Optimising the OP2 Framework for GPU Architectures

Mike Giles, Gihan Mudalige, Ben Spencer

mike.giles@maths.ox.ac.uk

Oxford e-Research Centre

MRSC 2011, April 12, 2011

Outline

- OP2 framework
- GPU optimisation
  - Array-of-Structs or Struct-of-Arrays?
  - auto-tuning
- Conclusions

Many-core hardware

For 10 years, 1995-2005, HPC was relatively simple:

- large clusters, with each node having 2 scalar CPUs
- MPI programming with FORTRAN / C / C++

Now things have become much more complicated:

- multi-core CPUs – up to 12 cores / 24 threads per CPU and each core has an increasingly large vector unit
- GPUs have up to 512 cores
- best programming approach unclear:
  - MPI + OpenMP, or MPI + ArBB for CPUs
  - CUDA for GPUs (and CPUs?)
  - OpenCL?

Software Challenges

- HPC application developers want the benefits of the latest hardware but are very worried about the software development costs, and the level of expertise required
- status quo is not an option – running 24 MPI processes on a single CPU would give very poor performance, plus we need to exploit the vector units
- For GPUs, we're happy with NVIDIA's CUDA (C with extensions) but like MPI it's too low-level for many
- For CPUs, MPI + OpenMP may be a good starting point, and PGI/CRAY are proposing OpenMP extensions which would support GPUs and vector units
- However, hardware is likely to change rapidly in next few years, and developers can not afford to keep changing their software implementation

Software Abstraction

To address these challenges, need to move to a suitable level of abstraction:

- separate the user's specification of the application from the details of the parallel implementation
- aim to achieve application level longevity with the top-level specification not changing for perhaps 10 years
- aim to achieve near-optimal performance through re-targetting the back-end implementation to different hardware and low-level software platforms

OP2

- framework aimed at unstructured grid applications
- user writes a code in C/C++ or FORTRAN, specifying
  - sets (e.g. nodes, edges, faces)
  - datasets (e.g. flow variables)
  - mappings (e.g. from edges to nodes)
  - parallel loops
- automated code generation produces efficient code for
  - GPUs (using CUDA or OpenCL)
  - many-core CPUs (using OpenMP + vectorisation)

OP2 API

Example of parallel loop syntax for a sparse matrix-vector product:

```
op_par_loop(res,"res", edges,
   A, -1,OP_ID, 1,"float",OP_READ,
   u,  0,pedge2,1,"float",OP_READ,
   du, 0,pedge1,1,"float",OP_INC);
```

This is equivalent to the C code:

```
for (e=0; e<nedges; e++)
   du[pedge1[e]] += A[e] * u[pedge2[e]];
```

where each “edge” corresponds to a non-zero element in the matrix `A`, and `pedge1` and `pedge2` give the corresponding row and column indices.

GPU Parallelisation

Could have up to 10^6 threads in 3 levels of parallelism:

- MPI distributed-memory parallelism (1-100)
  - one MPI process for each GPU
  - all sets partitioned across MPI processes, so each MPI process only holds its data (and halo)
- block parallelism (50-1000)
  - on each GPU, data is broken into mini-partitions, worked on separately and in parallel by different streaming multi-processors (SMs) in the GPU
- thread parallelism (32-128)
  - each mini-partition is worked on by a block of threads in parallel

AoS or SoA?

One key implementation decision is how to store datasets in which there are several data elements for each set element (e.g. 4 flow variables at each grid point)

- Array-of-Structs (AoS) approach views the 4 flow variables as a contiguous item, and holds an array of these

0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3

- Struct-of-Arrays (SoA) approach has a separate array for each one of the data elements

0	0	0	0	0	1	1	1	1	1	2	2	2	2	2	3	3	3	3	3

AoS or SoA?

The SoA approach is natural for streaming hardware, like old CRAY vector supercomputers

- memory sub-system designed to stream long vectors of data from memory to compute units and back again
- many think GPUs are modern descendents, and hence SoA is natural choice
- very suitable for structured grid applications as neighbouring grid points are worked on one after the other
- . . . but what about unstructured grids?

  CRAY systems had special gather/scatter hardware support – GPUs don't

AoS or SoA?

The AoS approach is natural for conventional CPUs

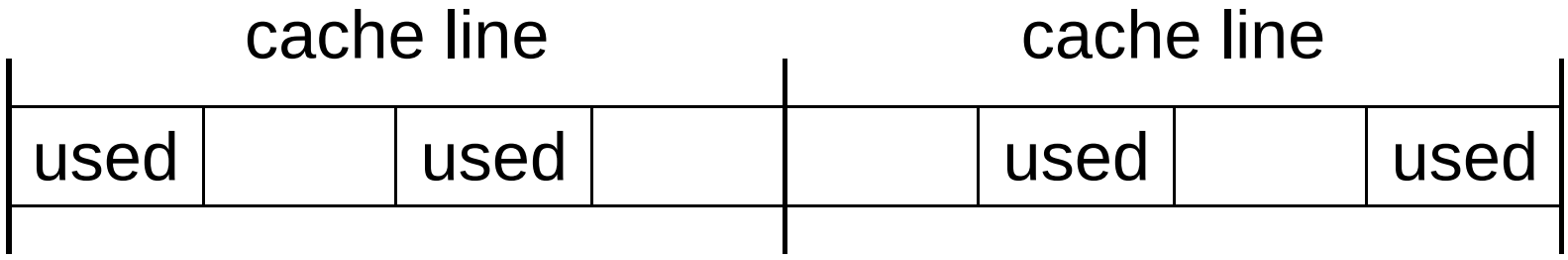
- only a few active virtual pages at a time

  (20 years ago, SoA approach was 10 times slower on an IBM RS/6000 system due to number of active pages and limited size of TLB – Translation Lookaside Buffer)

- provided all of the local elements are used, cache utilisation is good
- NVIDIA Fermi-based GPUs have L1 / L2 caches, so AoS is natural approach?

AoS or SoA?

For GPUs, key is cache utilisation:



- 1 float elements in a 128 byte cache line is equivalent to 4 float elements in a 512 byte cache line

  $\Longrightarrow$ SoA approach has bigger effective cache line, so less efficient for unstructured grid applications

- . . . but this assumes all the data at each point is needed

AoS or SoA?

What about coalesced memory transfers?

Not as important for Fermi GPUs as previous generation, but can still be achieved for simple loops by careful programming using shared memory:

```
float arg_l[4];                    % register array
__shared__ float arg_s[4*32];  % shared memory

for (int m=0; m<4; m++)
  arg_s[tid+m*32] = arg_d[tid+m*32];

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

By using a separate “scratchpad” for each warp, can generalise this without needing thread synchronisation

Auto-tuning

In the CUDA implementation there are various parameters and settings which apply to the whole code:

- compiler flags, such as whether to use L1 caching
- (whether to use AoS or SoA storage for each dataset)

and others which can be different for each CUDA kernel:

- number of threads in a thread block
- size of each mini-partition
- (whether to use a 16/48 or 48/16 split for the L1 cache / shared memory)

Auto-tuning

In each case, the optimum choice / value is not obvious, but it is possible to

- give a small set of possible values for each (usually two or three)
- state which can be optimised independently (e.g. the parameters for one kernel don't affect the execution of another kernel)

What is then needed is a flexible auto-tuning system to select the optimum combination by exhaustive "brute force" search.

The parameter independence is essential to making this viable.

Auto-tuning

A flexible auto-tuning package has been developed:

- written in Python
- input specification includes
  - parameters and possible values
  - a mechanism to compile the code, perhaps using some of the parameter values
  - a mechanism to run the code, again perhaps using some of the parameter values
  - by default, the run-time is used as the “figure-of-merit” to be optimised
- at present only brute-force optimisation is supported, but in the future other strategies may be included

Auto-tuning

Example configuration file:

```
#
# parameters and values
#
PARAMS = { flag, {block0, part0}, {block1, part1} }

flag   = {"-Xptxas -dlcm=ca", "-Xptxas -dlcm=cg" }  # compiler flag
block0 = {64, 96, 128}   # thread block size for loop 0
part0  = {128, 192, 256} # partition size for loop 0
block1 = {64, 96, 128}   # thread block size for loop 1
part1  = {128, 192, 256} # partition size for loop 1

#
# compilation and evaluation mechanisms
#
COMPILER = make -B flag=%flag% block0=%block0% part0=%part0%
                               block1=%block1% part1=%part1%
EVALUATION = ./executable
```

Airfoil test code

- 2D Euler equations, cell-centred finite volume method with scalar dissipation (miminal compute per memory reference – should consider switching to more compute-intensive “characteristic” smoothing more representative of real applications)
- roughly 1.5M edges, 0.75M cells
- 5 parallel loops:
  - `save_soln` (direct over cells)
  - `adt_calc` (indirect over cells)
  - `res_calc` (indirect over edges)
  - `bres_calc` (indirect over boundary edges)
  - `update` (direct over cells with RMS reduction)

Airfoil test code

Single precision performance for 1000 iterations on an NVIDIA C2070 using initial parameter values:

- mini-partition size (PS): 256 elements
- blocksize (BS): 256 threads

count	time	GB/s	GB/s	kernel name
1000	0.23	107.8		save_soln
2000	1.26	61.0	63.1	adt_calc
2000	5.10	32.5	53.4	res_calc
2000	0.11	4.8	18.4	bres_calc
2000	1.07	110.6		update
TOTAL	7.78			

Second B/W column includes whole cache line

Airfoil test code

Single precision performance for 1000 iterations on an NVIDIA C2070 using auto-tuned values:

count	time	GB/s	GB/s	kernel name	PS	BS
1000	0.22	101.8		save_soln		512
2000	1.09	74.1	75.4	adt_calc	256	128
2000	4.95	36.9	60.6	res_calc	128	128
2000	0.10	5.3	20.0	bres_calc	64	128
2000	1.03	94.7		update		64
TOTAL	7.40					

This is a 5 % improvement relative to baseline calculation. Switching from AoS to SoA storage would increase `res_calc` data transfer by approximately 120%.

Airfoil test code

Double precision performance for 1000 iterations on an NVIDIA C2070 using auto-tuned values:

count	time	GB/s	GB/s	kernel name	PS	BS
1000	0.44	104.9		save_soln		512
2000	2.62	52.9	53.8	adt_calc	256	128
2000	10.35	30.5	50.8	res_calc	128	128
2000	0.08	11.2	27.9	bres_calc	64	128
2000	1.87	104.5		update		64
TOTAL	15.36					

This is a 7.5 % improvement relative to baseline calculation. Switching from AoS to SoA storage would again increase `res_calc` data transfer by approximately 120%.

Airfoil test code

Single precision performance on two Intel “Westmere” 6-core 2.67GHz X5650 CPUs using auto-tuned values:

Optimum number of OpenMP threads: 16

count	time	GB/s	GB/s	kernel name	PS
1000	1.68	13.7		save_soln	
2000	11.15	7.3	7.5	adt_calc	128
2000	16.57	10.3	11.2	res_calc	1024
2000	0.16	3.2	11.9	bres_calc	64
2000	4.67	20.9		update	
TOTAL	34.25				

Minimal gain relative to baseline calculation with 12 threads and mini-partition sizes of 1024.

Airfoil test code

Double precision performance on two Intel “Westmere” 6-core 2.67GHz X5650 CPUs using auto-tuned values:

Optimum number of OpenMP threads: 12

count	time	GB/s	GB/s	kernel name	PS
1000	2.51	18.3		save_soln	
2000	11.68	11.8	11.9	adt_calc	1024
2000	20.99	12.8	13.5	res_calc	1024
2000	0.17	5.0	12.4	bres_calc	512
2000	9.29	21.1		update	
TOTAL	44.64				

Minimal gain relative to baseline calculation with 12 threads and mini-partition sizes of 1024.

Conclusions

- have created a high-level framework for parallel execution of unstructured grid algorithms on GPUs and other many-core architectures
- looks encouraging for providing ease-of-use, high performance and longevity through new back-ends
- auto-tuning is useful for code optimisation, and a new flexible auto-tuning system has been developed
- C2070 GPU speedup versus two 6-core Westmere CPUs is roughly $5\times$ in single precision, $3\times$ in double precision
- currently working on MPI layer in OP2 for computing on GPU clusters
- key challenge then is to build user community

Acknowledgements

- Carlo Bertolli, David Ham, Paul Kelly, Graham Markall, Florian Rathgeber (Imperial College)
- Nick Hills (Surrey) and Paul Crumpton
- Leigh Lapworth, Yoon Ho, David Radford (Rolls-Royce)
- Tom Bradley, Jon Cohen and others (NVIDIA)
- EPSRC, TSB, NVIDIA, Rolls-Royce and Oxford Martin Institute for financial support
- Oxford Supercomputing Centre