigates further the performance of the Airfoil code in an attempt to uncover insights into optimizing the OP2 framework. We target the GPU architecture for our analysis/efforts in order to demonstrate the expert techniques required to maintain performance on this emerging architecture. Our goal is to demonstrate that using a framework such as OP2 does not sacrifice performance improvements that can be gained by low-level optimizations/configurations available for a target back-end architecture. At the same time, we demonstrate the significant benefits of such a framework that generates these improvements without the intervention of the application programmer.

7.1. Thread colouring

We first quantify the performance degradation due to the thread colouring discussed in Section 4.1. Recall that colouring is required to avoid data dependency conflicts while executing indirect loops on both the CPU and GPU. However, the use of thread colouring when updating data arrays in the GPU implementation leads to a reduction in parallelism as there are only a few active threads within each CUDA warp. To assess the impact of this, we can turn off the colouring (which can be done with negligible reimplementation effort); note that the values that are then computed will be incorrect due to the data dependencies.

Column 3 of Table 6 presents the percentage performance difference when the Tesla C2050 executes the Airfoil code without thread colouring. Note that thread colouring is not applicable for direct loops. The maximum performance difference is in the `res_calc` loop, which execute 23% faster without thread colouring. The overall performance difference is about 14%, which suggests that there may be further room for improvement of the thread colouring implementation for the GPU architecture.

TABLE 6. C2050: Effect of thread colouring and L1 cache.

Loop	Original time (S)	Colouring OFF (% diff)	L1 OFF (% diff)
`save_soln`	0.80	n/a	−6.47
`adt_calc`	3.60	1.37	−4.87
`res_calc`	13.14	23.52	−1.91
`bres_calc`	0.12	2.41	−3.16
`update`	4.41	n/a	−2.97
Total	22.07	14.43	−2.78

TABLE 7. C2050: Optimized direct and indirect loops.

Loop	Original time (S)	L1 ON (% diff)	L1 OFF (% diff)
`save_soln`	0.80	26.29	26.22
`adt_calc`	3.60	3.28	−1.15
`res_calc`	13.14	12.10	11.71
`bres_calc`	0.12	13.98	15.14
`update`	4.41	15.44	15.55
Total	22.07	11.85	10.92

7.2. The L1 cache on the C2050

The inclusion of an L1 cache per SM is an important addition to the NVIDIA GPU architecture. Column 4 of Table 6 presents the performance difference of the runtime when L1 is switched off (negative values represents worse performance). The performance effects appear to be minor, due to the fact that the current OP2 library utilize the shared memory block per SM during the execution of indirect loops such as `res_calc`. For the solution of production grade codes such as Hydra on GPUs the number of registers per thread required will significantly increase causing register spillage into global memory. Thus we believe that in the long term the L1 cache is probably best left unused so that the memory can instead be used for register spillage in place of the slower global memory. This is a particular feature of the Fermi GPU and the above results reveal that the current OP2 implementation is already well optimized to handle such a scenario.

7.3. Improving indirect and direct loop execution

In this final section, we implement two optimizations to improve the performance of the execution of direct and indirect loops and explore their performance. For direct loops, OP2 does not need to handle dependency issues and set data used in them are currently fetched directly from global memory of the GPU. Thus utilizing shared memory for direct loops could provide some performance gains, particularly when the L1 cache is switched off. This forms our first optimization.

On the other hand, recall that the execution of indirect loops on the GPU is done by firstly loading indirect data from GPU global memory into an SM's shared memory. The second optimization looks into the benefits of using a different thread numbering scheme to achieve improved coalescence for fetching data from GPU global memory in to shared memory.

Figure 9 illustrates the current and optimized thread numbering schemes in relation to the execution of `res_calc`. The *global* data array consists of the indirectly accessed data that is used in the computation within `res_calc` where—as mentioned in Section 4.2—each element consists of four floating-point values representing the pressure (P), velocities in the two dimensions (u, v) and density (d). The data used within a particular mini-partition of the unstructured mesh are scattered across the GPU global memory as illustrated in the figure. The current implementation utilizes one thread per element to access the data. Thus thread 0 will access all four values, P, u, v, d of the first element, thread 1 will access all four values, P, u, v, d of the next element and so on. Such an access pattern causes a performance loss due to strided memory fetches by each thread. The optimized version will use a different thread (up to the maximum number of threads in the thread block with wrap around) per floating-point value to fetch data. Now, each element is copied as a single coalesced memory access in to the shared memory, followed by a call to `__synchthreads` for synchronizing the threads.

Table 7 details the combined performance gains of both the earlier-mentioned optimizations. The percentage improvement over the original implementation is reported both with and without the L1 cache. We see over 25 and 15% performance gain for the direct loops `save_soln` and `update`, respectively, while the indirect loops `res_calc` and `bres_calc` shows about 11 and 15% gains, respectively. The effect of the L1 cache remains negligible.

We believe that further improvements could be made to indirect and direct loops. Future work will look into the use of the `float4` data type to load four floating-point values directly to registers within an SM bypassing the shared memory and removing the `__synchthreads` statement to eliminate thread synchronization overheads. Padding the shared memory to avoid memory bank conflicts when a CUDA warp accesses shared memory will also be investigated.

8. CONCLUSIONS

We have presented an early performance analysis of the OP2 'active' library, which provides an abstraction framework for the solution of unstructured mesh applications. OP2 aims to decouple the scientific specification of the application from its parallel implementation to achieve code longevity and

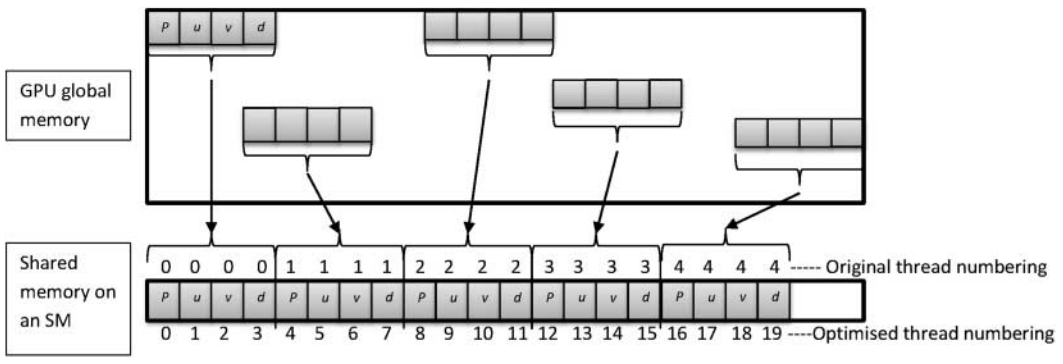

GPU global memory
Shared memory on an SM
P u v d
0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3 4 4 4 4 ----- Original thread numbering
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ----Optimised thread numbering

FIGURE 9. Indirect loop optimization.

near-optimal performance through re-targeting the back-end to different hardware. OP2's code transformation framework was used to generate back-end code for a significant CFD application, targeting multi-threaded executables based on OpenMP and NVIDIA CUDA. The performance of this code was benchmarked during its solution of a mesh consisting of 1.5 million edges on Intel multi-core/multi-threaded CPUs (Penryn and Nehalem) and NVIDIA GPUs (GTX260 and Tesla C2050).

Performance results show that for this application the Tesla C2050 performs about 3.5× and 2.5× better in single precision and double precision arithmetic, respectively, compared with two high-end Intel quad-core processors executing 16 OpenMP threads. These results suggest competitive performance by the GPUs for this class of applications at a production level, but we have also highlighted key concerns, such as memory bandwidth limitations on multi-core/many core architectures at increasing scale, which can limit the achievable performance.

Finally, we presented performance comparisons for a number of low-level hardware and software re-configurations and optimizations on the GPU architecture. Our efforts demonstrate the skilled techniques needed to maintain performance on an emerging architecture such as GPUs and point to the significant benefits of utilizing the OP2 expert framework which does this for novice programmers developing unstructured mesh applications.

The full OP2 source and the Airfoil test case code are available as open source software [6] and the developers would welcome new participants in the OP2 project.

ACKNOWLEDGEMENTS

This research used HPC resources at the Oxford Supercomputing Centre (OSC). We are thankful to Leigh Lapworth, Yoon Ho and David Radford at Rolls-Royce, Carlo Bertolli, David Ham and Florian Rathgeber, at Imperial College London, Jamil Appa and Pierre Moinier at BAE Systems, Tom Bradley at NVIDIA and Nick Hills at the University of Surrey for their contributions to the OP2 project.

FUNDING

Funding has come from the UK Technology Strategy Board and Rolls-Royce plc. through the Siloet project, the UK Engineering and Physical Sciences Research Council projects EP/I006079/1, EP/I00677X/1 on Multi-layered Abstractions for PDEs and the Natural Environment Research Council project NE/G523512/1.

REFERENCES

[1] Crumpton, P.I. and Giles, M.B. (1996) Multigrid Aircraft Computations Using the OPlus Parallel Library. In Ecer, A., Periaux, J., Satofuka, N. and Taylor, S. (eds), *Parallel Computational Fluid Dynamics: Implementations and Results Using Parallel Computers*, pp. 339–346. Elsevier, Amsterdam.

[2] Burgess, D.A., Crumpton, P.I. and Giles, M.B. (1994) A Parallel Framework for Unstructured Grid Solvers. In Wagner, S., Hirschel, E., Periaux, J. and Piva, R. (eds), *Proceedings of the Second European Computational Fluid Dynamics Conference*, Stuttgart, Germany, September, pp. 391–396. John Wiley and Sons.

[3] Campobasso, M.S. and Giles, M.B. (2003) Effect of flow instabilities on the linear analysis of turbomachinery aeroelasticity. *AIAA J. Propuls. Power*, **19**, 250–259.

[4] Moinier, P., Muller, J.D. and Giles, M.B. (2002) Edge-based multigrid and preconditioning for hybrid grids. *AIAA J.*, **40**, 1954–1960.

[5] Giles, M.B., Duta, M.C., Muller, J.D. and Pierce, N.A. (2003) Algorithm developments for discrete ad-joint methods. *AIAA J.*, **42**, 198–205.

[6] Giles, M.B. (2011) *OP2 for many-core platforms*. http://people.maths.ox.ac.uk/gilesm/op2/.

[7] Giles, M.B., Ghate, D. and Duta, M.C. (2008) Using automatic differentiation for adjoint CFD code development. *Comput. Fluid Dyn. J.*, **16**, 434–443.

[8] Howes, L.W., Lokhmotov, A., Donaldson, A.F. and Kelly, P.H.J. (2009) Deriving Efficient Data Movement from Decoupled Access/execute Specifications. In Seznec, A., Emer, J., O'Boyle, M., Martonosi, M. and Ungerer, T. (eds), *High Performance Embedded Architectures and Compilers*,

pp. 168–182. Lecture Notes in Computer Science 5409. Springer, Berlin/Heidelberg.

[9] Chafi, H., DeVito, Z., Moors, A., Rompf, T., Sujeeth, A.K., Hanrahan, P., Odersky, M. and Olukotun, K. (2010) Language Virtualization for Heterogeneous Parallel Computing. *Proc. ACM Int. Conf. Object Oriented Programming Systems Languages and Applications*, Reno/Tahoe, Nevada, USA, October 17–21, pp. 835–847. OOPSLA '10. ACM, New York, NY, USA.

[10] HMPP workbench. http://www.caps-entreprise.com/.

[11] Scala - general purpose programming language. http://www.scala-lang.org/.

[12] DeVito, Z., Joubert, N., Medina, M., Barrientos, M., Oakley, S., Alonso, J., Darve, E., Ham, F. and Hanrahan, P. (October 2010) Liszt: Programming mesh based pdes on heterogeneous parallel platforms. Presentation given by the Stanford PSAAP Center, http://psaap.stanford.edu.

[13] Corrigan, A., Camelli, F.F., Löhner, R. and Wallin, J. (2009) Running Unstructured Grid-based CFD Solvers on Modern Graphics Hardware. *19th AIAA Computational Fluid Dynamics Conference*, Grand Hyatt Hotel, San Antonio, Texas, June 22–25, pp. 1–11. Curran Associates, Inc., NY, USA.

[14] Asouti, V.G., Trompoukis, X.S., Kampolis, I.C. and Giannakoglou, K.C. (2011) Unsteady CFD computations using vertex-centered finite volumes for unstructured grids on graphics processing units. *Int. J. Numer. Methods Fluids*, **66**.

[15] Giles, M.B. (2010) *OP2 developer's manual*. http://people.maths.ox.ac.uk/gilesm/op2/dev.pdf.

[16] Giles, M.B. *Hydra*. http://people.maths.ox.ac.uk/gilesm/hydra.html.

[17] (2010), *CUDA C Programming guide*. http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf.

[18] *The ROSE Compiler*. http://www.rosecompiler.org/.

[19] Intel Xeon Processor E5540 specifications. http://ark.intel.com/Product.aspx?id=37104.

[20] NVIDIA Tesla C2050/C2070 GPU Computing Processor. http://www.nvidia.com/object/product_tesla_C2050_C2070_us.html.