OP2 Developer's Guide

Mike Giles

July 23, 2010

Abstract

This document explains some of the algorithms and implementation details inside OP2. It is intended primarily for those who are developing OP2, and it is assumed they have already read the OP2 Notes document. Some of the material in that may be transferred to this document in the future.

Those who are only using OP2 should instead read the User's Manual.

1 Plan construction

This section deals with the algorithms in routine **plan** within the file **op_lib.cu**. This is called whenever a parallel loop has at least one indirect dataset.

1.1 indirect datasets

"Sets" are things like nodes or edges, a collection of abstract "elements" over which the parallel loops execute. "Datasets" are the data associated with the sets, such as flow variables or edge weights, which are the arguments to the parallel loop functions.

In a particular parallel loop, an "indirect dataset" is one which is referenced indirectly using pointers. Note that more than one argument of the parallel loop can address the same indirect dataset. For example, in a typical CFD edge flux calculation, two arguments will correspond to the flow variables belonging to the nodes at either end of the edge, and another two arguments will correspond to the flux residuals at either end.

The pre-processor **op2.m** identifies for each parallel loop the number of arguments `nargs`, the number of indirect datasets `ninds` and the mapping from arguments to indirect datasets `inds[]`. The last of these has an entry for each argument; if it is equal to -1 then the argument does not reference an indirect dataset. All of this information is supplied as input to the routine **plan**.

1.2 local renumbering

The execution plan divides the execution set into mini-partitions. These are referred to in the code as “blocks” because it’s a shorter word. This is a slightly different use of the word “block” compared to CUDA thread blocks, but each plan block is worked on by a single CUDA block so it’s hopefully not too confusing.

The plan blocks are sized so that the indirect datasets will fit within the limited amount of shared memory available to each SM (“streaming multiprocessor”, NVIDIA’s preferred term to describe each of the execution units in their GPUs). The idea is that the indirect datasets are held within the shared memory to maximize data reuse and avoid global memory traffic. However this requires renumbering of the pointers used to reference these datasets.

For each plan block, and each indirect dataset within it, the algorithm for the renumbering is:

- build a list of all references to the dataset by simply appending to a list
- sort the list and eliminate duplicates – this then defines the mapping from local indices to global indices
- use a large work array to invert the mapping, to give the mapping from global indices to local indices (note: this is obviously only needed for the global indices occurring within that block)
- create a new copy of the pointer table which uses the new local indices

Note that each indirect dataset ends up with its own duplicate pointer table. In some cases, the indirect datasets had the same original pointer tables; for example, in the CFD edge flux loop described before the flow variables and flux residuals were referenced using the same edge-node pointers. In this case, we are currently wasting both memory and memory bandwidth by duplicating the renumbered pointer tables. This should be eliminated in the future, by identifying such duplication, de-allocating the duplicates, and changing the pointer to the duplicate table to point to the primary table.

Note also that for multi-dimensional pointers one has to use the appropriate pointer index as specified in the inputs, and the re-numbered pointers which are stored are for that index alone.

1.3 coloring

Coloring is used at two levels to avoid data conflicts. The elements within each block are colored, and then the blocks themselves are colored.

We start by describing the element coloring. The goal is to assign a color to each element of that no two elements of the same color reference the same indirect dataset element.

Conceptually, for each indirect dataset element we maintain a list of the colors of the elements which reference it. Starting with this list initialised to be empty, the mathematical algorithm treats each set element in turn and performs the following steps:

- loop over all indirect dataset elements referenced by the set element to find the lowest index color which does not already reference them
- set this to be the color of the element
- loop again over all indirect datasets, adding this color to their list

The efficient implementation of this uses bit operations. The color list for each indirect dataset element is a 32-bit integer in a work array, with the i^{th} bit set to 1 if it is referenced by an element of color i. The first step is performed by using the bit-wise `or` operation to combine the color lists into a variable called `mask`, followed by using the `ffs` instruction to find the first zero bit. The third step is also performed by a bit-wise `or` operation.

Doing it in this way, we can process up to 32 colors in a single pass. This is probably sufficient for most applications, but when it is not, the code loops back. i.e. in the first pass, if the first step finds that all bits are already set, it doesn't assign a color to the element, and goes on to the next element. Then at the end it goes back to process the elements which have not yet been colored, with the color lists re-initialised to indicate that the indirect set elements are not referenced by any of the "new" colors. This outer loop (controlled by the variable `repeat`) is repeated until all elements have been colored.

The block coloring is performed in exactly the same way, except that in the first and third steps the loop is over all indirect dataset elements referenced by all of the elements in the block, not just by a single element.

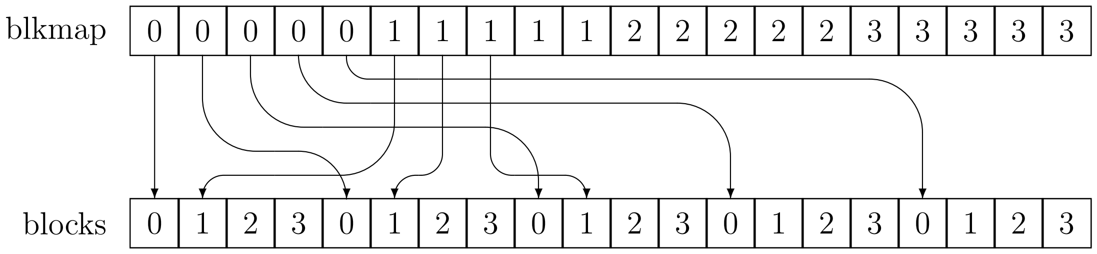

Figure 1: Illustration of block mapping, with colors indicated as 0, 1, etc.

1.4 block mapping

The final part of **plan** defines a block mapping. As illustrated in the bottom row of Fig. 1, **plan** constructs blocks and stores their data in the order in which they are generated, so they are not grouped by color.

Rather than reordering the blocks to group them by color, I instead construct the `blkmap` mapping from a grouped arrangement to the actual blocks. This, together with the number of blocks of each color, is all that is needed for the later kernel execution.

The algorithm to compute `blkmap` is:

- compute the total number of blocks of each color
- do a cumulative summation to obtain the sum of all blocks of preceding colors
- processing each block in turn, add the number of preceding blocks of the same color to the cumulative sum of preceding colors, to obtain its position in the `blkmap` array
- finally, undo the cumulative summation operation to store the number of blocks of each color

1.5 rest

The first part of **plan** checks whether there is an existing plan to deal with this parallel loop, and if not it does some self-consistency checking.

The final part of **plan** computes the maximum amount of shared memory required by any of the blocks, and copies the plan arrays over onto the GPU, keeping the pointers in the `plan` structure.

1.6 op_plan struct

The first part of the **op_plan** struct stores the input arguments used to construct the plan. These are needed to determine whether the inputs for a new parallel loop match an existing plan.

The second part contains the data generated by the **op_plan** routine:

- `nthrcol`*: an array with the number of thread colors for each block
- `thrcol`*: an array with the thread color for each element of the primary set
- `offset`*: an array with the primary set offset for the beginning of each block
- `ind_ptrs`*: a 2D array, the outer index over the indirect datasets, and the inner one giving the local → global renumbering for the elements of the indirect set
- `ind_offs`*: a 2D array, the outer index over the indirect datasets, and the inner one giving the starting offset into `ind_ptrs` for each block
- `ind_sizes`*: a 2D array, the outer index over the indirect datasets, and the inner one giving the number of indirect elements for each block
- `ptrs`*: a 2D array, the outer index over the datasets, and the inner one giving the indirect pointers to local indices in shared memory
- `nelems`*: an array with number of primary set elements in each block
- `ncolors`: number of block colors
- `ncolblk`: an array with number of blocks for each color
- `blkmap`*: an array with mapping to blocks of each color
- `nshare`: number of bytes of shared memory require to execute the plan

The data in the arrays marked * is initially generated on the host, then transferred to the device, with only the array pointers being retained on the host.

1.7 op_plan_check

This routine checks the correctness of various aspects of a plan. The checks are performed automatically after the plan is constructed, depending on the value of the diagnostics variable `OP_DIAGS`.

2 op2.m preprocessor

In this section I describe the code transformation performed by **op2.m** and discuss various aspects of the code which is generated.

2.1 parsing the op_par_loop calls

As explained in Section 1.1, the pre-processor finds each `op_par_loop` call and parses the arguments to identify the number of indirect datasets which are used, and to define the `inds[]` mapping from the arguments to the indirect datasets. It also identifies how each of the arguments is being used (or "accessed").

If there are no indirect datasets the stub and kernel functions which are generated are fairly simple. The descriptions in the next two sections are for the more interesting case in which there is at least one indirect dataset.

2.2 the stub routine

The stub routine is the host routine which is called by the user's main code. If there are any local constants or global reduction operations it starts by transferring this data to the GPU.

It then calls **plan** to generate the execution plan, passing into it the information about indirect datasets which has been determined by the parser.

It then calls the kernel function to execute the plan. This is done within a loop over the different blocks colors, with an implicit synchronization between each color to avoid any data conflicts

One of the kernel parameters is the amount of dynamic shared memory; this comes from the maximum requirement determined by **plan**.

Finally, for global reductions it fetches the output data back to the CPU.

2.3 the CUDA kernel routine

Most of the code in **op2.m** is for the generation of the CUDA kernel routines. To understand this it is probably best to look at an example of the generated code (e.g. `res_kernel.cu`) while reading this description.

The key bits of code which are generated do the following:

- declare correct number and type of input arguments, including indirect datasets
- declare working variables, including local arrays which will probably be held in registers (or in L1 cache on Fermi)
- get block ID using `blkmap` mapping discussed in Section 1.4
- shift the pointers for the direct datasets to the correct starting point for this block
- set the dynamic shared memory pointers for indirect datasets; see the CUDA Programmer's Guide for more info on this
- copy the read-only indirect datasets into shared memory, and zero out the memory for those being incremented
- synchronize to ensure all shared memory data is ready before proceeding
- loop over all set elements in the block, and for each one
  - zero out the local arrays for those being incremented
  - execute the user function
  - use thread coloring to increment the shared memory data, with thread synchronization after each color
- increment global storage of indirect datasets
- complete any global reductions by updating the values in the main device memory

Note: it is likely that the compiler will put small local arrays of known size into registers. This is why users should specify op_par_loop array dimensions as a literal constant. (Currently, **op2.m** doesn't handle dimensions which are set at run-time, but that capability should be added in the future.)

There's one technical implementation detail which is very confusing. In the code, the number of elements in the block is nelems. The variable nelems2 is equal to this value rounded up to the nearest multiple of the number of threads in the thread block. This ensures that every thread goes through the main loop the same number of times. This is important because the thread synchronization command __syncthreads(); must be called by all threads. The if test within the loop prevents execution for elements beyond the end of the block, and the default color is set to ensure no participation in the colored increment. If we switch to atomic updates, the thread synchronization will no longer be needed, and so the main loop can simply go up to nelems.

(Note: the generated code includes a number of pointers which are probably held in registers, even though they have the same value for all threads. It might be better to explicitly put them into shared memory, using thread 0 to set their value.)

2.4 the new source file

The new source file generated by **op2.m** has only minor changes from the original source file:

- new header file and declaration of function prototypes
- new names for each op_par_loop call

2.5 the master kernels file

The master kernels file is a single file which includes the kernel files for each parallel loop along with some header files and the declaration of the global constant variables which come from parsing any op_decl_const calls.

It has to be done this way in the current version of CUDA for the constants to have global scope over all kernel files.

3 Global reductions

3.1 Summation

The global value on the GPU is initialised by transfer from the CPU.

Each thread sums its contributions, with the sum being initialised to zero. The combined contributions from a single thread block are then combined in a binary tree reduction process modelled on that the SDK "reduction" example, using shared memory.

The thread block sum is added to the global value by using an atomic lock to ensure that only one block at a time adds its contribution, and then once the CUDA kernel call is complete the final value is transferred back to the CPU

3.2 Min/max reductions

The treatment of min/max reductions is very similar. At the thread level the minimum/maximum is initialised using the current global value and then updated using the thread's subsequent contributions. A binary tree approach combines these to form a thread block minimum/maximum, and then this is combined with the global minimum/maximum using the atomic lock.

4 User-defined datatypes

OP2 supports user-defined datatypes for datasets, global constants, local constants and global reductions.

4.1 Run-time type-checking

Run-time type-checking is implemented in a portable way in which the user is required to provide a `type_error` routine for each user-defined datatype. The `type_error` routines for the standard datatypes are defined in `op_datatypes.h`.

4.2 Zero element for incrementing

When executing `op_par_loop` for a case in which a dataset is incremented, the CUDA code which is generated by **op2.m** initialises the increment to zero, calls the user-supplied kernel function to compute/add the new increment, and then uses coloring to add the increment to the dataset.

With a user-defined datatype, the addition in the last step requires that the user has defined an overloaded addition operator. The user also has to specify a zero element, called `ZERO_typename`, which is used in the first step to initialise the increment correctly.

The zero elements for the standard datatypes are defined in `op_datatypes.h`. Note that `typedef` is used to alias `unsigned int`, `long long` and `unsigned long long` to `uint`, `ll` and `ull`, respectively. This is because `typename` must not contain any spaces, otherwise there are problems with the compiler parsing of `ZERO_typename`.

4.3 Global reductions

Summations should be fine, and min/max reductions also ought to work provided the user has correctly overloaded the inequality operators so that for all a, b, c,

- $a\!>\!b \;\cap\; b\!>\!c \implies a\!>\!c$ and $a\!<\!b \;\cap\; b\!<\!c \implies a\!<\!c$
- either $a\!<\!b$, or $a\!>\!b$, or $a\!=\!b$

4.4 Alignment padding

When user-defined datatypes have elements with different sizes, the compiler automatically introduces padding to ensure correct data alignment, plus padding at the end, if necessary, to ensure that the next element in an array starts with the correct alignment.

However, when using dynamic shared memory in CUDA, it is the programmer's responsibility to ensure the correct alignment, as explained in section B.2.3 of the CUDA Programming Guide, version 3.0. This requires the use of a `ROUND_UP` macro in **op_lib.cu** and the CUDA kernels generated by **op2.m**. This macro is modelled on the macro `ALIGN_UP` in the Programming Guide.

Similar padding is used in assembling local constants and global reduction values into contiguous arrays for transfer to/from the GPU.

4.5 Future MPI treatment

Looking to the future, MPI messaging will be performed by treating all datasets as having elements of type `MPI_CHAR` with the appropriate byte size. The only minor issue with this is the inability to use standard MPI reduction operators, but this could be handled by implementing an all-to-all data exchange followed by local reduction, with an option for MPI reductions being used for standard datatypes.

5 CUDA 3.0 notes: 20/02/10

Having read the new CUDA 3.0 Programmer's Guide and the guide on Tuning CUDA Applications for Fermi, here I make a few notes which I should read over again in the future.

- should think about turning off L1 caching of global data so L1 cache is preserved for register spilling of local variables
- should use the `restrict` qualifier to enable compiler optimization
- new LDU instruction will be good for local constants; must make sure to use `const` qualifier
- use 48kB shared memory for kernels with indrect datasets, and 48kB L1 cache for those without
- L2 cache has the equivalent of 48kB per SM, as much as the maximum shared memory, so maybe it would be good to use it? But on the other hand, the effective size is smaller (as it also stores data which is not wanted) and we're unlikely to be able to achieve a perfect allocation of data to shared memory and cached global memory, so maybe it's not worth worrying about?
- no point in using texture mapping?
- consider the compiler options `-ftz=true -prec-div=false -prec-sqrt=false` for improved performance

6 To do list

- Add capability to handle dataset (and local and global constant) dimensions which are not known until run-time.

- Set block sizes to maximum which will fit in in worst case (and write up why I think it's a good default choice)

- Renumber sets and pointers to improve data reuse.

- Modify **op2.m** to cope with more than one parallel loop using the same user kernel function, and more than one call to set the value of the same global constant.

- For indirect datasets, maybe use only first warp for read/write to improve caching? Or change the implementation so that each element of an dataset for a particular set element is loaded by a different thread in the warp (instead of the same thread as at present)?

- For some parallel loops (like the gradient calculation in HYDRA) which are data-intensive and not compute-intensive, it might be better for the thread coloring to also control the user kernel execution, so that the indirect arrays held in shared memory can be directly incremented rather than using local variables to hold the temporary increments.