

OPlus2 for many-core platforms

Mike Giles

November 1, 2009

Abstract

This document lays out the plans for a new open source OPlus2 library for unstructured grid parallel execution on NVIDIA GPUs and other many-core hardware platforms.

This document will be all-encompassing to begin with. Later, it will make sense to split it into different documents (e.g. Users Guide; Developers Guide; Algorithms for partitioning, colouring and halo construction) but for now I think it is simplest to have it all together in one place.

1 Motivation

The original OPlus library was developed more than 10 years ago for distributed memory parallel execution of algorithms based on unstructured grids. It was originally developed in PVM by Paul Crumpton and myself, but more recently Nick Hills ported it to MPI and made numerous internal changes (such as parallel partitioning) which greatly improved its performance.

The strengths and weaknesses of the existing OPlus are discussed below, but the main motivation for a new version is the emergence of multicore architectures. Not only are Intel, AMD, and others all pursuing a multicore CPU strategy, with up to 6 cores today but maybe 16 cores (and up to 64 threads?) within a few years, but also NVIDIA and ATI (now part of AMD) have graphics cards with huge numbers of cores. The current NVIDIA GTX280 GPU, for example, has 240 cores, each of which is as capable as an Intel Core 2 Duo core in single precision.

The objective of OPlus2 is to provide a library which will enable efficient execution on these new architectures at the lowest level, while retaining the ability to perform distributed memory parallelisation on the highest level.

2 OPlus

After more than 10 years of use, it is a good time to review the strengths and weaknesses of the existing OPlus library.

Strengths:

- general distributed set/pointer representation has worked well
- single application source code leading to both sequential and parallel executables
- fairly clean definition of parallel loops
- good scalability on big problems with up to 10^8 grid nodes on 100+ processors (partly due to the original design but largely due to Nick Hills' improvements)
- works well under both Unix/Linux and Windows operating systems

Weaknesses:

- file I/O is poor, not parallel
- `OP_ACCESS` declaration of data dependencies in parallel loops is tedious, and errors can be hard to debug
- only FORTRAN is supported, and it is only used by the HYDRA CFD code

The natural goal with OPlus2 is to retain the strengths, address the weaknesses, and add the capability to efficiently exploit new many-core architectures.

References

- 1) P.I. Crumpton and M.B. Giles. 'Multigrid aircraft computations using the OPlus parallel library'. in *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. [\(PDF\)](#)
- 2) D.A. Burgess, P.I. Crumpton, and M.B. Giles. 'A parallel framework for unstructured grid solvers'. In *Computational Fluid Dynamics '94: Proceedings of the Second European Computational Fluid Dynamics Conference*, pages 391-396. S. Wagner, E.H. Hirschel, J. Periaux, and R. Piva, editors. John Wiley and Sons, 1994. [\(PDF\)](#)

3 Key assumptions and requirements

The key assumptions are the same as for OPlus: that the application involves a number of static sets (e.g. nodes, edges, faces, cells) connected by pointers, and each step in the algorithm involves an operation applied to all elements of a set (with at most one level of pointer indirection to data associated with another set) and the result is independent of the order in which the elements are processed (e.g. Jacobi iteration is allowed but not the standard Gauss-Seidel).

The requirements are:

- single application source code leading to efficient execution on
 - multicore CPUs using OpenMP and/or SSE/AVX instructions
 - GPUs using CUDA or OpenCL
- very good scalability up to very large problem sizes using MPI to exploit multiple GPUs and/or multicore CPUs within a cluster
- support for both C/C++ and FORTRAN
- support for Unix/Linux and Windows operating systems
- options for extensive error-checking
- compatibility with HYDRA’s use of automatic differentiation for generating linear and adjoint code

With OPlus, the user code was compiled and then linked to either a sequential single-thread library or a parallel library (based on MPI message passing). There was no use of program transformation, partly because it wasn’t needed and partly because we had no knowledge of such tools.

With OPlus2, the efficient support of multicore CPUs as well as GPUs will require the use of program transformation tools, with the expertise being supplied by Paul Kelly’s group at Imperial College. However, I think it is still desirable to have a non-efficient single thread CPU implementation which does not require program transformation; this can be used for application development and debugging and should include the extensive error-checking tools mentioned above.

4 Outline workplan

The development of OPlus2 will proceed in two phases, with perhaps some overlap between the two.

- Phase 1 will develop OPlus2 for use on a single GPU or a single shared-memory system with one or more multicore CPUs (using OpenMP and SSE/AVX instructions).
- Phase 2 will extend this by adding on top an MPI layer which is very similar to the existing OPlus implementation. This will enable distributed-memory computations on a cluster of nodes, each with one or more GPUs and/or multi-core CPUs.

The partitioning and parallelisation strategies required by the two phases will be similar in some ways, but will also be different because of differences in key hardware aspects. For example, typical cluster compute nodes have GBs of shared memory, whereas each “multiprocessor” in an NVIDIA GPU currently has only 16kB of shared memory. As explained later, this leads to a different approach in handling data dependency constraints.

The phase 1 workplan is as follows (IC = Imperial College post-doc):

- single-thread CPU version of test code written in C, including non-efficient CPU library (Mike, Sept 09)
- CUDA version of test code written in C (Mike, Oct/Nov 09)
- document describing program transformation requirements for C \rightarrow CUDA (Mike/Paul, Dec 09)
- single-thread CPU version of test code written in FORTRAN, including non-efficient CPU library (Mike, Jan 10)
- C and FORTRAN versions of 2D airfoil test code (Mike, Mar 10)
- program transformation for C \rightarrow CUDA (IC, June 10)
- OpenMP/SSE version of original test code (Mike, June 10)
- program transformation for FORTRAN \rightarrow CUDA (IC, Sept 10)
- test program transformation tools on 2D airfoil test code (Mike, Sept 10)
- program transformation for C \rightarrow OpenMP/SSE parallel loops (IC, Dec 10)
- OpenCL version of original test code (Mike, Dec 10)

- “lint”-style syntactic analysis and error-checking for parallel loop kernels (IC, Mar 11)
- program transformation tool to aid porting of HYDRA OPlus calls to OPlus2 (IC, June 11)
- begin phase 2 (Mike, July–Sept 11)
- program transformation for C, FORTRAN \rightarrow OpenCL (IC, Sept 11)

At the end of the first phase we will have a viable library and program transformation tools for execution on a single host with one or more NVIDIA GPUs.

The porting of HYDRA will overlap with the final parts of phase 1 and the early parts of phase 2, with the three main steps being:

- convert remaining parallel loops into kernel format (this can be done at any time, independent of work on phase 1);
- convert OPlus parallel loops into OPlus2 parallel loops, aided by the tool developed by Imperial;
- convert file I/O to the new parallel file I/O agreed for phase 2.

The timing for phase 2 is a bit uncertain, but the key steps (some of which can be overlapped) are:

- review existing OPlus algorithms and code (Mike, 3 months?)
- review HDF5 and MPI I/O and design OPlus2 file I/O (Leigh, 3 months?)
- review key implementation issues and concerns (Nick, 3 months?)
- implement new file I/O (1 month?)
- implement phase 2 data partitioning and halo construction (3 months?)
- test on a conventional cluster (1 month?)
- optimise GPU partitioning to overlap MPI communication with GPU computation (1 month?)
- test on a GPU cluster (1 month?)
- test HYDRA on a GPU cluster (1 month?)
- develop performance monitoring tools to guide further development (1 month?)

The work in most of these items will be done jointly by the Imperial College postdoc and myself.

5 Data access and dependency conflicts

Suppose that one wants to solve the sparse system of linear equations

$$A u + r = 0$$

using Jacobi iteration with

$$u^{n+1} = u^n + (A u^n + r).$$

A can be stored in a sparse edge-based format $A, p1, p2$ in which each non-zero element has a corresponding “edge” e with $A[e] \equiv A_{i,j}$, $p1[e] \equiv i$, $p2[e] \equiv j$.

Using this, the key steps in the algorithm can be implemented using the C routines in Figure 1. The access patterns for the arrays can be classified as

- “r” read-only
- “w” write-only
- “b” both read and write
- “i” increment (a special case of read/write)

In `res`, the access for A , u , du is “r”, “r” and “i”, respectively, while in `update`, the access for u , du , r is “i”, “b” and “r”, respectively.

It is clear from looking at these routines that it makes no difference the order in which the edges are processed in `res`, or the nodes are processed in `update`, and so these conform to the requirements for OPlus parallelisation. The one potential

```
void res(int ne, int *p1, int *p2, float *A, float *u, float *du) {
    for (int e=0; e<nedge; e++)
        du[p1[e]] += A[e] * u[p2[e]];
}

void update(int nn, float *u, float *du, float *r) {
    for (int n=0; n<nn; n++) {
        u[n] += du[n] + r[n];
        du[n] = 0.0f;
    }
}
```

Figure 1: C routines for simple Jacobi iteration

problem is a potential data dependency conflict in incrementing the array `du` in `res`. Because of the indirect addressing using `p1`, it is possible that different edges may try to update the same element of `du` “at the same time”. Unless the hardware supports atomic updates for floating point variables, this could lead to incorrect results.

In OPlus, the parallelisation partitions the data so that each partition “owns” some of the edges and some of the nodes. Each partition “executes” the edges and nodes which affect the data it “owns”. Near partition boundaries, this may require one partition to execute an edge belonging to a different partition. In addition, if the execution of the edge requires the use of data belonging to another partition, a copy of that data has to be made; this is referred to as “halo” data in the distributed-computing community.

Within each partition, a single thread sequential execution model is used and so there is problem with the data dependency issue. However, in more complex applications where executing an edge may involve updates to more than one node, there is a redundancy/duplication issue, since the same edge may be executed by more than one partition. This is acceptable provided the partition sizes are large enough so that the number of halo edges and nodes is relatively small.

In OPlus2, the same strategy will be used in phase 2 for distributed computing across multiple GPUs, but in phase 1 the same approach will not be acceptable because the partition sizes to fit into each of the multiprocessors within a single GPU are so small that there will be a significant level of duplication. We will instead use an alternative partition “colouring” strategy which is described in a later section.

6 CUDA parallelisation strategy

The first step is to reorder/renumber all sets to improve data locality, so that neighbouring nodes or edges have indices, and therefore memory locations, which are close to each other. There are lots of ways in which this might be done, but one simple effective way which we have used in the past is recursive coordinate bisection. For a 3D dataset, one would first bisect each set in the x -direction, then the y -direction, then the z -direction, and then repeat the sequence recursively to build up a binary tree whose leaves are then numbered sequentially.

The idea then is to parallelise loop operations by using blocks (mini-partitions) which are small enough to fit into the limited shared memory on each multiprocessor within an NVIDIA GPU. For the edge operations in `res` in Figure 1, the operations for each block would be

- load in required values of `u` into shared memory
- calculate increment
- add increment to shared memory array `inc`
- add increment `inc` to array `du` in main graphics memory

The first step of loading `u` into shared memory is performed because the same values of `u` are likely to be used by several edges, and so doing this minimises the total data transfer requirements from the main graphics memory.

The second step is simple, and can be performed in parallel by all threads in the block (with just one edge per thread?) without any data conflicts. The potential data conflict is in the third step when the increments are combined. This will be avoided by “colouring” the threads so that no two threads update the same variable at the same time. This is a well-established technique for avoiding data conflicts in vector code.

(To minimise the number of colours within each thread “warp”, it may be best to reorder the threads within each block. This would scramble up the identity mapping, used here for the array `A`, so there would be some cost but I think there would be savings overall.)

The fourth step adds the combined increments to the data in the main graphics memory. Here again there is a potential data conflict problem due to different blocks trying to update the same data. This will be avoided by colouring the blocks so that no two blocks of the same colour update the same data. The blocks would be executed one colour at a time, with a synchronisation between each colour.

6.1 Execution plans

In standard MPI computations, the partitioning is done once and the same partitioning is used for stage of the computation. This is because the cost of re-partitioning the data is excessive.

In contrast, between each stage of the computation on the GPU the data resides in the main graphics memory, and so the blocking for each loop calculation can be considered independently of the requirements of the other parallel loops.

Based on ideas from FFTW, I plan to construct for each parallel loop a “plan”, which is a customised blocking of the execution on the GPU for that loop, making optimum use of the local shared memory on each multiprocessor considering in detail the memory requirements of the loop computation.

The plan will also include the colouring of the blocks, and colouring of the threads within a block, to avoid data conflicts. A typical CFD calculation may need 50-200 plans each of which may be executed more than once during a single timestep or multigrid iteration.

6.2 Some rough numbers

A big HYDRA CFD calculation will use almost all of the 4GB on a Tesla card. Given that the local shared-memory size is 16kB, this suggests that the data intensive parallel loops will have over 100,000 blocks. ¹

10 colours might be needed to avoid data conflicts, suggesting up to 10,000 blocks per colour. Given this number it doesn’t matter if there are variations in the amount of compute per block, the important thing is to make sure each block uses as much shared memory as possible.

HYDRA needs up to 40 floats per grid node. 16kB corresponds to 4000 floats which is equivalent to 100 nodes, which is roughly 5^3 . This should be big enough to get a fair degree of re-use of nodal data within the block, maybe 50% of maximum. The block may have 400 edges, with maybe 40 per colour so we need to use colour-locked increments, with thread synchronisation in between colours.

¹On the one hand, this will be an underestimate since some of the data will be used by more than one block, but on the other hand it will be an overestimate because not all of the data will be needed for just one loop.

7 OP2 execution

An OPlus2 calculation has three stages of execution, each with a number of steps. In the description below, the steps in parentheses are the extra ones needed for the phase 2 MPI parallelisation:

1. initialisation

- declare sets, pointers and datasets
- (partition sets and renumber pointers)
- (compute halos)
- (load and distribute datasets)
- renumber elements and pointers
- initialise the GPU, copy constants to its constant memory and copy datasets to its global memory

2. parallel computation

- (exchange halo data as necessary)
- build new CUDA execution “plan”, if necessary
- execute “plan”

3. termination

- copy datasets back to host
- terminate CUDA operations on the GPU
- (write datasets to disk)

8 Parallel loop syntax

My initial idea is that the parallel loop should have the following syntax:

```
op_par_loop(kernel, set, label,  
            arg1, ptr1, index1, type1, dim1, access1,  
            arg2, ptr2, index2, type2, dim2, access2,  
            ..., ..., ..., ..., ..., ...,  
            argN, ptrN, indexN, typeN, dimN, accessN)
```

kernel	name of the kernel function to be applied to each element
label	a label string for diagnostic purposes
set	set whose elements are to be processed
arg1	first dataset
ptr1	pointer from main set to set on which arg1 is defined
index1	index of pointer to access arg1
type1	type of dataset (e.g. “float”, “int”)
dim11	dimension of dataset
access1	information on how arg1 is used within kernel “r” = read-only “w” = write-only “b” = both read + write “i” = increment

Here **kernel** is a routine which process a single element in **set**. This will get converted by a pre-processor into a routine called by the CUDA kernel function. The pre-processor will also take the specification of the arguments and turn this into the CUDA kernel function which loads in the data from the device main memory into the shared storage, then calls the converted **kernel** for each element, then writes back the updated data to the device main memory.

Parallel loops of the above form will form the main bulk of the application code, after an initialisation stage in which the sets, the pointers between the sets and the data associated with the sets are all declared.

This syntax combines the functions of **OP_PAR_LOOP** and **OP_ACCESS** in the current OPlus, and the specification of the single element **kernel** function fits in well with the use of automatic differentiation in HYDRA.

There is a fair amount of redundant information here, but the implementation will check it is consistent; see discussion in Blog section.

There are also routines to declare sets, pointers and datasets:

op_decl_set(elems, dim, coords, set, label, mapping, maplabel)

elems	number of elements in the set
dim	dimension of the coordinate data (for renumbering)
coords	coordinate data
set	structure containing set info (set by routine)
label	a label string for diagnostic purposes
mapping	an identity mapping (needed for some parallel loops)
maplabel	a label for the identity mapping

op_decl_ptr(from_set, to_set, dim, ptrdata, ptr, label)

from_set	set from which it points
to_set	set to which it points
dim	number of pointers per element
ptrdata	input pointer data
ptr	structure containing ptr info (set by routine)
label	a label string for diagnostic purposes

op_decl_dat(set, dim, type, data, dat, label)

set	set with which the data is associated
dim	amount of data per element
type	type of data (e.g. “float”, “int”)
data	input data array
ptr	structure containing data info (set by routine)
label	a label string for diagnostic purposes

9 Execution plan construction

New notes: 1/11/09

I’ve almost finished the coding for the execution plan construction so it’s time to make some notes, mainly for my own benefit but also to help anyone reading the code.

A call to **op_par_loop** leads to a call to **plan** which looks to see if there is an existing plan. If there is, it returns its ID, otherwise it constructs a new plan and returns its ID. There is a match to an existing plan if all of the arguments match. Note that this needs to be a run-time match; it is not something that can be decided at compile-time since the same call in the code can be used for different levels within a multigrid solver, and each level requires its own execution plan.

In constructing a new plan, the first step is to see if the loop involves any indirection. If it does not, the execution plan and the corresponding CUDA code is very simple.

When there is indirection, the construction of the plan involves the following steps:

- identification of the datasets involving indirection, and the “increment” subset of these which involve an **OP_INC** access producing a potential data conflict which requires coloring
- for each thread block:
 - for each indirection dataset:
 - * assemble list of elements pointed to
 - * sort into ascending order and eliminate duplicates
 - * store away for use by CUDA routine
 - * re-number pointers accordingly and store them
 - color the threads (and perhaps re-order threads to group by color to minimise warp divergence?)
- color the blocks (and perhaps re-order the blocks to group by color?)

9.1 Coloring

Coloring is potentially time-consuming, so an important “trick” here is to assign 32 colors at a time using bit operations. I explain it here for thread coloring but the same approach is also used for block coloring.

The key to the implementation is an array with a single 32-bit unsigned integer for each indirectly addressed increment set element. This is initialised to zero and then, as colors are assigned, the corresponding bits will be set to indicate the color of the elements accessing that indirectly addressed element.

The algorithm proceeds as follows:

- loop over all elements which have not yet been colored, and for each one
 - initialise mask variable to zero
 - loop over all indirect increment set elements and “add” to the mask using the logical OR operation
 - find first bit of mask which is not set; this is the color of the new element (provided there is a bit not set)
 - reset the mask to have the appropriate color bit set
 - loop back over all indirect increment set elements “adding” the mask using the logical OR operation to set the appropriate the color bit
- if 32 colors weren’t sufficient, reset the work array to zero and repeat the whole procedure for those which have not yet been colored

In most cases I expect that 32 colors will be sufficient, and so a single pass through each thread block will be all that is required for the thread coloring, and then a single pass through the entire set for the block coloring.

9.2 Different access type for same dataset

What should be done if different argument involve the same dataset with different access types?

- treat each access type as giving a different indirect dataset for the purposes of the execution plan construction
- requires there to be no data conflict between these different sets; must be explained carefully in documentation, and checked in code which validates correct usage

10 Blog

Here I make various quick notes, partly for my own benefit so that I can remember why I did something, or to record things for the future, and partly to inform others about the reasons for my actions.

The thoughts are not carefully organised or well explained – please ask questions if you think something looks interesting but unclear.

- In designing the parallel loop syntax I wanted something simple and clean, without lots of redundant information. This is how I designed the initial CPU implementation, but the more I think about it and how the source code transformation will turn it into optimised code, the more I think it's better to go with something closer to the original OPlus syntax with redundant information. This will enable compile-time optimisations (e.g. unrolling of loops copying data into local shared-memory arrays), and the redundant info can be checked for consistency.
- I'm not sure how best to handle different datatypes. My initial implementation supported only datasets composed of floats. Rather than having different dataset types for each basic type, I think it may be best to change the struct to use void pointers and store the underlying datatype. Alternatively, can a C++ guru construct a parameterised class? If I use void pointers, then I should do run-time type-checking.
- I'm going to have to be careful about the total size of all of the execution plans, but I think they won't be too large compared to the datasets.
- The optimal block sizes may be hard to determine a priori. It may make sense to use dynamic optimisation, adjusting them at run-time to reduce execution time.
- When pointers are not used, data can be loaded straight into registers, not shared-memory.
- When pointers are used, it's best to use shared-memory rather than relying on caching because with shared-memory only need to store the data which is needed.
- In the future, the labels for various routines can be handled automatically either through using macros or through the source code transformation.
- There are alternatives to the thread-colouring approach. One is to use atomic locks, and another is to store the increments in shared-memory without combination then use one thread for each output to combine things sequentially. Ideally, it would be good to try each of these alternatives.

- Coping with a variable number of parameters might be painful. C++ has `va_list` to handle a variable number of inputs, but unlike MATLAB I think there's no support for calling a function with a variable number of parameters. FORTRAN can't do either which is why OPlus had a separate call for each argument, which coincidentally is similar to what happens in OpenCL. Maybe I should stick with that approach for FORTRAN, and for C++ use a template with the number of parameters being specified as a template parameter?
- The `m` loops in `op_par_loop` would be unrolled by the source code transformation, and lots of code eliminated by evaluating many of the `if` tests. The NVIDIA compiler would then automatically unroll the little `p` loops in most cases.