Performance Analysis of the OP2 Framework on Many-core Architectures

M.B. Giles, G.R. Mudalige
Oxford e-Research Centre
University of Oxford
mike.giles@maths.ox.ac.uk,
gihan.mudalige@oerc.ox.ac.uk

Z. Sharif, G. Markall, P.H.J Kelly,
Dept. of Computing
Imperial College London
{zohirul.sharif09, graham.markall08,
p.kelly}@imperial.ac.uk

ABSTRACT

This paper presents a performance analysis and benchmarking study of the OP2 "active" library, which provides an abstraction framework for the solution of parallel 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 back-end to different hardware.

Runtime performance results are presented for a representative unstructured mesh application written using OP2 on a variety of many-core processor systems, including the 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 on 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.

Categories and Subject Descriptors

D.4.8 [**Performance**]; C.1.2 [**Multiple Data Stream Architectures**]

Keywords

CFD, Performance, GPU, OpenMP, CUDA, OP2

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 organisations. However, the future of such a programming model's use in an environment of increasingly complex hardware architecture is unsustainable. This problem is becoming even more compounded as the High Performance Computing (HPC) industry begins to focus on delivering exa-scale 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 them from exploring the capacity of available hardware. It is therefore clear that a level of abstraction must be achieved so that computational scientists can reach an increased level of 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 (1) a higher application level where scientists and engineers focus on solving domain specific problems and write efficient code that remains unchanged for different underlying hardware architectures and (2) a lower implementation level, that focus 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) [10], 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 [6]. To this day it is used as the underlying parallelization library for Hydra [12, 7, 20, 16] a production-grade CFD application used in turbomachinary 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 program transformation tools, so that the user code is transformed into the appropriate form so that it can be linked against the required parallel implementation (e.g. MPI, OpenMP, CUDA, OpenCL, AVX etc.) enabling execution on different target back-end hardware platforms [14]. 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 an early 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 [17]) written using OP2 on a range of systems. These consist of representative currently prevalent 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:

1. 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.

2. The multi-core, multi-threaded CPU performance of Airfoil is compared against equivalent GPU solutions executing back-end code generated by OP2 based on NVIDIA CUDA. Performance on a number of CUDA capable GPUs are presented, including a GTX260 consumer card and a Tesla C2050 based on the new Fermi architecture - NVIDIA's current flagship GPU offering.

3. Our analysis demonstrates the pros and cons between the use of CPU and GPU architectures and related parallel implementations for the Airfoil computation. These include the significance of choosing the correct partition and thread-block configuration, the factors limiting GPU performance and insights into optimizations for near optimal performance.

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 parallelising unstructured mesh applications; Section 5 and Section 6 presents performance figures for the execution of Airfoil on CPU and GPU systems respectively, including comparisons between the two architectures. Finally Section 7 concludes the paper.

2. RELATED WORK

Although OPlus predates it, OPlus and OP2 can be viewed as an instantiation of the AEcute (access-execute descriptor) [18] 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 the Hybrid Multi-core Parallel Programming (HMPP) [1] workbench and LISZT [8].

HMPP allows the user to annotate codelets with HMPP directives. The program is then processed through the toolchain 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.

LISZT is a domain specific language developed with the goal of leveraging domain specific optimizations. Domain specific languages, in contrast to general purpose languages, can infer a large amount of information about the structure of data and/or the nature of the algorithms in the code. Thus aggressive and platform specific optimizations can be applied.

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 LISTZ framework have been presented in [11]. The authors reports the performance of Joe, a fluid flow unstructured mesh application solving a mesh of 750K cells. Joe is first ported to the LISTZ framework and the resulting code compared to the original code running on a cluster of 4-socket 6-core 2.66GHz Xeon CPUs each with 36GB RAM per node using MPI. Both codes demonstrates equivalent performance (LISTZ runtime in fact faster with better scalability) illustrating that no performance loss has resulted due to the use of the LISTZ framework. Speed-up figures for the above code running on a Tesla C2050 (implemented using CUDA) against an Intel Core 2 Quad, 2.66GHz processor is provided next, with results showing a speed-up of about 30× in single precision arithmetic and 28× in double precision.

Related work in the solution of unstructured mesh applications on GPUs, particularly in the CFD domain have also appeared elsewhere. In [9], 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× is observed during the execution of the GPU implementation on an NVIDIA Tesla 10 series card against an equivalent optimized OpenMP implementation on an Intel Core 2 Q9450 running 4 threads.

Similarly in [5, 19], GPU performance of a Navier-Stokes solver for steady and unsteady turbulent flows on unstructured/hybrid grids are detailed. The computations were carried out on NVIDIA's GeForce GTX 285 graphics cards (in double precision arithmetic) and speed-ups up to 46× (vs two Quad Core Intel Xeon CPUs at 2.00 GHz) are reported.

It is unfortunate that, most of the performance analysis results from the above works are compared against either a single thread on a CPU [11], or older generation CPUs [9, 5, 19]. Although there are significant contributions made in these works for achieving implementations capable of utilising emerging many-core hardware, we believe that performance comparisons should compare the best available CPU solution to that of the best GPU solution. In this paper our comparison is between systems that represent the current state-of-the art in both the GPU and CPU systems for HPC. In this case we use up to 16 OpenMP threads on a node consisting of two Intel Xeon E5540 quad-core processors (based on Intel's current flagship Nehalem micro-architecture) to be compared against a C2050 GPU based on NVIDIA's state-of-the-art Fermi architecture.

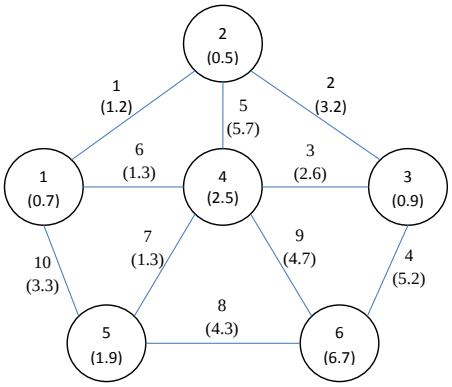


Figure 1: An example mesh

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 electro magnetics); structural mechanics; and general finite element methods. In three dimensions often millions of elements are needed for the required solution accuracy, consequently, a large computational expense is incurred.

The OPlus approach to the solution of such unstructured mesh problems involves breaking down the unstructured grid algorithms into four distinct parts: (1) sets, (2) data on sets, (3) mappings between sets and (4) operations over sets regardless of the application. These lead to an API through which one can completely and abstractly define any mesh or graph.

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 which 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 Figure 1 is defined as two sets, nodes (vertices) and edges, each with their sizes, using the API as follows:

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

The connectivity is declared through mappings (or pointers) between the sets. The two integer type arrays `pEdge1` and `pEdge2` can be used to represent how an edge is connected to two different vertices.

```
int pEdge1[10] = {1, 2, 3, 3, 2, 1, 4, 6, 4, 1};
int pEdge2[10] = {2, 3, 4, 6, 4, 4, 5, 5, 6, 5};

op_ptr pointer_edge1;
op_decl_ptr(set_edges, set_nodes, 1, pEdge1,
          pointer_edge1, "pointer_edge1");
op_ptr pointer_edge2;
op_decl_ptr(set_edges, set_nodes, 1, pEdge2,
          pointer_edge2, "pointer_edge2");
```

Each element of `set_edges` is mapped to two different elements in `set_nodes`. In our example, index 0 of `pEdge1` and `pEdge2` maps an edge to its vertices 1 and 2 respectively. Thus the `pEdge1` and `pEdge2` arrays of indices define the connectivity between the two sets. When declaring a mapping we first pass the source (e.g. `set_edge`) then the destination (e.g. `set_node`). We then pass the dimension of each element (e.g. 1; as each edge is mapped to 1 node). 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(set_edges, 1, "float",
            dEdge, data_edges, "data_edges");

op_dat data_nodes;
op_decl_set(set_nodes, 1, "float",
            dNode, data_nodes, "data_nodes");

op_dat data_nodes_u;
op_decl_dat(set_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, performing some arithmetic, then writing (possibly through the mappings) back to data arrays. The OP2 library provides a parallel loop declaration syntax which allows the user to declare the computation over the sets in these loops [15]. For the mesh illustrated in Figure 1 a loop over all the edges which updates the nodes can be written in the following sequential execution loop:

```
void kernel(int nedge, int *pEdge1, int *pEdge2,
                       float *dEdge, float *dNode) {
for (int e=0; e<nedge; e++)
  dNode_u[pEdge1[e]] += dEdge[e] * dNode[pEdge2[e]];
}
```

The user declares this loop using the API as follows, leaving the library to handle the parallelization of the loop on a particular architecture.

```
op_par_loop_3(kernel,"kernel", set_edges,
              dEdge,  -1,OP_ID,  1,"double",OP_READ,
              dNode,   0,pEdge2, 1,"float", OP_READ,
              dNode_u, 0,pEdge1, 1,"float", OP_INC);
```

This general decomposition of unstructured algorithms imposes no restriction on the actual algorithms, just separates the components of a code. However OP2 makes an

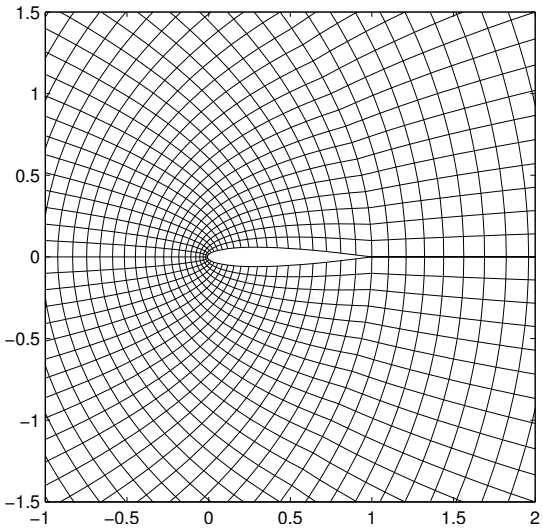


Figure 2: Rendering of a simple 120x60 mesh used in Airfoil

important restriction that the order in which the elements are processed must be independent of the final result, that is the order in which we process the elements should not affect the results. 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 in a third set).

Although it might appear that these restrictions are quite severe, the straightforward programming interface, 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 which use explicit smoothers; Krylov subspace methods with explicit preconditioning; semi-implicit methods were 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), falls beyond the capabilities of the API.

The example application used in our analysis, Airfoil, is a non-linear 2D inviscid airfoil code that uses an unstructured grid [17]. It is a much simpler application than the production Hydra [12] CFD application used at Rolls Royce plc., for the simulation of turbomachinary design, 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 Figure 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 720K nodes, 720K cells and about 1.5 million edges. The code consists of five parallel loops: `save_soln` , `adt_calc`, `res_calc`, `bres_calc`, `update` where on 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.

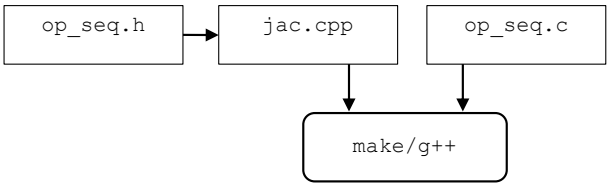


Figure 3: Sequential build process

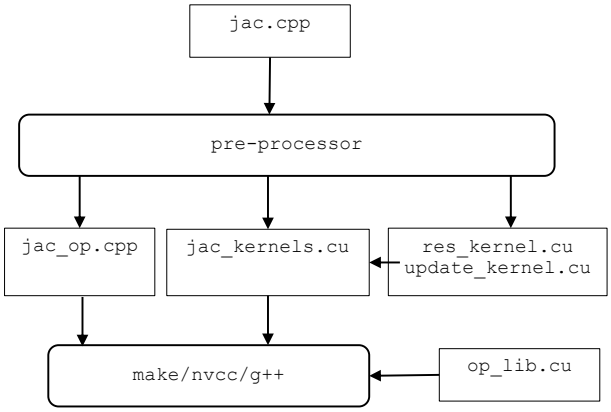


Figure 4: CUDA build process

4. OP2

The original OPlus library [10], was developed over 10 year 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 and OpenMP and will later include code generation for OpenCL and AVX, thus supporting a wide range of CPU and GPU hardware. OP2 will also including support for the above to be executed on distributed memory CPU and GPU clusters in conjunction with MPI. 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.

Figure 3 and Figure 4 illustrates and contrasts 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

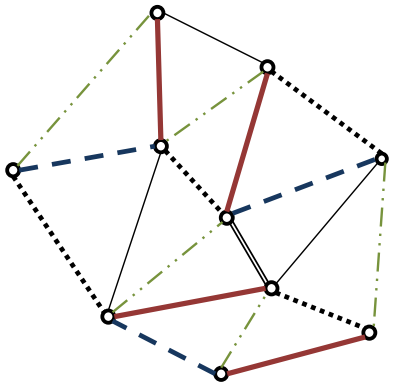


Figure 5: An example coloring of edges

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 pre-processor is currently implemented as a source to source translator using the ROSE compiler framework [4].

4.1 Parallelising Unstructured Mesh Applications

A key technical difficulty of solving 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 do 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 "color" for the edges so that no two edges of the same color update the same node. This allows for parallel execution for each color 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 *atomics* which combines read/add/write into a single operation. However, this requires hardware support which may or may not be available and may have slow implementations.

The OP2 design attempts to resolve the data dependency problems using the above three methods, at three levels of parallelism. Using method 1, given a global mesh with sets and data, in the future 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 execute on their own elements. However, it is possible that one partition may need to access data which belongs to another partition; in that case a copy of the required data is provided by the other partition. This follows the standard "halo" exchange mechanism used in distributed memory message passing parallel implementations. As the partition sise 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.

Table 1: CPU node system specifications

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

On the GPU, updating the same node could occur either (1) by multiple threads executed by a single processing unit (an SM) updating data held in its shared memory (i.e. a mini-block) or (2) when the result of the shared memory are written back to the main graphics memory which is used by other processing units. In OP2 thread coloring is used for the former and a block coloring is used for the latter.

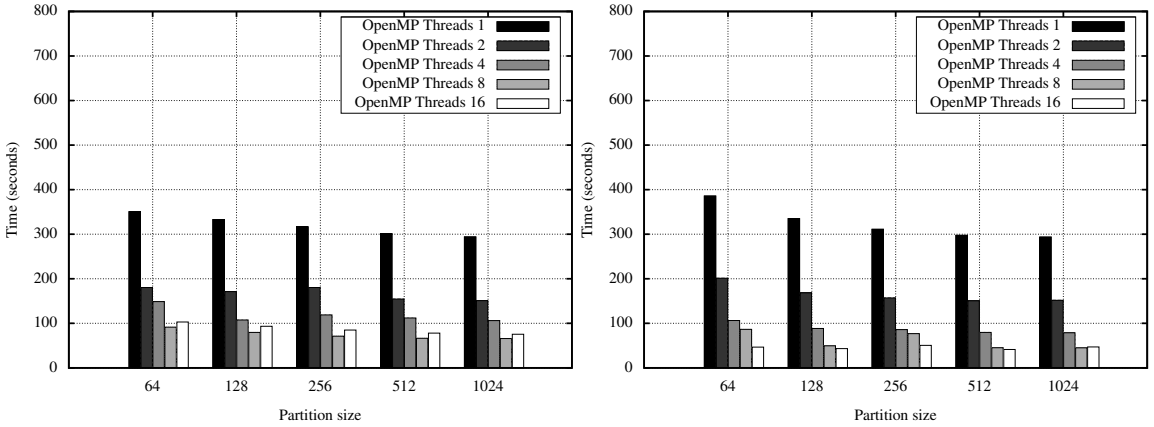
Edges are colored so that two edges with the same color never update the same node (see Figure 5). As a result, the edges with the same color can be processed in parallel by different threads. The coloring is performed very efficiently in a runtime initialization using a bitwise operation on a 32 bit integer for each of the edges [13]. Similarly, a block coloring scheme is used so that results from shared memory, after processing a mini-block, are not used by any other mini-block being processed simultaneously. On a production grade CFD application such as Hydra a single run would consist of over 100K blocks each needing to fit into the shared-memory of a GPU. 10 colors might be needed to avoid data conflicts, suggesting up to 10K blocks per color.

A similar technique is used for multi-core processors. The difference is that each mini-block is executed by a single OpenMP thread. The mini-blocks are colored 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 main memory to shared memory transfer) and no need for low level thread coloring.

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 of multi core, multi-threaded CPUs. This section presents the results for single precision performance on the CPUs. Double precision performance is detailed 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 consist of two Intel Xeon E5462 quad-core (total of 8 cores) processors operating at 2.8GHz clock rate per core and has access to 16GB 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 2.5GHz per core clock rate and access to 25GB 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.

For our experiments we mainly use the Intel ICC 11.1 compiler for generating OpenMP enabled executables on the above two systems. For performance comparisons we also use the GNU C++ compiler version 4.4.3. For ICC we use



(a) Intel Xeon E5462 (Penryn) (b) Intel Xeon E5540 (Nehalem)

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)

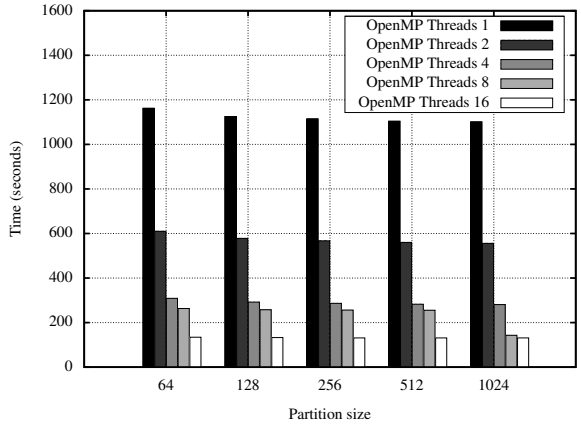


Figure 7: Runtime of Airfoil on Intel Xeon E5540 (Nehalem), Compiled with GCC 4.4.3, with up to 8 OpenMP threads - Single Precision (1000 Iterations)

`-O3 -fast -parallel` compiler flags and for `gcc` we compile using `-O2 -msse3 -funroll-loops`.

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-block. Coloring is used to stop multiple mini-blocks interfering with the same data. Thus, a key parameter in our study will be to investigate the mini-block 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 on upto 16 OpenMP threads. 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. The best runtime is given by a partition size of 1024 using 8 OpenMP threads on both processors. On the Nehalem the 16 thread run uses SMT and provides as expected a much smaller performance improvement than the improvement gained by increasing the thread count from 4 to 8 for all partition sizes except 1024. Regardless of the partition size increasing the number of threads from 1 to 8 provides diminishing returns. Thus 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 the execution of Airfoil. Given that the maximum available bandwidth of these processors are 25.6GB/s [2] the code appears to be saturating the processor's bandwidth capacity. Thus we suspect that the memory bandwidth of the single node system may become the bottleneck limiting future thread scalability.

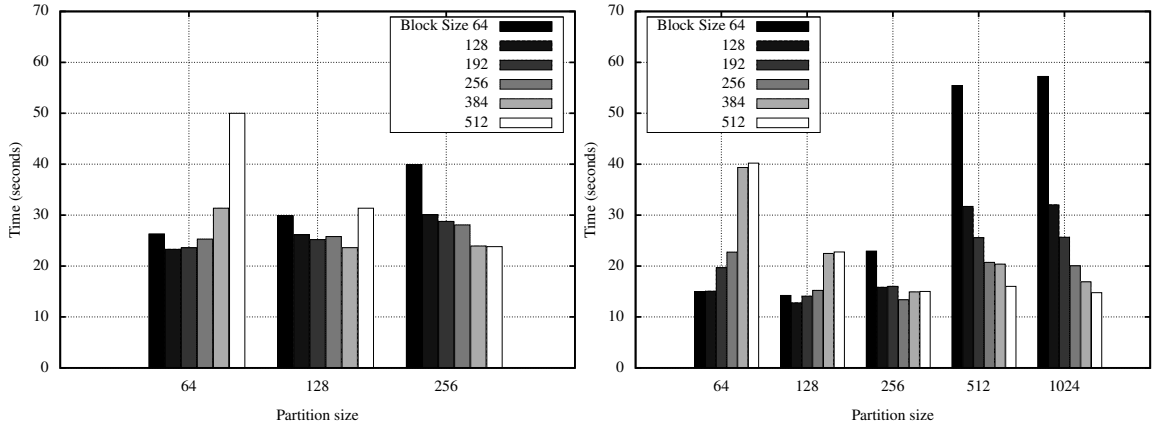
Table 2: GPU node system specifications

GPU	Cores	Clock GHz	Global Mem	Shared Mem /SM	Driver /Comp. Cap.
GeForce GTX260	216	1.4	0.8GB	16kB	3.2/1.3
Tesla C2050	448	1.15	3.0GB	48kB	3.2/2.0

As a performance comparison we also compiled the Airfoil code using gcc 4.4.3 on the Nehalem processor based node. Runtime results are detailed in Figure 7. There is a significant difference between the performance of the two executables produced by the ICC and gcc compilers. The code compiled using the Intel compiler perform roughly three times better than the gcc executable on the best partition size configuration. Thus, these results demonstrate the importance of comparing performance figures with the best available compiler and hardware, particularly when making decisions about switching to a new target platform (e.g. GPUs or multi-core CPUs) for production scientific workloads.

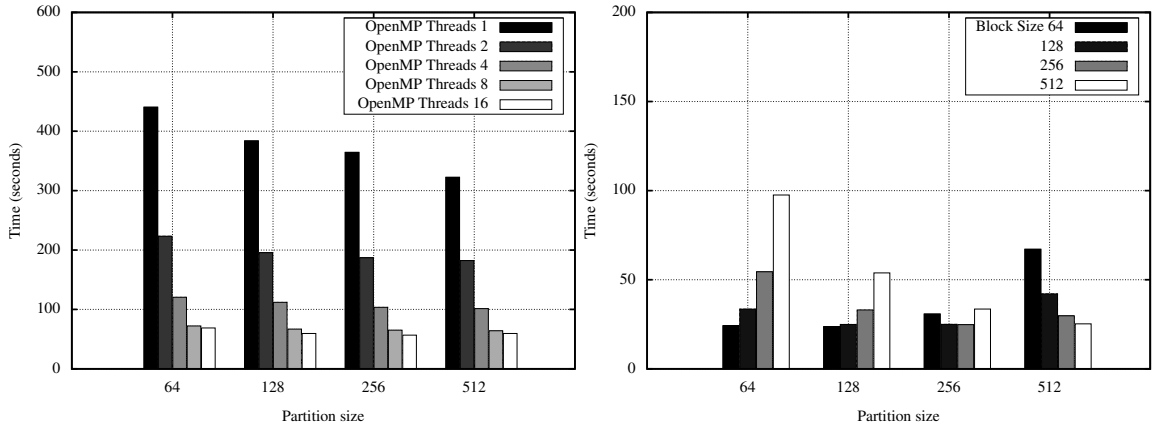
6. GPU PERFORMANCE

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 to be executed on the GPUs. Table 2 details the specifications of each system. The host CPUs used in the GTX260 is an AMD Athlon X2 dual core processor at 2GHz, while for the Tesla the host is a quad-core Intel Xeon E5530 processor operating at 2.4GHz. In both GPU systems NVIDIA's CUDA/C compiler `nvcc` was built using the GNU C compiler 4.4.5.



(a) GTX260 (b) Tesla C2050

Figure 8: 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 9: Runtime of Airfoil on Intel Xeon E5540 (Nehalem) and NVIDIA Tesla C2050 - Double Precision (1000 Iterations)

Figure 8 presents the total runtime of Airfoil (executing single precision mathematics) on the two NVIDIA cards. For these runs the number of CUDA threads allocated per mini-block provides an additional configuration parameter. The GTX260 could only execute partition sizes up to 256 due to its limited memory. The GTX260 performs only about two times slower than the Tesla C2050 due to their comparable single precision floating-point performance. The best performance- just over 12 seconds - 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 under 4× compared to the Intel Nehalem processor systems performance on 8 OpenMP threads with a partition size of 1024.

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 approximately 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. This translates to 15 GFlops per second on the Intel Nehalem processor based system and about 30 GFlops per second on the Tesla C2050. Thus we see that only a fraction of the peak single-precision floating-point performance on the GPU is achieved.

A key concern in determining whether GPUs are suitable for main-stream HPC and production scientific work is how its performance compares against the traditional processors when executing double precision floating-point codes. Because 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 final benchmarking study for us therefore is to investigate the Airfoil execution in double-precision. Figure 9 details the double precision performance of Airfoil on the Intel Nehalem and Tesla C2050. The results demonstrate a speed-up of approximately 2.5× on the Tesla C2050 compared to the best runtime on the Intel Nehalem. The observed memory bandwidth utilization in this case is close to 90 GB/s for some of the routines. This is a utilization of over 60% of the available memory bandwidth (144 GB/sec) on the Tesla C2050 [3].

Given that each element on the unstructured mesh could be computed independently, it is not surprising that the GPU architecture out-performs the traditional multi-core processors. But it is interesting that only about a factor of 4× speed-up is achieved in single precision and only about 2.5× speed-up gained in double precision. We believe that

several factors might be responsible for this. The use of integer pointer arithmetic in computing indirect references in unstructured mesh computations increases the GPU execution time as there is no separate integer pipeline on these simple cores, unlike mainstream CPUs. Thus an integer operation costs as much as a floating-point operation (at least in single precision). Similarly, memory bandwidth limitations will be hindering performance as all routines in the Airfoil code have at least 30% bandwidth utilization and some even close to the available upper limit.

7. 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 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 4× and 2.5× better in single precision and double precision mathematics respectively compared to two high end Intel multi-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.

The full OP2 source and the Airfoil testcase code are available as open source software [14] and the developers would welcome new participants in the OP2 project.

Acknowledgements

This research used HPC resources at the Oxford Supercomputing Centre (OSC).

8. REFERENCES

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

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

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

[4] The ROSE Compiler. `http://www.rosecompiler.org/`.

[5] V. G. Asouti, X. S. Trompoukis, I. C. Kampolis, and K. C. Giannakoglou. Unsteady CFD Computations Using Vertex-Centered Finite Volumes for Unstructured Grids on Graphics Processing Units. *International Journal for Numerical Methods in Fluid*, to appear 2010.

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

[7] M. S. Campobasso and M. B. Giles. Effect of flow instabilities on the linear analysis of turbomachinery aeroelasticity. *AIAA Journal of Propulsion and Power*, 19(2):250–259, 2003.

[8] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language Virtualization for Heterogeneous Parallel Computing. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 835–847, New York, NY, USA, 2010. ACM.

[9] A. Corrigan, F. F. Camelli, R. Löhner, and J. Wallin. Running Unstructured Grid-based CFD Solvers on Modern Graphics Hardware. *International Journal for Numerical Methods in Fluid*, 2010.

[10] 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*. 339-346, A. Ecer, J. Periaux, N. Satofuka, and S. Taylor, editors, North-Holland, 1996.

[11] Z. DeVito, N. Joubert, M. Medina, M. Barrientos, S. Oakley, J. Alonso, E. Darve, F. Ham, and P. Hanrahan. Liszt: Programming Mesh Based PDEs on Heterogeneous Parallel Platforms. Presentation given by the Stanford PSAAP Center, Oct 2010 `http://psaap.stanford.edu`.

[12] M. Giles. Hydra. `http://people.maths.ox.ac.uk/gilesm/hydra.html`.

[13] M. Giles. OPlus2 Developer's Manual. `http://people.maths.ox.ac.uk/gilesm/op2/dev.pdf`.

[14] M. Giles. OPlus2 for Many-Core Platforms. `http://people.maths.ox.ac.uk/gilesm/op2/`.

[15] M. Giles. OPlus2 User Manual. `http://people.maths.ox.ac.uk/gilesm/op2/user.pdf`.

[16] M. B. Giles, M. C. Duta, J. D. Muller, and N. Pierce. Algorithm developments for discrete ad-joint methods. *AIAA Journal*, 42(2):198–205, 2003.

[17] M. B. Giles, D. Ghate, and M. C. Duta. Using Automatic Differentiation for Adjoint CFD Code Development. *Computational Fluid Dynamics Journal*, 16(4):434–443, 2008.

[18] L. W. Howes, A. Lokhmotov, A. F. Donaldson, and P. H. J. Kelly. Deriving efficient data movement from decoupled access/execute specifications. In A. Seznec, J. Emer, M. O'Boyle, M. Martonosi, and T. Ungerer, editors, *High Performance Embedded Architectures and Compilers*, volume 5409 of *Lecture Notes in Computer Science*, pages 168–182. Springer Berlin/Heidelberg, 2009.

[19] I. C. Kampolis, X. S. Trompoukis, V. G. Asouti, and K. C. Giannakoglou. CFD-based analysis and two-level aerodynamic optimization on graphics processing units. *Computer Methods in Applied Mechanics and Engineering*, 199(9-12):712–722, 2010.

[20] P. Moinier, J. D. Muller, and M. Giles. Edge-based multigrid and preconditioning for hybrid grids. *AIAA Journal*, 40(10):1954–1960, 2002.