OP2 Developers Guide

Mike Giles

December 2013

Abstract

This document explains some of the algorithms and implementation details inside OP2. It is intended primarily for those who are developing OP2; those who are only using OP2 should instead read the Users Manual.

Contents

1 Overall parallelisation strategy

The OP2 design uses hierarchical parallelism with two principal levels.

At the higher level, OP2 is parallelized across distributed-memory clusters using MPI message-passing. This uses essentially the same implementation approach as the original OPlus. The domain is partitioned among the compute nodes of the cluster, and import/export halos are constructed for message-passing. Data conflicts when incrementing indirectly referenced datasets are avoided by using an "owner-compute" model, in which each process performs the computations which are required to update data owned by that partition. This approach involves some amount of redundant computation; for example, if the computation for a particular edge results in increments to the data at two nodes belonging to different partitions, then both of those partitions will need to perform this edge computation. However, there will be a minimal level of redundant computation if the cluster satisfies the first of our key assumptions:

Key assumption 1: each compute node will have GBs of memory

At the lower level, there are different kinds of parallelism depending on the target hardware:

- on NVIDIA GPUs, the lower level is split into two, with multiple thread blocks, and multiple threads within each block;
- on CPUs, the lower level may also be split in two, with shared-memory multithreading through OpenMP, and a possibility of each thread using vectorization to exploit the capabilities of SSE/AVX vector units.

In both cases, we have two further key assumptions:

Key assumption 2: memory bandwidth is a major limitation

In the case of GPUs, this refers to the bandwidth between the main graphics memory and the GPU, whereas for CPUs it is the bandwidth between the main system memory and the CPUs.

Key assumption 3: there is very little local shared memory

In the case of GPUs, this refers to shared memory with a SM (Streaming Multiprocessor) unit of the GPU, whereas for CPUs it refers to the size of the L1 cache.

These assumptions will motivate the implementation design decisions.

2 CUDA parallelisation strategy

Because of Key Assumption 2, the starting point in designing the CUDA implementation is to minimize the amount of data which needs to be transferred between the graphics memory and the GPU.

There are two ways to do this; through explicit staging in shared memory which is used when generating code for Fermi-generation GPUs, or by relying on caches (texture and L2), which is used for Kepler GPUs.

For indirect loops (loops which involve indirectly-referenced datasets) this leads to the idea of working with mini-partitions which are small enough so that all of the indirect datasets for each of the mini-partitions is able to fit into the limited shared memory on each SM within an NVIDIA GPU.

Read-only (and read-write) indirect data is loaded in once at the beginning of a CUDA kernel and then re-used as needed by the threads working within the thread block. Write-only (and read-write) indirect data is written back to graphics memory at the end of a kernel.

For data which is incremented, the shared memory array of increments is initialized to zero, and then at the end of the kernel the increments are applied to the global array held in the graphics memory. In doing this, we need to avoid the potential data conflicts (race conditions). For example, in the "airfoil" testcase in the OP2 distribution, different edges in the `res` routine can update data at the same cell. The problems are avoided by coloring [1] at two levels:

- the blocks are colored so that there is no data conflict between two blocks of the same color – a separate CUDA kernel call is used for each block color, with a synchronization between colors to ensure there is no conflict;
- the threads within each block are colored, and the increments are applied one color at a time, with a `__syncthreads` thread synchronization between each thread color – this results in some loss of performance due to CUDA warp divergence (i.e. a lot of threads are idle during this incrementing process) but overall it is not large. [2] [3]

2.1 Execution plans

In standard MPI computations, because of the cost of re-partitioning, the partitioning is usually done just once and the same partitioning is then used for each stage of the computation.

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 parallel loop calculation can be considered independently of the requirements of the other parallel loops.

[1] This was a well-established technique for vector computing.

[2] To minimize the number of colors within each thread "warp", it may be best to reorder the threads within each block. This would scramble up the identity mapping used for directly-referenced data, so there would be some communication cost, but it's possible there would be savings overall.

[3] An alternative to thread coloring would be for each thread to store its increment separately in the shared memory, and then have an additional loop within the kernel in which the increments for each data element are combined by a unique thread. This approach has been used for a finite element application by C. Cecka at Stanford, but it requires a lot of shared memory. Because of Key Assumption 3, this would lead to extremely small mini-partitions, which in turn would give very poor data reuse, and so we chose not to try this approach.

Based on ideas from FFTW, we construct for each parallel loop an execution "plan" which is a customized 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.

2.2 Renumbering

The execution plans construct mini-partitions by breaking the set being looped over into a number of contiguous blocks. This is good for direct datasets since they will be contiguous and it will lead to coalesced memory transfers. However, for indirect datasets we want to make sure that each mini-partition references elements which are held close together in the graphics memory.

An initial step (which has not yet been implemented: see the "To Do list" at the end of this document) is therefore to reorder/renumber all sets to improve data locality, so that neighboring 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.

2.3 Some rough numbers for HYDRA

A big calculation by the Rolls-Royce HYDRA CFD code will use most of the 6GB on a C2070 card. Given that the local shared-memory size is 48kB, this suggests that the data intensive parallel loops will have over 100,000 blocks. 10 block colors might be needed to avoid data conflicts, suggesting up to 10,000 blocks per color.

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 color so we need to use color-locked increments, with thread synchronization in between colors.

3 Plan construction

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

3.1 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 a mapping from another set. 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 **op_plan_core**.

3.2 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 mappings 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 mapping table which uses the new local indices

Note that each indirect dataset ends up with its own duplicate mapping table. In some cases, the indirect datasets had the same original mapping tables; for example, in the CFD edge flux loop described before the flow variables and flux residuals were referenced using the same edge-node mappings. In this case, we are currently wasting both memory and memory bandwidth by

duplicating the renumbered mapping 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 mappings one has to use the appropriate mapping index as specified in the inputs, and the re-numbered mappings which are stored are for that index alone.

3.3 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 initialized 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-initialized 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.

3.4 block mapping

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

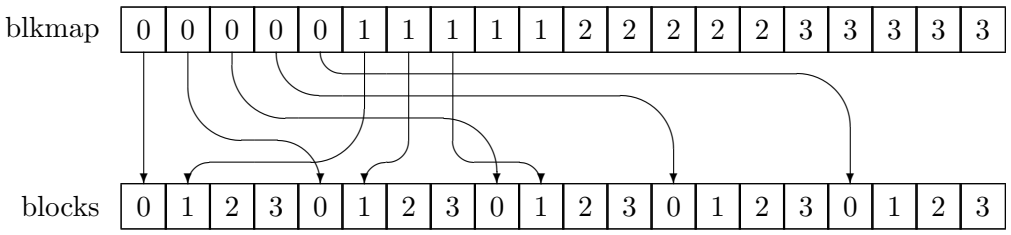

Figure 1: Illustration of block mapping, with colors indicated as 0, 1, etc.

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

3.5 rest

The first part of **op_plan_core** 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 **op_plan_core** 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 `op_plan_core` structure.

3.6 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_maps`*: a 2D array, the outer index over the indirect datasets, and the inner one giving the local $\rightarrow$ 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_maps` 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
- `maps`*: a 2D array, the outer index over the datasets, and the inner one giving the indirect mappings 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.

3.7 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`.

4 Data layout

One key implementation choice is how to store datasets in which there are multiple items for each set element. For example, in the "airfoil" testcase there are four flow variables for each cell.

The two alternatives are:

- SoA: for each component, store the data for all of the set elements as a contiguous block.
- AoS: for each set element, store all of the components together as a small contiguous block;

The SoA format (which corresponds to what is referred to as "a struct of arrays") is what used to be used on CRAY vector computers 25 years ago. It is ideally suited for systems which stream data into and out of the processing unit, and a number of GPU programmers believe it is the best approach for GPUs as well because it leads to natural coalescence in memory transfers.

The AoS format (which corresponds to the alternative "array of structs") is the preferred choice for systems with a cache hierarchy. Assuming that all of the components are used within the computation, this exploits spatial locality within a cache line; having loaded a cache line to access one component, the other components are immediately available as well.

To understand the tradeoff between these two storage formats, we need to consider cache line efficiency, the extent to which a computation uses all of the data in a cache line. In the "airfoil" testcase we need 4 floats per cell, and the Fermi cache line size is 128 bytes, corresponding to 32 floats. When data is accessed indirectly, the SoA format can lead to a worst-case scenario in which only 1/32 of the cache line is used, whereas with the AoS format the worst case is that only 1/8 of the cache line is used. Hence, in extreme cases with almost random addressing, the AoS format could be 4 times more efficient than the SoA format, and therefore require 1/4 of the data transfer from the graphics memory to the GPU relative to SoA. The savings could be even larger for applications with more data per set element.

Another way of looking at this is that if there are K data items per set element, then the SoA format with a cache line of size S will have the same cache line efficiency as the AoS format with a cache line of size KS. Hence, the AoS format increases the cache line size relative to the SoA format, and therefore decreases the cache line efficiency.

For this reason, I have chosen to use the AoS format. The potential disadvantage of this choice is that it destroys the natural memory coalescence of the SoA format for direct loops. However, the memory coalescence is still achieved in OP2 through some additional programming effort.

4.1 direct loops

To achieve memory coalescence for direct loops we need to use shared memory. As shown below in an example taken from `save_soln_kernel.cu` in the "airfoil" testcase, the threads in a CUDA warp use a coalesced memory transfer to load into shared memory, and then copy it into local arrays which the `nvcc` compiler will map to registers.

```
for (int m=0; m<4; m++)
  ((float *)arg_s)[tid+m*nelems] = arg0[tid+m*nelems+offset*4];

for (int m=0; m<4; m++)
```

```
arg0_l[m] = ((float *)arg_s)[m+tid*4];
```

Here `tid=threadIdx.x%OP_WARPSIZE` is the thread ID (modulo the warpsize) and `nelems` is the number of active threads in the warp; this is usually equal to the warpsize, but if the set size is not an integer multiple of the warpsize then there's a final warp in which some threads are not active.

Usually when using shared memory one has to use `__syncthreads` to synchronize the actions of different warps. However, this has a significant impact on the overall performance, and is avoided by giving each warp its own separate shared memory "scratchpad" to work in so there are no conflicts between different warps. This is accomplished by the declaration:

```
char *arg_s = shared + offset_s*(threadIdx.x/OP_WARPSIZE);
```

with the scratchpad size `offset_s` set to be large enough to handle each of the datasets being loaded.

The same technique is used in reverse for coalesced writing back to the graphics memory:

```
for (int m=0; m<4; m++)
  ((float *)arg_s)[m+tid*4] = arg1_l[m];

for (int m=0; m<4; m++)
  arg1[tid+m*nelems+offset*4] = ((float *)arg_s)[tid+m*nelems];
```

Note: this treatment is only needed when there is more than one data element per set element. When there is only one, there is no need to pre-load it into a register; the user kernel can load it directly from device memory in a single coalesced transfer.

4.2 indirect loops

In indirect loops the loading of data from graphics memory into shared memory involves code of the following form, taken from `res_calc_kernel.cu` in the "airfoil" testcase.

```
for (int n=threadIdx.x; n<ind_arg1_size*4; n+=blockDim.x)
  ind_arg1_s[n] = ind_arg1[n%4+ind_arg1_map[n/4]*4];
```

Here `ind_arg1_map[]` is an array giving the indices of the set elements to be loaded. This example has 4 components of data per set element. Hence the first 4 threads load the data for the first element, the next 4 threads load the data for the second element, and so on.

Since the indices in `ind_arg1_map[]` are sorted in ascending order, the indices into the array `ind_arg1` are also in ascending order. This maximizes the degree of memory coalescence achievable. Each cache line will be accessed by at most two consecutive warps, which minimizes the reliance on L1 and L2 caches. This is important because in the future we may want to turn off L1 caching (using the compiler flag `-Xptxas -dlcm=cg`; see section G.4.2 in the CUDA Programming Guide) so that the L1 cache memory is instead reserved exclusively for register spills.

Writing indirect datasets back to graphics memory is accomplished in a similar manner.

5 op2.py preprocessor

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

5.1 parsing the op_par_loop calls

As explained in Section **??**, 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.

5.2 the CUDA 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 **op_plan_get** to get an existing plan or generate a new one, 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 **op_plan_core**.

Finally, for global reductions it fetches the output data back to the CPU and does the final processing, combining the partial results from each thread block.

5.3 the CUDA kernel routine

Most of the code in **op2.py** 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 **??**
- set the dynamic shared memory pointers for indirect datasets; see the CUDA Programmer's Guide for more info on this

- zero out the memory for those being incremented, for Fermi, copy the read-only indirect datasets into shared memory
- 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/global (Fermi/Kepler) memory data, with thread synchronization after each color
- on Fermi cards, increment global storage of indirect datasets from shared memory
- 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.py** 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.

The generated code includes a number of pointers which are computed by thread 0 and held in shared memory. This is because the same values are needed for all threads, and this minimizes register usage.

5.4 the CUDA 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.

5.5 the OpenMP files

The OpenMP files generated by **op2.py** execution pattern is very similar to the CUDA one, but it is not necessary to use element coloring within a mini-partition nor to launch a kernel, therefore the OpenMP implementation uses a single function, computing indices and offsets and then looping over the elements of the mini-parition, calling the user kernels with pointers to "global" memory; no staging is used.

The other point worth discussing is the OpenMP implementation of global reductions. This is handled in a similar way to the CUDA implementation. Each OpenMP thread does a partial reduction using thread-specific local variables. These are then combined in a final sequential step.

5.6 the sequential files

While OP2 can run sequentially (and with MPI) through the *op2_seq.h* header file, this is aimed at debugging and verification purposes; it uses very generic logic and function pointers which add a significant oberhead and prohibit some compiler optimisations. Therefore we generate sequential stub files (referred to as genseq) that use information parsed by **op2.py** to generate code specific to each loop; mappings for indirectly accessed data are only read once and then re-used, pointers are spelled out explicitly during the iteration through the elements in the set.

5.7 the new source file

The new source file generated by **op2.py** for both CUDA and OpenMP execution 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

6 Global reductions

6.1 Summation

Each thread block has a separate entry in a device GPU array which is initialized to zero.

Each thread sums its contributions, with the sum being initialized 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 then added to the appropriate global entry for that block. Once the CUDA kernel call is complete the final block values are transferred back to the CPU and added to the original starting value.

6.2 Min/max reductions

The treatment of min/max reductions is very similar. Each thread block again has a separate entry in a device GPU array, but in this case it is initialized to the initial CPU input value.

At the thread level the minimum/maximum is initialized using the current global value for that block, and then updated using the thread's subsequent contributions. A binary tree approach combines these to form a thread block minimum/maximum.

The thread block minimum/maximum values are used to update the global value. Once the CUDA kernel call is complete the final block values are transferred back to the CPU and combined to give the overall minimum/maximum.

6.3 User-defined datatypes

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,

- $a>b,\ b>c \implies a>c \quad$ and $\quad a<b,\ \ b<c \implies a<c$
- either $a<b$, or $a>b$, or $a=b$

In the future, the MPI implementation could be handled by implementing an all-to-all data exchange followed by local reduction, with an option for standard MPI reductions to be used for standard datatypes.

7 AVX implementation

Experimental code generators can create code for AVX execution by explicitly combining a number of set elements into vector operations, first gathering data into vector registers, executing the user kernel with vector types that have operators overloaded and then scattering the data from vector registers. The code generated uses C++ classes to hide the use of vector intrinsics, which are defined in *op2_vechtor.h*

Since alignment is an important issue for directly accessed data, the loop over set elements is broken into three parts; a scalar pre-sweep to get aligned to 256/512 bits, the main vectorised loop and then a scalar post-sweep to handle the elements left over.

There are some restrictions: branching can not be used in the user kernel because we cannot overload that, instead *select*() instructions can be used.

8 Auto-tuning

To optimize the run-time performance, I think it will be essential to use auto-tuning techniques, building on ideas already used by packages such as ATLAS and FFTW. To emphasize this point, I don't think I am talking about squeezing out an extra 10% of performance; I think it is more likely that auto-tuning will double the performance.

Here I list some of the choices to be optimized, and the tradeoffs they involve:

- Size of block partition

  To maximize date reuse within each block, the natural choice is to make the block partition as large as possible subject to the constraint that it fits inside the shared memory. However, making it a bit smaller would allow multiple blocks to run concurrently on the same multiprocessor, and the run-time scheduler would then naturally overlap I/O for one block with computation for another.

- Number of threads per block

  One option is to have one thread per set element, giving the run-time scheduler the maximum freedom. However, this increases the cost of thread synchronization, might run into difficulties with the total register usage, and doesn't amortize startup costs over multiple elements.

These choices exist for each parallel loop, but what is optimal for one loop may not be optimal for another. Optimization strategies which could be used include:

- Brute force exhaustive search
- Genetic algorithms (or other stochastic method)
- Run-time optimization

9 To do list

- 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.
- Add capability to handle dataset (and local and global constant) dimensions which are not known until run-time.
- Add recursive geometric partitioning for local renumbering of sets and mappings to improve data reuse.
- 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.
- Within a thread block, could reorder elements to reduce the number of different colors within each warp. This would require the storage and fetching of the permuted identity mapping
- Should use the `restrict` qualifier where appropriate to enable compiler optimization
- There is scope for relatively simple checkpointing. An **op_checkpoint** instruction in the user code would trigger the saving of the state of all of the OP2 datasets. In addition, all global reduction values generated by OP2 would be saved. In the event of an application failure, the application could be restarted and "fast-forwarded" to the last checkpoint at which point the state would be restored. The global reduction data is all that is needed for the fast-forward to proceed correctly, as this is the only data which is transferred from OP2 to the user's main application.
- Some of the block-invariant data which is currently initialised by thread 0 and put in shared memory could instead be pre-computing by the host, and then accessed through the constant cache.
- For 3D structured grids, we use a block "pencil" with a 3-plane active working set. It's possible something similar could be done here, with each thread block working on a sequence of mini-partitions, with some threads loading in the data for the next one while others are doing the computation for the current one. Testing this idea could be a lot of work.