Some reflections on automated code generation

Mike Giles

Gihan Mudalige, Istvan Reguly and others

Oxford e-Research Centre

NAIS Workshop

July 1, 2014

Outline

- Motivation
- Approaches
- 5 Oxford code generation projects (2009 – now)
  - OP2
  - StochSimGPU
  - HMMingbird
  - OPS
  - Computational finance

Details of all of the projects are available online – I will concentrate on the main ideas, some of the pros and cons, and some of the differences from other research in the literature.

Motivation

I come from engineering/maths application domains, so I approach this subject from the perspective of a "user" who wants two things:

- high performance on a range of different hardware, today and tomorrow (up to 10 years?) – "future proofing" is a key buzzword these days
- ease of software development – i.e. high performance without needing to become an expert in new technologies

DARPA HPC^2:

High Performance Computing – High Productivity Computing

Performance

Good news:

- performance continues to double every 18 months or so

Bad news:

- increased performance is coming through massive parallelism
- increasing variety in the high-end computing hardware platforms
- also increasing diversity in the corresponding software platforms:
  - CPUs: OpenMP 4.0, MPI, TBB, Cilk Plus, OpenCL, vector intrinsics
  - GPUs: CUDA, OpenCL, OpenACC

Our approach

Our approach is motivated by applications:

- first, work out how to achieve good performance on a variety of hardware platforms
  - this can lead to demo codes which are helpful to others as an illustration of various tips/ "tricks" /techniques
  - often involves careful consideration of data placement and movement – often the main performance bottleneck these days, and can require different treatments on different platforms
- second, think about how we can deliver this performance to application developers, without them needing to become HPC experts
  - this can lead to the development of either library software or a new code generation project

Libraries

For relatively simple well-defined tasks, libraries are often (usually?) the best way of delivering high performance and simplicity to application developers.

In Oxford, we have contributed to NVIDIA's GPU libraries:

- random number generation and associated special functions (e.g. inverse error function)
- sparse matrix-vector multiplication
- batched tridiagonal solvers

Libraries

For larger, less well-defined tasks, libraries are less appropriate:

- unable to hide complexities of GPU programming and still achieve good performance?
- unable to hide complexities of distributed-memory programming?
- performance suffers from conditional branching needed to handle lots of different capabilities?
- performance suffers from (lots of) dynamic memory allocation since array sizes not known *a priori*?

Alternatives?

The US DoE labs use C++ meta-programming extensively

- I don't know/understand the details
- my impression is that the approach is very powerful
- they're clearly happy with the application performance and ease-of-use
- I've heard that debugging the meta-programming is very difficult – errors generate strange cryptic compiler error messages which are very hard to interpret
- my impression is that it needs guru-standard C++ developers to develop the meta-programming

Alternatives?

The other big alternative is automated code generation – the reason we are here today.

This can be sub-divided into three approaches:

- DSL (domain-specific languages)
  - defines a "new" language for a specific class of applications
    — will application developers be happy to use a new language?
    — what about porting existing applications?
  - usually built on an open-source compiler system (e.g. LLVM) or perhaps a special-purpose underlying language (e.g. Scala)
    — this third-party dependence might be viewed as a weakness
  - usually proposed by computer scientists with compiler expertise; application scientists may lack the knowledge to develop this approach

Alternatives?

- DSA (domain-specific abstraction/API)
  - also referred to as an embedded-DSL, or high-level framework
  - defines new capabilities within an existing language
  - could be either an API or pragmas/directives in code
  - does not necessarily require parsing of the entire user code, so does not necessarily require an open-source compiler such as LLVM
  - a viable approach for both application and computer scientists
- dynamically-generated libraries
  - takes a specification (in what format?) and produces code to perform some task
  - a viable approach for both application and computer scientists

Our approach

First, we start with a particular motivating application and determine

- the target set of hardware platforms (usually include GPUs)
- how to achieve good performance on these using hand-coded implementations

Next, we think about an easy-to-use generalisation:

- library or DSA?
- API design?
- user-supplied specification file, or parsing (some of) the user's code?
- language for the code generator (e.g. MATLAB, Python, C/C++) ?
- any benefits from run-time JIT compilation?

Will address these points in discussing 5 projects over the past 5 years.

OP2

Joint project with Paul Kelly and David Ham at Imperial College, with funding from Rolls-Royce and EPSRC

- aimed at finite volume / finite element applications on unstructured grids
- an update to the Oxford OPlus library for Rolls-Royce's HYDRA CFD code
- OPlus was a classic library developed in 1995 for distributed-memory clusters with single core processors
- OP2 was designed to keep distributed-memory support, and add support for GPUs and multi-core CPUs (and maybe Xeon Phi) as basis for HYDRA for next decade
- the new capabilities required code generation, and a new API
- also added support for both FORTRAN90 and C++

OP2 Abstraction

- sets (e.g. nodes, edges, faces)
- datasets (e.g. flow variables)
- mappings (e.g. from edges to nodes)
- parallel loops
  - operate over all members of one set
  - datasets have at most one level of indirection
  - user specifies how data is used
    (e.g. read-only, write-only, increment)

Restrictions:

- set elements can be processed in any order, doesn't affect result to machine precision
  - explicit time-marching, or multigrid with an explicit smoother is OK
  - Gauss-Seidel or ILU preconditioning is not
- static sets and mappings (no dynamic grid adaptation)

OP2 API

```
void op_init(int argc, char **argv)

op_set op_decl_set(int size, char *name)

op_map op_decl_map(op_set from, op_set to,
                   int dim, int *imap, char *name)

op_dat op_decl_dat(op_set set, int dim,
                   char *type, T *dat, char *name)

void op_decl_const(int dim, char *type, T *dat)

void op_exit()
```

OP2 API

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

```
op_par_loop(res,"res", edges,
  op_arg_dat(A,-1,OP_ID, 1,"float",OP_READ),
  op_arg_dat(u, 0,col,1,"float",OP_READ),
  op_arg_dat(du,0,row,1,"float",OP_INC));
```

This is equivalent to the C code:

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

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

OP2

Notes:

- the parallelism is inherent in the specification of the parallel loops
- the data handles are opaque to the user (can't be touched/used) and the loop specification says how the data is being used
- OP2 has complete freedom in how the data is distributed across multiple nodes, and how it is stored (SoA or AoS) – can be stored differently on different platforms, with OP2 doing the required data transformations internally
- data access information enables OP2 to decide when "halo" data needs to be exchanged

OP2

OP2's execution has two components:

- user code for "command and control"
- OP2 execution of parallel loops

This gives us a lot of flexibility which we are not yet fully exploiting:

- automatic checkpointing – since OP2 knows about all of the principal distributed datasets, and how they are used/changed, it can decide what to store, and when
- normally, parallel loops are executed one after the other, but OP2 has the flexibility to use "lazy execution", record the fact that certain loop calculations have to be performed, but do them later – opens up new optimisation possibilities, such as tiling

OP2

- I developed initial GPU implementation by hand for a simple test application (2D airfoil code)
- for development/debugging purposes, developed a header file which can be used for single-threaded execution without code generation
- Paul Kelly is a computer scientist, so plan was for his group to develop a compiler-based code generator using third-party ROSE system
- however, I simplified the API design to the point where I was able to implement a simple prototype in MATLAB for C/C++ applications — key was that only the API calls had to be parsed
- the ROSE-based code generator processed the entire user code, which enabled the API to be simplified – however, it could not cope with complexity of code in HYDRA, so we dropped it
- instead, we switched from MATLAB to Python – a better open-source solution, especially for companies like Rolls-Royce

OP2

Final status:

- Python-based generator handles all of HYDRA, only parsing the API calls
- GPU performance is great for a 2D tsunami code from UCL, but not so good for HYDRA which uses too many registers
- OpenMP backend works well for HYDRA
- still more work to do on CPU vectorisation
- the distributed-memory MPI layer works on top of any of the low-level backends
- `http://www.oerc.ox.ac.uk/projects/op2`

- Paul & David have developed a Python variant, PyOP2, for FEniCS, a flexible FE solver written in Python — this exploits JIT for a cleaner API

StochSimGPU

Part of PhD project in Maths – Guido Klingbeil:

- `http://people.maths.ox.ac.uk/klingbeil/STOCHSIMGPU/`
- Guido had combined maths/computer science background
- very specific application area – stochastic biochemical reaction simulations
- motivated by desire for GPU performance and MATLAB interface for ease-of-use for users

StochSimGPU

Approach:

- API is a simple MATLAB function which specifies the reactants and reactions
- MATLAB code then
  - generates CUDA code
  - compiles it
  - executes it
  - gets the output for further MATLAB processing
- all of the internal operation is hidden from the user
- all of this was fairly straightforward to accomplish – main effort was in the initial hand-coding to achieve the best GPU performance

StochSimGPU

Benefits over a classic library approach:

- in a classic library, the code would loop over an arbitrary number of reactions, each involving an arbitrary number of reactants specified through indirect addressing
- with code generation, the generated code handles each reaction individually, with almost no conditional code
- great for GPU execution – static allocation of all variables in registers, and almost no conditional branching
- Guido's found a $10\times$ improvement in performance

HMMingbird

PhD project in Computer Science by Luke Cartey
(supervised by Oege de Moor):

- `http://www.hmmingbird.co.uk/`
- Hidden Markov Model compiler for bioinformatics applications
- requirements developed with users in Statistics
- approach was to construct a simple DSL
- polyhedral code generator written in ClooG (`http://www.cloog.org/`) within Java
- good performance achieved for a few applications, compared to other single-application codes

OPS

New EPSRC-funded project
(`http://www.oerc.ox.ac.uk/projects/ops`)

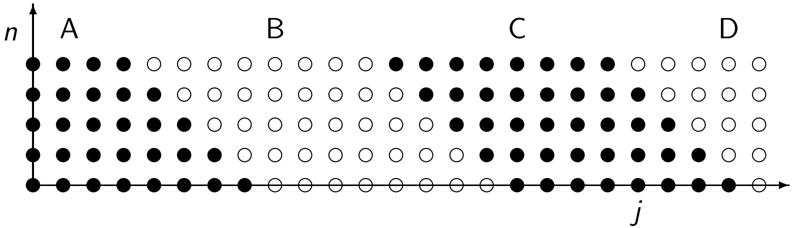
- similar to OP2, but extending the domain to block-structured grids
- each grid block is structured (typically 2D or 3D), but blocks have fairly arbitrary connectivity
- approach and methodology is essentially the same as OP2
- initial work has achieved great performance for a AWE test code, Cloverleaf – a single simple user code achieves performance comparable to a number of hand-coded versions developed by others
- one key future objective is tiling through lazy execution

Tiling

Consider explicit time-marching: $u_j^{n+1} = a\, u_{j-1}^n + b\, u_j^n + c\, u_{j+1}^n$

Usually, start with $n=0$, then do $n=1$, then $n=2$, etc

Instead can do whole "tiles" at a time – much greater cache re-use:



Potentially large savings in the future, as memory bandwidth is increasingly the bottleneck. Coding this manually would be painful – we think automating it through a DSL is preferable to using compiler techniques.

New computational finance project

Financial modelling:

- multi-dimensional SDEs (stochastic differential equations)
  - Monte Carlo simulation with lots of independent paths
  - naturally-parallel application, executes well on GPUs using NVIDIA random number generation library
- multi-dimensional PDEs (typically dimension 1-4)
  - Monte Carlo simulation with lots of independent paths
  - either explicit or implicit time-marching
  - implicit uses ADI splitting, involving solution of tridiagonal equations in each coordinate direction
  - over last year, have worked on very efficient implementations on GPUs and CPUs
  - `http://people.maths.ox.ac.uk/gilesm/codes/BS_1D/`
    `http://people.maths.ox.ac.uk/gilesm/codes/BS_3D/`

New computational finance project

Code generation:

- new project by Saurabh Pethe, a visiting student from IISc Bombay
- background in material science, with a little knowledge of Python and CUDA
- wrote code generator in Python to generate 1D code with lots of options (e.g. uniform grid or grid supplied, same or different volatility for each financial derivative, . . .)
- generates CUDA code for GPUs, or OpenMP/vector code for CPUs
- specification taken from an XML file – Python package
- alternatively, comes from a simple GUI – Python package

Reflections

After 5 years, what do I think about code generation?

- code generation is not difficult – any computational scientist can do it
- hard thing is code parsing/analysis – I would avoid it as far as possible
- code generation can do things which regular libraries can't – often essential for new architectures
- with a general package, can justify putting extra effort into difficult optimisations and extensive error-checking
- Python is now my default choice for writing a generator, but you can really use whatever you are comfortable with
- in the future, may want to move to JIT compilation
- I think DSLs/DSAs have proven their potential to provide a future-proof approach for application programmers
- one concern is long-term support – who will fund this?