A DSL for Hardware-accelerated Grid-based Scientific Models

Alexander N. Gavrilov

R&D Department
Lanit-Tercom, Inc.
Saint-Petersburg, Russia
E-mail: Alexander.Gavrilov@lanit-tercom.com

***Abstract*—This paper presents an ongoing effort to develop a domain specific language that would simplify exploiting hardware acceleration in grid-based scientific models. The project is being implemented in Common Lisp, which has the best built-in metaprogramming capabilities among industrial-level compiled programming languages. The DSL is applied to a hydrodynamic model of the atmosphere.**

***Keywords*-GPU computing; SSE; lisp; metaprogramming; DSL; simulation**

I. Introduction

Grid-based physics simulations are among the most easily parallelizable programming tasks, belonging to the category of so called embarrassingly parallel applications. They also are very close to the kind of calculations that modern GPU chips are designed to perform, and therefore should benefit greatly from using them.

Unfortunately, currently available free tools for programming NVidia GPUs are basically limited to the official C compiler [1] and accompanying libraries distributed by NVidia Corporation. Moreover, before PGI released a GPU-capable version of their commercial Fortran compiler [2] near the end of 2009, that was the only available tool at all.

The C-based tools are designed to provide very close control of the GPU, and thus require the programmer to manually manage GPU memory allocation, data alignment, textures and other low-level features of the platform. This makes GPU programming quite difficult to approach for people who are not professional system programmers.

Moreover, with existing tools it is still impossible to use the same code for both CPU and GPU, and fully exploiting the high performance computing features of modern CPUs in complex cases requires manual use of even more low-level SSE intrinsic functions. Which cases are to be considered complex depends on the capabilities of the compiler, but these usually include any loops that contain conditionals – even though the actual SSE instruction set is flexible enough to be able to express them[1].

All these issues were encountered during an attempt to speed up execution of an atmospheric model described in [3], which resulted in the creation of the DSL.

[1]This however requires dropping the lazy evaluation property of conditionals, i.e. works by evaluating both branches and then choosing which result to use.

II. Related Work

In addition to the official C toolkit, there exist a number of GPU programming libraries for other programming languages like Java[4] or Python[5]. However, they mostly only wrap the run-time library that is used to manage GPU memory, load code to the GPU and execute it. The actual GPU code still has to be written in C.

These wrappers already remove a large part of the burden and make GPU programming a lot more accessible, as user reports indicate. However, as noted above, due to the lack of native meta-programming support in the mentioned programming languages it is impossible to achieve seamless integration. The Python version goes further than the Java wrapper and allows C code to be expressed as a data structure tree instead of strings, but that is the limit of what can be done. This also means that these languages are not a very good choice for implementing a new DSL.

There are also some existing differential equation solver libraries, e.g. the OpenCurrent[6] C++ library created by a member of NVidia Research, but they are quite irrelevant to the task of implementing a specific unique calculation.

III. Methodology

A. Initial problem

The project started as an attempt to optimize a hydrodynamic model [3] of the atmosphere. The initial program was written in Fortran; Waterloo Maple was used to derive the mathematical expressions and generate most of the code.

As follows from their name, grid-based simulations are based on a representation of the state of the world produced by sampling the relevant physical quantities at regular space intervals. These samples are then grouped into 2D or 3D arrays depending on the model, and recalculated iteratively using a set of mathematical formulae as in-model time progresses.

Since most of the time the value of a cell being calculated depends only on array cells within a fixed nearby area, these calculations can be easily parallelized along array axes. Some models however improve computational stability by employing additional smoothing schemes that are serial along one of the axes; this is the case for the model in question. These schemes are an obstacle

to achieving the best performance and may result in a bottleneck on highly parallel architectures like the GPU.

Another major source of complications seen in this model is the use of a skewed grid, i.e. a scheme where physical coordinate grids used to sample different quantities are displaced in relation to each other by a certain fraction of the grid step (normally 1/2). This can be used to noticeably improve the precision of the model, but results in a certain technical difficulty:

The most straightforward and natural way for a human to handle a skewed grid scheme is to actually allocate a common grid with twice as many indexes, and use a sparse sub-grid for every specific quantity; any other way is very error-prone. However this naive approach wastes half of the memory bandwidth and prevents the use of some optimizations with strict memory positioning requirements, e.g. Intel SSE instructions.

The skewed grid issue is most naturally solved via automated transformation, i.e. meta-programming.

B. Implementation language

ANSI Common Lisp[7] is a dynamically typed multi-paradigm compilable programming language that is defined by an ANSI standard [8] finalized in 1994, and has multiple independent implementations. It allows writing code in functional, imperative or object-oriented style.

Like all languages of the lisp family, Common Lisp provides excellent meta-programming support via macros, which, unlike the identically called feature of the C preprocessor, are actually full-featured Lisp functions that are configured to be automatically called by the compiler, and operate on syntax trees. Macros can do anything that ordinary functions can do, including operations like creating temporary files and calling external programs.

The existence of macros transforms arbitrary meta-programming from something that usually requires modifying the compiler via patching or plug-ins, or implementing external preprocessors that operate on raw program text, into a relatively mundane task that is done to some extent by every proficient lisp programmer.

On the other end of the abstraction spectrum, almost every Common Lisp implementation provides facilities that can be used to access raw memory from lisp code and directly call external C libraries. One notable implementation in this regard is Embeddable Common Lisp([9], [10]), which compiles all lisp code by transforming it into C and calling the system's C compiler. This allows it to provide support for using C directly from lisp code, much like C compilers and inline assembler.

A couple of widely accepted utility libraries (e.g. [11]) can be used to call C functions from lisp code in an implementation-independent way. When the developers of the C library take care to maintain binary compatibility (as is the case with NVidia drivers), this makes the development and build process a lot more convenient by removing the need for any kind of wrappers written in C.

C. Target platform

At the hardware level modern GPU systems produced by NVidia are multi-core SIMD processors capable of executing divergent code[12], with native support for multi-threading and a hardware scheduler. Processor cores also include special units for texture fetching and interpolation, limited caches for textures & constants, and on-chip shared memory buffers for inter-thread communication. Ordinary memory was not cached until the recently announced Fermi architecture[2].

This hardware is exploited through a programming model that involves launching a large grid of threads, every one of which executes the same function to process one small chunk of input data. The threads are not required to follow the same execution path, but unless the pattern of branch divergence is matched to the underlying SIMD hardware it results in heavily reduced performance.

More specifically, the thread grid is partitioned into identical blocks of up to 1024 threads. Threads within one block can communicate via shared memory and a barrier primitive; different blocks are independent and may be scheduled by hardware in any order.

When a block starts execution, it is assigned to a processor core and gets a slice of its shared memory buffer and register pool[3]. These memory and register requirements determine how many blocks can run on one core at once. The threads of all running blocks are executed by the core's pre-emptive scheduler in static groups of 32 threads, which are officially called warps.

If all threads of a warp execute the same instruction, it is executed simultaneously in SIMD fashion; divergence forces the scheduler to serialize processing by temporarily disabling SIMD units for threads that don't need the instruction that it is going to execute.

In a similar fashion, if all threads of a warp access adjacent addresses in global memory, the scheduler can coalesce these operations into one large memory transaction, thus reducing the overhead caused by the lack of a traditional memory cache. Memory access transaction coalescing is heavily affected by data alignment.

GPU code cannot allocate or free memory, call functions by pointer or use recursive functions. GPU memory cannot be directly accessed from host code, but modern video cards allow mapping specially allocated regions of host memory into GPU address space.

D. Current state of the DSL syntax

This work on rewriting the original program has led to the creation of the following DSL:

1) Infix formula syntax: Common Lisp normally uses identical prefix syntax for function invocation and arithmetic operations. This is necessary for achieving the existing degree of meta-programming support, and tolerable

[2]Since Fermi-based cards are new technology that was not available for purchase at the moment of writing, this paper does not take that architecture into account.

[3]Each core has 16KB of shared memory, and, depending on the GPU version, 8192 or 16384 32-bit registers

in ordinary code, but not at all convenient for expressing huge mathematical expressions that are often used in the problem domain.

Fortunately, this can be fixed using the extensibility of the lisp reader (i.e. low-level code parser). A straightforward add-on with a simple expression parser allows writing infix expressions delimited by curly brackets:

```
{ NEW_DT[i,MW+1] := (TMP_ANU[MW-1]*TMP_EPS[MW+1]
                     +TMP_ANU[MW+1])
                    / (1.0-TMP_EPS[MW+1]*TMP_EPS[MW-1]) }
```

The parser recognizes arithmetic operators, array references, function calls and assignments. The syntax reflects the one used by Maple and many other computer algebra systems.

2) Virtual arrays: As a way to solve the skewed grid issue outlined above, the system is based on virtual arrays that have separate logical and physical dimension structure.

Dimensions may be arbitrarily reordered, and any particular dimension can be configured to either contain regular gaps, or distribute consecutive logical indexes along a hidden fixed dimension. Either of these special modes achieves contiguous arrangement in memory for logical indexes that are separated by a specific fixed stride.

```
(def-multivalue DR ((i 1 N1) :by 2) (k 1 MW :by 2)))
(def-multivalue PL ((i 1 N1) (k 1 {MW+1} :bands 2)))
```

Virtual arrays are accessed via an *iref* macro, which is similar to the standard *aref* function, but applies appropriate static transformations to its index arguments. The infix expression parser can be configured to generate either of those as a representation of its array reference syntax.

3) Virtual array index loops: A useful macro allows iterating over logical dimensions of a virtual array in a way that reflects the underlying physical structure.

```
(do-indexes OUT_U (i k)
  (format vel (formatter "~12,3E ~12,3E ~14,5E ~14,5E~%")
          (iref xcoord i) (iref zcoord k)
          (iref OUT_U i k) (iref OUT_V i k)))
```

The actual loop variables in the macro expansion correspond to physical dimension indexes, while the public logical indexes are expanded to expressions based on them. When said logical index variables are used as arguments to the iref macro within the loop body, the expressions are automatically simplified, and the resulting code appears as if it was written directly for the physical structure.

The order of index names in the loop header (which must match the names used in the virtual array definition) determines the loop nesting order. The iteration ranges are derived from the array definition. Extended syntax can be used to specify iteration direction and stepping, or reduce its range.

4) The compute statement: Finally, main code translation features of the library are encapsulated in the compute statement, which allows one to define a complete computation, composed of a target array, index range limits, iteration directions, and a mathematical expression to compute.

```
(compute PL ((i :skip (1 1) :step (* 2))
             (k :step (* 2)))
  { t3*t10/2.+2.*t1*_grp(t2/t3)*t12 }
  :parallel i
  :with { t1 := PL[i+1,k];
          ... })
```

Every such statement is converted to a separate GPU invocation. Additional clauses can be used to specify which index to focus parallelization efforts on, define inter-value dependencies, or pass additional optimization hints to the GPU compiler.

If the code translator fails to handle some parts of the expression, the statement is expanded into ordinary lisp loops. This ensures graceful integration of the functionality in the language, and, very importantly, allowed the optimized code generator to be implemented gradually *after* the program was fully converted from Fortran to Lisp.

E. Code optimizations

In order to ensure good performance of the generated code, the compute statement applies a number of optimization techniques. They can be classified into general expression restructuring and hardware-specific modifications.

1) Expression restructuring: The following generic simplifications are applied to the computation:

- Nested trees of associative-commutative operators are flattened and reordered to reveal the internal structure. This can be inhibited where needed[4] using a special _grp(...) syntax, which works like unbreakable parentheses.
- Very basic arithmetic expression simplification is performed.
- Branch-on-sign[5] expressions are simplified using hints about value signs supplied by the developer.
- Common sub-expression elimination with loop-invariant value extraction is performed. CSE exploits associativity of arithmetic operations.

2) Efficient GPU memory usage: The GPU has several ways to store and access data with varying characteristics. In order to achieve good performance it is necessary to use the best method for the task. For instance:

- The code generator supports using texture fetches to access arrays:
  Before Fermi, textures were the only way to achieve cached access to memory. In C using them requires rewriting all statements that access the data, but the generation-based approach reduces the complexity to a simple hint clause with a list of arrays that should be cached.
- Values that are the same for all threads of a block are stored in shared memory:

[4]Since floating-point calculations are not associative, automated reordering sometimes leads to severe precision degradation and situations like 0/0.

[5]The program that was used as a base for the experiment uses many expressions like (ifsign x 0.0 0.5 1.0). The branch value argument often contains constants and other sub-expressions that don't affect the sign.

Due to rather simple data access patterns combined with complex computations being common in the problem domain, shared memory is generally underused and thus cheaper than registers. Using it when appropriate increases the number of blocks that can run in parallel.

3) Serial inner loop representation: Due to memory coalescing requirements, the innermost loop must always be mapped to an in-block thread grid. If it has a serial data dependency, this results in a performance problem.

First of all, this means that the whole of the inner loop range must be handled within one block in order to allow data exchange through shared memory. This reduces the number of independent blocks.

The second issue is that the data-dependent part must still be executed serially using inter-thread synchronization barriers. The only way to improve performance here is to move as much code as possible out of this part. This especially applies to global memory access expressions.

In order to handle this case, the code generator includes special dependency analysis and code extraction functionality.

4) Operation reordering: Reducing the number of registers needed by every thread is instrumental in increasing the degree of parallelism and hiding memory latency. Unfortunately, optimal expression DAG reordering under register pressure is proven to be a NP-complete problem ([13], [14]), so no compiler can be expected to produce the best possible code in all cases.

The code generator applies a simple ordering heuristic that seems to help the official NVidia compiler produce better register mappings.

IV. Results

The above DSL was used to produce both SSE-ified CPU code (integrated via ECL's inline C feature), and GPU kernels. The CPU code, including the original program, was tested on an Intel Core 2 Quad 2.66GHz CPU[6], while the GPU version was executed using an NVidia GeForce GTX 275 graphics card.

The SSE version showed a 14x speedup against the original Fortran+Open-MP version. This can be explained in the following way:

- Expected 4x speedup due to the use of single-precision floating-point SIMD instructions.
- 40% speedup was observed after ifsign expression simplification was implemented.
- 2.5x speedup probably can be explained by better cache usage behavior, removal of function calls from inner loops, etc.

The GPU version was observed to be about 5 times faster than SSE code run on 3 of 4 cores. This number slowly grows as the array dimensions are increased; this is expected due to effects like diminishing utility of the CPU's caches.

[6]Only 3 cores were used for measurement to reduce the impact of other processes running on the same computer.

The raw peak FLOPS characteristics of the GPU allow one to expect a 26x maximum speedup. However, these numbers are computed under assumption that the MADD/-MUL double-issue feature is active; when it is not (and no code consists only of those two instructions), the expected speedup decreases to somewhere around 8-10x.

GPU code profiling and some calculations show that serialized inner loops are a noticeable local bottleneck in the current version.

V. Future work

These results show that the approach works for this particular model. The next major goal is to make the system more easily usable for implementation of similar programs, while also improving its performance. This requires fixing some shortcomings in the current implementation of the library.

A. New code generation back-end

The low-level C code generation back-end has been implemented in a hurry, and is too inflexible. This has become an obstacle in exploring better representations of serialized inner loops and other features.

This is being addressed by means of implementing a new, more generic, and separately usable GPU interface library for Lisp. The end result should be more or less similar to the existing bindings for Java and Python, but allow writing GPU code using normal Common Lisp syntax and a subset of the standard library with appropriate restrictions and extensions. Of course, the set of supported data types and operations will (at least initially) reflect the needs of this project.

The core part of the new library already works, but many specific features like textures are missing.

B. New DSL syntax

The syntax outlined above is both too general for the problem domain, and too specific in some technical matters like inter-iteration dependencies.

This prevents easily implementing features like non-SMP parallelism (e.g. multiple GPUs or clusters), and makes the code difficult to write for people who are not acquainted with the implementation of the library.

For example, it doesn't make much sense to declare different physical index orderings for different arrays. The array dimensions correspond to physical spatial dimensions, and it is more natural to select a global arrangement for them. Neither is it necessary to allow transposing indexes like this:

```
(compute foo (i j) { bar[j,i] })
```

Explicitly disallowing this would enable adding more code generation features.

One additional possibility is integrating with the Maxima[15] computer algebra system, which is also implemented in Common Lisp. Things like ifsign simplification more properly belong in the context of a CAS.

REFERENCES

[1] "CUDA Downloads," NVIDIA Corporation, 2010. [Online]. Available: http://developer.nvidia.com/object/cuda_download.html

[2] "PGI CUDA Fortran Compiler," The Portland Group, 2009. [Online]. Available: http://www.pgroup.com/resources/cudafortran.htm

[3] S. Kshevetskii and N. Gavrilov, "Vertical propagation, breaking and effects of nonlinear gravity waves in the atmosphere," *Journal of atmospheric and solar-terrestrial physics*, vol. 67, no. 11, pp. 1014–1030, 2005.

[4] "Java bindings for CUDA." [Online]. Available: http://www.jcuda.org/

[5] A. Klockner, "PyCUDA," Brown University. [Online]. Available: http://mathema.tician.de/software/pycuda

[6] J. Cohen, "OpenCurrent," NVidia Research. [Online]. Available: http://code.google.com/p/opencurrent/

[7] K. Pitman, *Common Lisp HyperSpec*, LispWorks, 1996. [Online]. Available: http://www.lispworks.com/documentation/common-lisp.html

[8] American National Standards Institute and Information Technology Industry Council, *American National Standard for information technology: programming language — Common LISP: ANSI X3.226-1994*. pub-ANSI:adr: pub-ANSI, 1996.

[9] G. Attardi, "The embeddable common Lisp," in *Papers of the fourth international conference on LISP users and vendors*. ACM, 1994, p. 41.

[10] J. J. Garcia-Ripoll, "Embeddable Common Lisp." [Online]. Available: http://ecls.sourceforge.net/

[11] J. Bielman and L. Oliveira, *CFFI User Manual*, 2005-2009. [Online]. Available: http://common-lisp.net/project/cffi/manual/index.html

[12] *CUDA Programming Guide 2.3*, NVIDIA Corporation, 2009. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf

[13] J. Bruno and R. Sethi, "Code generation for a one-register machine," *Journal of the ACM (JACM)*, vol. 23, no. 3, p. 510, 1976.

[14] C. Kessler, "Scheduling expression DAGs for minimal register need," *Computer Languages*, vol. 24, no. 1, pp. 33–53, 1998.

[15] *Maxima 5.20.1 Manual*, 2009. [Online]. Available: http://maxima.sourceforge.net/docs/manual/en/maxima.html