hiCUDA: A High-level Directive-based Language for GPU Programming

TDT24 Summary – Jørgen Nystad

Motivation

- Eliminate or simplify mechanical steps in the process of converting single-threaded programs to CUDA programs
- Introduce OpenMP-like methods for parallelizing programs (compiler directives)
- Reduce development time and errors by providing high level commands

Usual CUDA development process

- Identify and create kernels
- Determine partitioning scheme
- Setup data communication (host to device, device to host)
- Repeat for each kernel until happy (or out of time):
  - Optimize memory usage in kernels (use shared memory, coalescing, etc.)
  - Balance single-thread performance and level of parallelism

Issues with the traditional approach

- Involve significant code changes
- Tedious and error-prone
- Repeated tests to determine best combination of optimizations
- Increased development time
- Makes programs non-intuitive (hard to imagine result as a whole)
- Management and optimization of data in GPU involve "heavy manipulation of array indices"

The Cure

- High-level directives that perform the steps required in the conversion process
- Involves a source to source-compiler for translating hiCUDA programs to CUDA programs

The hiCUDA directives

- #pragma hicuda *some directive with params*
- Possible control directives:
  - kernel
  - loop_partition
  - singular
  - barrier
- Possible data directives:
  - global
  - constant
  - shared
  - shape (not covered)

The *kernel* directive

- kernel – defines a code region to be executed on GPU
  - kernel *kernel_name thread_block_clause thread_clause* [nowait]
    - tblock*(dim-sz {, dim-sz}*)* – specifies block grid
    - thread*(dim-sz {, dim-sz}*)* – specifies thread grid
    - nowait – host does not wait for kernel to complete
  - kernel_end
- Converted to a CUDA kernel…
- Local variables inside the kernel scope are automatically allocated in registers
- Other variables uses data directives

The *loop_partition* directive

- Declared before a for-loop
- Defines how the for-loop is partitioned
  - loop_partition [over_tblock [BLOCK|CYCLIC]] [over_thread]
    - over_tblock – defines distribution across thread blocks
    - over_thread – defines thread distribution within thread blocks
- Allows nesting, nested operators control consecutive dimensions
- Dimensions of grid and blocks must be pre-defined
- hiCUDA compiler generates guard code for imperfect partitions
- See fig. 6 in article for a visual example

The *singular* and *barrier* directives

- Let the enclosed code run on one thread per thread block only
  - singular – opens singular context
  - singular_end
- Synchronize threads
  - barrier – translates directly to a __syncthreads();

The Data Directives

- Two notations for variables
  - *variable*
    - *var-sym* {[*start-idx : end-idx*]}*
    - Variable symbol with associated index range (if array, can be multi-dimensional)
  - *var-sym*
    - Variable symbol only

The *global* directive

- Used outside of kernel
- Three forms:
  - global alloc $variable_a$ [{copyin [$variable_b$]} | clear]
    - Host to device
    - Allocate $variable_a$ and copy contents from $variable_b$ if copyin is present, or initialize to zero if clear is present
  - global copyout $variable_a$ [to $variable_b$]
    - Device to host
    - Copy contents from $variable_a$ to $variable_b$ (or $variable_a$ if not present)
  - global free *var-sym*
    - Free allocated variable
- Never exposes global memory variables to programmer
- See fig. 7 in article for example (without kernels…)

The *constant* directive

- Two forms:
  - constant copyin *variable*
    - Copy contents of *variable* to constant memory
  - constant remove *var-sym*
    - Undeclare constant memory variable *var-sym*

The *shared* directive

- Three forms:
  - shared alloc *$variable_a$* [copyin[(nobndcheck)] [*$variable_b$*]]
    - Global mem to shared mem
    - Allocate *$variable_a$* and copy contents from *$variable_b$* if copyin is present. If not, uninitialized kernel local shared memory is created.
  - shared copyout[(nobndcheck)] *$variable_a$* [to *$variable_b$*]
    - Shared mem to global mem
    - Copy contents from *$variable_a$* to *$variable_b$* (or *$variable_a$* if not present). Must not be performed on a kernel local shared memory variable
  - shared remove *var-sym*
    - Free allocated variable
- Compiler handles merging of consecutive data requests

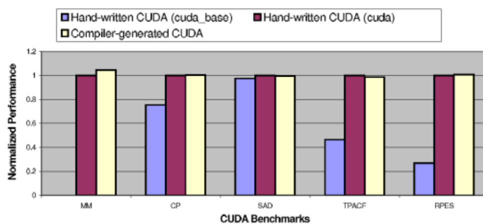
Experimental Results

- Authors built a prototype hiCUDA compiler around Open64 (v4.1)
- Compares execution of bench-mark programs on a GeForce 8800GT
  - Matrix multiplication – own sequential to hiCUDA vs. NVIDIA CUDA SDK version
  - Three benchmarks from Parboil suite, which provides *base*, *cuda_base* and *cuda* versions, which represent sequential, basic CUDA and optimized CUDA, respectively

> *For each benchmark, we start from a sequential version and insert hiCUDA directives to achieve the transformations that result in the corresponding CUDA version.*

- hiCUDA is applied to *base* versions to achieve transformations done in the *cuda* version (!)

Performance



A Note

> *The reader should note that, although we are able to achieve all CUDA optimizations in hiCUDA for these benchmarks, we are sometimes required to make structural changes to the sequential code before applying hiCUDA directives.*
>
> *Such changes include array privatization, loop interchange and unrolling, and the use of fast math libraries. We believe that these standard transformations could be easily automated and incorporated into our language in the future.*

Real World Case

- Medical Research group at University of Toronto
- Monte-Carlo simulation of multi-media tissue
- Manual CUDA version written in 3 months achieved 27x speedup on a 8800GTX vs single-threaded on Intel Xeon (3.0 GHz)
- hiCUDA version of the same code with the same optimizations written in 4 weeks achieving the same speedup
- Impressed?

Conclusion

- Authors concludes that high performance can be achieved using hiCUDA compared to manually optimized versions
- The real world case confirms that development time can be significantly reduced
- Authors admit that significant loop alterations may be necessary before using hiCUDA and would like to automate this in the future
- Would also like to simplify the current directives or make compiler insert them automatically, ultimately resulting in automatic parallelization for GPUs...