PTX Backend for Psuedo Language with PARALLELFOR Construct

David Fiala

CSC 766 – Final Project

December 2010

Contents

Project Description 2
Pseudo Language Syntax and Modifications 3
Variable Declaration 3
PARFOR Construct and REDUCE Operators 4
Additional Mathematical Functions 4
IO Placement Restrictions (READ/WRITE) 4
PTX Driver 5
Overview 5
IO Communication Buffers 6
Kernel Execution 6
Providing Multiple Streams for Concurrent Global Memory Access 6
Thread Count 7
PTX Compiler 8
Overview 8
PTX Target Version & Compute Capability 8
Variable Storage 8
Register Allocation 8
PARFOR Construct 9
Variable Privatization 9
Parallel Reduction 9
Open Issues 11
PARFOR Construct within Loops 11
Parallel Reduction on Floats (Workaround Used) 11
Results 12
Closing Remarks 16
For Fun: Definite Integral Approximations using PARFOR 17

Project Description

This project built on the original pseudo Language specification from class homework 2, but produces raw PTX assembly instructions that can be run on NVIDIA GPUs. In addition to providing all of the functionality of the homework 2 language, my compiler provides a PARALLELFOR construct that exploits the high degree of parallelism available on today's GPUs. In an effort to make the PARALLELFOR construct more functionally complete, a REDUCE operator has been implemented that provides parallel reduction capabilities on the PARFOR construct while taking advantage of the PTX instruction set to provide reductions in an efficient manner. Finally, a specialized PTX "driver" application was developed for the purpose of loading generated PTX code on the GPU, launching, timing, and providing input/output functions to the PTX code.

Pseudo Language Syntax and Modifications

Several additions were made to the original language syntax. Additions and modifications are shown in bold:

Variable Declaration	**DECLARE** ***VariableName ArrayName[IntegerAsSize] …*** **ENDDECLARE**
Assignment	Variable := Expression
	Variable[Expression] := Expression
Expression	Variable
	Variable[Expression]
	IntegerNumber
	RealNumber
	Expression ArithmeticOp Expression
	MathFunct(Expression)
MathFunct	**SQRT SIN COS RCP**
LogicOp	AND OR
ComparisonOp	= <> > < >= <=
Comparison	Expression ComparisonOp Expression
	Comparison LogicOp Comparison
	NOT Comparison
Block	BEGIN ... END
Test	IF Comparison THEN ... ELSE ... ENDIF
	IF Comparison THEN ... ENDIF
Loop	WHILE Comparison DO ... ENDWHILE
	REPEAT ... UNTIL Comparison ENDREPEAT
	FOR Variable := Expression TO Expression DO ... ENDFOR
	PARFOR Variable := Expression TO Expression **[PRIVATE (Variable [*More Variables...*])]** **[REDUCE *ADD/MIN/MAX* ReductionVariable]** **DO … ENDPARFOR**
Input	READ(Variable)
Output	WRITE(**Expression**)

Variable Declaration

To avoid a multi-pass compilation process, all variables are required to be declared at the start of the program. Unlike the original java-based pseudo language which dynamically added memory as needed, scalars and arrays are allocated a fixed amount of memory in the GPU device's shared memory. The DECLARE section is required to be the first section of the pseudo language source code and must precede all other instructions.

PARFOR Construct and REDUCE Operators

The PARFOR construct provides a very similar syntax as compared to the original FOR construct, except for the addition of a PRIVATE clause and a REDUCE clause which was heavily influenced by OpenMP's reduction operator for parallel for loops. The PRIVATE clause takes a space-separated list of scalar variable identifiers inside a pair of parenthesis. Each variable listed will be assumed to be private to each thread/iteration of the PARFOR construct. It should be noted that the PARFOR index variable is automatically assumed to be a private variable in addition to any other explicitly stated privates.

The REDUCE clause expects an operator ADD, MIN, or MAX which will be applied to the specified scalar variable. At the completion of the PARFOR block the result of the reduction operation across all iterations will be stored in the specified scalar. The ADD operator can be thought of as a summation operator while the MIN and MAX operators return their respective minimum and maximum value encountered throughout all iterations of a PARFOR.

Additional Mathematical Functions

Providing more advanced mathematical functions inside the language provides a more appealing platform for mathematical- or scientific-based applications. Therefore functions for calculating the square root, sine, cosine, and reciprocals of numbers were implemented as functions within the new language specification.

IO Placement Restrictions (READ/WRITE)

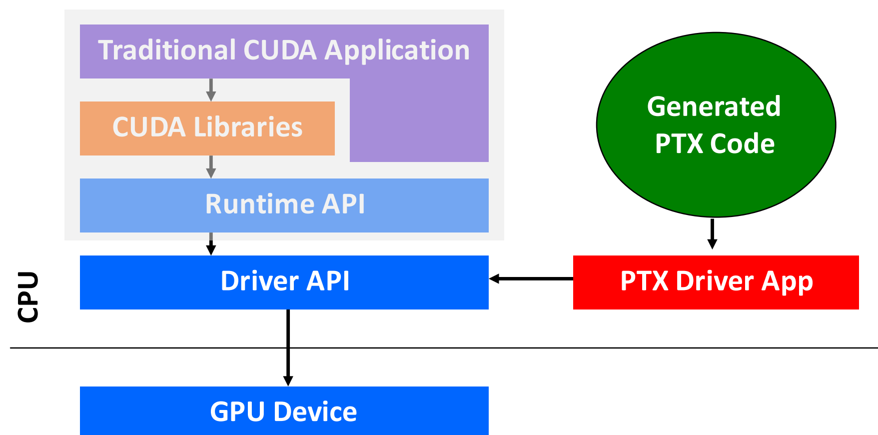
The purpose of the READ and WRITE functions is to read and print numbers to and from STDIN/STDOUT. Since the PTX code is run on a GPU which has no direct access to the user's terminal, a customized solution was created to allow IO communication between the host and the GPU. Due to the unique constraints imposed by IO communication on the GPU, it is not possible to provide READ/WRITE functionality concurrently among multiple running threads such as in a PARFOR block. READ/WRITE statements are permitted anywhere outside of a PARFOR block.

PTX Driver

Overview

Traditionally, code meant to be run on an NVIDIA GPU is designed and written in C language which then gets compiled by a special NVIDIA compiler that translates the C code in to a GPU kernel binary code which is embedded in to the compiled executable. Inside the C code, a call to a specially decorated function will transparently run on the GPU itself. Therefore, after compilation a user need only run their x86/x86_64 executable, and the portion destined to run on the GPU is automatically loaded and run without any user intervention. The entire process from design-time to runtime is moderately trivial for the developer and entirely transparent to the user (Provided they have a compatible GPU installed!).

In this project however, the output of the compilation process is raw PTX code and not an executable binary for the host system. A special host driver application is required in order to run the PTX code on the GPU. In order to communicate with the GPU, one of NVIDIA's CUDA APIs will be needed. There are two major APIs for CUDA programming: The low-level API called the CUDA driver API and the higher-level API called the CUDA runtime API which is implemented on top of the driver API. Typically developers use the runtime API as it eases device management, and provides implicit initialization, context management, and module management. However, the driver API requires more code, is harder to program and debug, but it offers a much higher degree of control over the GPU device. Namely, the driver API can load raw PTX code (or the binary form called CUBIN) directly, while the runtime API has absolutely no way to execute custom PTX code directly.



For this project the PTX Driver application was implemented using the NVIDIA driver API which allows direct loading of generated PTX code as well as very fine control over GPU kernel execution parameters. Additionally, the driver API allows read/write access to the GPU's global memory which will provide a means for IO with a running kernel as described in the next section.

IO Communication Buffers

The READ and WRITE functions require the ability to both read and write from the user's terminal in order to function properly. Since the PTX code is running on an entirely separate processor, there is no direct way to allow communication with the user's terminal. However, the driver application may access the GPU's global memory space. By allocating a portion of global memory as an intermediate buffer for communication, it is possible to allow the PTX program to write data to a buffer which the host driver will be able to read and display to the user. Likewise, the driver can listen for keyboard input and then write it to another separate global memory buffer which the PTX code may access.

In the simplest sense, each buffer could contain a set-aside amount of buffer space as well as a pointer to the most recent data-item. While this technique may look promising initially, unfortunately the available buffer space will get exhausted as soon the pointer reaches the end of the buffer space. By using two pointers instead of one we can create a circular-array that allows for unlimited IO between the CPU and GPU.

In the driver implementation of this project, I maintain two buffers to provide READ and WRITE communication. Each buffer has a static size of 100 elements, but being circular in nature it can handle as much data as is needed over time. The host driver watches the keyboard for STDIN input and then immediately stores the data on to the READ buffer. Additionally, the driver monitors the GPU's WRITE buffer for any new data to write to STDOUT for the user to see. In the event that either circular buffer becomes full, one processor waits for the other one to remove an element from the full buffer and then proceeds. Due to order in which writes are ordered, there is no race condition updating the pointers or data in either buffer.

Kernel Execution

In order to launch the PTX kernel, the driver must prepare the GPU and environment. Upon startup, the driver finds a compatible GPU device, creates an execution context and proceeds to load the PTX code in to GPU memory. The driver then allocates memory for the IO buffers and provides pointers to the buffers as kernel parameters. Finally, the driver marks the time before launching the kernel using precise CUDA driver timing functions. Once the environment setup is complete, the kernel is launched via the driver and upon completion another timestamp is taken to compute the total time spent running on the GPU.

Providing Multiple Streams for Concurrent Global Memory Access

Whenever the driver makes any CUDA API calls, they may block until the operation completes. Namely, when a kernel is executed and then a read access to global memory is posted by the CPU, the actual read request is blocked until the kernel execution is completed. Under these restrictions it is impossible to provide concurrent READ/WRITE IO to a running GPU program—The CPU driver application is immediately blocked whenever it tries to read memory on the GPU device! In order to overcome this problem, during initialization the driver makes two separate "streams" to the

GPU. One stream is used exclusively for timing and launching the GPU kernel, while the other stream is used exclusively for IO support. While the kernel stream is running, the IO stream does not block and is allowed timely access to the GPU's global memory. The IO stream constantly scans global memory for changes to the IO buffers and processes new data elements by printing them to screen.

Along the same lines, an additional CPU host thread is utilized to process keyboard STDIN input. When new data is read from the keyboard it is then passed over the IO stream for read access on the GPU.

Thread Count

The driver takes as a command line argument the number of threads to execute. While preparing to the execute the kernel, the driver API sets the "shape" of the thread block to launch and uses the thread count to specify the dimension of the thread block. Since the PTX code uses shared memory for variable storage, you can only allocate as many threads as fit in a single thread block since shared memory is only shared between a single thread block. For this project's target NVIDIA compute capability, a maximum of 512 threads is allowed.

PTX Compiler

Overview

The PTX backend for this project is written using flex and bison and produces raw PTX code. Since there is some common functionality and boiler-plate initialization code necessary for every PTX program, a small template is written out before the generated PTX code for every file. The template includes initializations for setting up the IO buffers, calculating the thread index, and allocating register space for later use.

The compiler was designed with a few goals in mind:

- Minimal register allocation (to reduce pressure with limited register space available on a thread block)
- Evaluate expressions and computations entirely with minimal off-chip bandwidth (register reuse when possible)
- During parallel sections, provide opportunities for memory access coalescence (non-divergent code)
- Exploit PTX instruction set to provide simple, elegant code
- Use shared memory for variable and array storage to lower access latencies

PTX Target Version & Compute Capability

In order to support the most recent PTX instruction set architecture, code generated by this compiler requires PTX target version 1.4. Although PTX ISA 1.4 supports a wide variety of functionality such as parallel reduction multiple data-types, the compute capability of the NVIDIA GPU you run on determines what functionality can actually be *used.* To support some of the advanced functionality such as parallel reductions on signed integers, this compiler produces PTX code that requires a compute capability of at least 1.2.

Variable Storage

Variables declared at the start of a program are entered in to a symbol table during parsing. When PTX code is generated, the symbol table is traversed to create PTX directives that allocated shared memory for all of the necessary variables. To provide a unified data-type for all operations inside the PTX program, all variables are considered to be a 32bit float in the PTX code. The decision for a 32bit float relies on the fact that a scientific application will likely require floating point numbers that a signed integer could not provide. Furthermore, given the limited GPU shared memory space available, a double precision float would likely be detrimental to the maximum size of arrays.

Register Allocation

While generating code, registers are treated as a stack in many cases. Whenever a register is temporarily needed, one is allocated and the stack’s count is increased. Since the control flow of an application is also treated as a stack, this makes a very elegant solution to register allocation. Additionally, registers are accessed relative to the top of the stack.

When the control flow enters a new block and additional registers are used and then eventually released, the same relative offsets still apply to code after exiting a block.

When dealing with expression evaluation, the stack approach also produces good results and minimizes register pressure. As an expression is evaluated, the parser provides the correct order of operations, and in many cases only two registers are needed to evaluate even complex numeric expressions.

PARFOR Construct

When a PARFOR construct is encountered, all of the running threads converge at the start of the block and allocate several registers to store their individual states.

Each thread needs to know:

- The index value of the last iteration (the TO value)
- How many times to loop (loops remaining)
- The thread's initial index value

The total number of times to loop is calculated by dividing the number of iterations by the number of threads:

```
loops = (end value – start value) / num threads
```

Initially, the thread's index value will be equal to the thread's ID plus the start value. After executing an iteration, the thread will decrement the loops remaining value, increment its index by the number of threads available, and check if it should continue on to another iteration or be finished.

The entire PARFOR section has all threads executed converged and allows for as many iterations as specified even if the number of iterations is several times that the number of available of threads. No matter if the user selects 1, 200, or 512 threads total, the result will still be serializable.

Variable Privatization

During a parallel for, private variables are likely to be used frequently which makes them a good candidate to be stored as registers. Additionally, storing private variables as shared memory would require a large chunk of dedicated space which is generally available. For this reason, private variables are allocated as registers during a PARFOR block, and a separate symbol table entry is used to keep track of which register a private variable resides in. Fortunately, by modularizing the compiler's internal accesses to variables, it is very easy for the code generator to switch between accessing variables as shared memory to registers.

Parallel Reduction

To perform efficient parallel reduction, all threads simultaneously issue atomic adds, mins, or maxs at the end of each loop iteration. The PTX ISA efficiently makes use of a simultaneous atomic reduces to compute and store the result of the reduction in to shared memory. The next time reduction is performed, the previous value in the shared memory

result is looked up and is part of the reduction. In that manner, an existing value is added in to an ADD and a previous MIN/MAX is also taken in to consideration.

Prior to the first parallel reduction in a PARFOR block, it is also necessary to initialize the shared memory result to get proper results. When an ADD reduction is performed, the initial value should be set to 0. For a MIN, the maximum value for an integer* is used. Finally, for a MAX, the minimum value for an integer* is used.

**See open issues for the reason an integer max/min is used.*

Open Issues

PARFOR Construct within Loops

In designing the way threads diverge and converge during sequential regions, I took an approach where only thread zero runs during sequential regions and all of the other threads branch ahead to the next PARFOR region and wait. Unfortunately this turned out to be a naïve approach when you consider what happens with a PARFOR inside of a loop. During the first loop outer loop iteration, all threads will eventually converge at the PARFOR block and run until the completion of the PARFOR. The extra threads are then instructed to jump ahead to the next PARFOR block within the code (if any – otherwise jump to exit). Meanwhile, thread zero gets to the end of the first iteration of the outer loop and then jumps to the start for another iteration. Once thread zero reaches the PARFOR block again it is unable to reconverge with the other threads as they are at another point in the code waiting for thread zero.

It should be noted that this in no way affects loops inside of a PARFOR. Any combination of instructions remains valid and functional inside of a PARFOR except for embedding another PARFOR inside of itself.

As a possible solution, instead of having the parallel threads jump to the next PARFOR block immediately, they should jump to points where thread zero does a comparison before deciding to repeat a loop or not. Thread zero would need to broadcast its next control-flow decision to the other threads and allow them all to jump to the next destination together instead of getting left behind.

Parallel Reduction on Floats (Workaround Used)

During a parallel reduction, an atomic reduce is used to compute the reduction results. Unfortunately NVIDIA compute capabilities prior to version 2.0 do *not* support atomic reductions on 32bit floats. Although PTX target 1.4 syntactically supports them, it is up to the compute capability to determine what can actually run on the GPU.

As a workaround, during any parallel reduction in this code, 32bit floats are temporarily converted in to 32bit signed integers during the reduction and then returned to 32bit floats. While this workaround is successful, it does result in a loss of decimal precision. If a certain level of decimal precision is absolutely required, you could multiply your reduction variable by a power of 10 before reduction, and then divide the final result by the same power. This solution also carries its own overflow risks and should be used with caution.

Results

To demonstrate the speedup provided by a PARFOR, I will use a trivial micro-benchmark that calculates the squares of [0..n] and stores the results in an array. Result printing has been removed to reduce space.

Sequential Version

```
DECLARE
i arr[1024] j
ENDDECLARE

FOR i := 0 TO 1024 DO
        FOR j := 0 TO 4000 DO
                arr[i] := i * i;
                arr[i] := SQRT(arr[i]);
                arr[i] := i * i;
                arr[i] := SQRT(arr[i]);
                arr[i] := i * i;
                arr[i] := SQRT(arr[i]);
                arr[i] := i * i;
                arr[i] := SQRT(arr[i]);
                arr[i] := i * i;
                arr[i] := SQRT(arr[i]);
                arr[i] := i * i;
                arr[i] := SQRT(arr[i]);
                arr[i] := i * i;
        ENDFOR;
ENDFOR;
```

Parallel Version

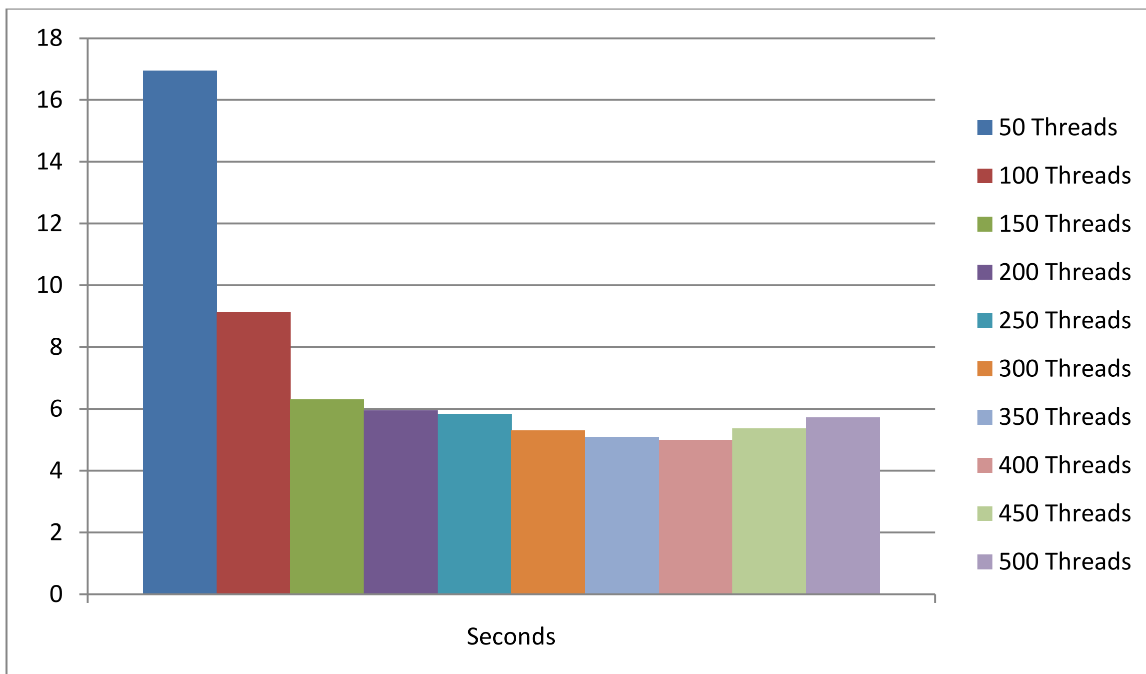
```
DECLARE
i arr[1024] j
ENDDECLARE

PARFOR i := 0 TO 1024 PRIVATE(j) DO
        FOR j := 0 TO 1000000 DO
                arr[i] := i * i;
                arr[i] := SQRT(arr[i]);
                arr[i] := i * i;
                arr[i] := SQRT(arr[i]);
                arr[i] := i * i;
                arr[i] := SQRT(arr[i]);
                arr[i] := i * i;
                arr[i] := SQRT(arr[i]);
                arr[i] := i * i;
                arr[i] := SQRT(arr[i]);
                arr[i] := i * i;
                arr[i] := SQRT(arr[i]);
                arr[i] := i * i;
        ENDFOR;
ENDPARFOR;
```

Directly comparing the performance difference between the sequential version and parallel version is made difficult because the parallel version runs magnitudes faster. Obtaining meaningful results for the parallel version would require so many loop iterations that the sequential version would timeout before finishing.

Nonetheless, the results of a direct comparison between the sequential version (4000 loop iterations) and the parallel version with 1 thread (modified for 4000 loop iterations) produces a very interesting result: The sequential version runs in 8.39 seconds, whereas the parallel version with 1 thread took only 3.01 seconds. The reason behind this huge difference arises from the use of register allocation of privatized variables. Specifically, in the parallel version “i" and “j” both are registers, whereas in the sequential version each access requires waiting for shared memory. This interesting result not only shows a huge performance gain in register use, but also provides a good reason to not perform direct comparison between sequential and non-sequential codes when variable allocation schemes are different.

Next, a comparison of the parallel code version shows an intriguing speedup as the thread count is increased with 1,000,000 iterations:



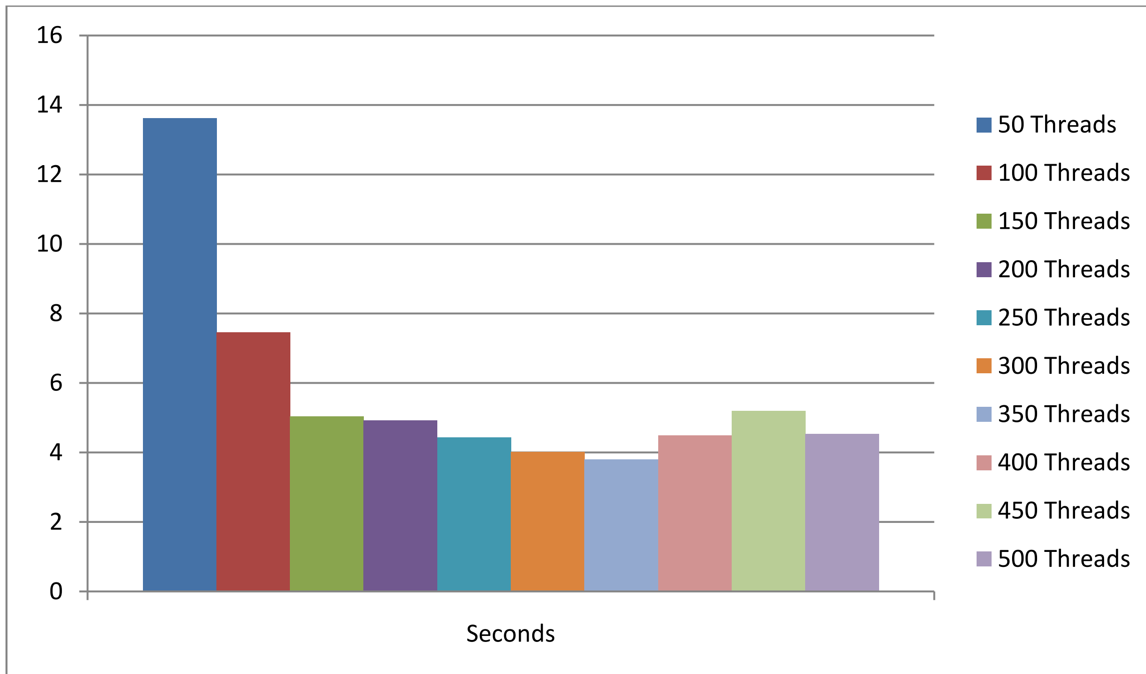
We can see that the peak speed is achieved somewhere around 400 threads, but I think that the answers to these results are a little deeper than they first look. Since the parallel version benchmarked was accessing shared memory for the arr[] array, I decided to try another version that removed all accesses to shared memory, but increased register pressure by allocating another private variable. Since the previous version should have allowed coalesced memory accesses, I doubted that shared memory was the problem, but another test is needed to rule it out:

Parallel Version #2 (No shared memory accesses in PARFOR)

```
DECLARE
i j x arr[2000]
ENDDECLARE

PARFOR i := 0 TO 1024 PRIVATE(j x) DO
        FOR j := 0 TO 10000000 DO
                x := i*i;
                x:= SQRT(x);
                x := i*i;
                x:= SQRT(x);
                x := i*i;
                x:= SQRT(x);
                x := i*i;
                x:= SQRT(x);
                x := i*i;
                x:= SQRT(x);
                x := i*i;
                x:= SQRT(x);
                x := i*i;
                x:= SQRT(x);
                x := i*i;
                x:= SQRT(x);
                x := i*i;
                x:= SQRT(x);
                x := i*i;
                x:= SQRT(x);
                x := i*i;
                x:= SQRT(x);
                x := i*i;
                x:= SQRT(x);
                x := i*i;
                x:= SQRT(x);
                x := i*i;
                x:= SQRT(x);
                x := i*i;
                x:= SQRT(x);
                x := i*i;
                x:= SQRT(x);
                x := i*i;
                x:= SQRT(x);
                x := i*i;
                x:= SQRT(x);
                x := i*i;
                x:= SQRT(x);
                x := i*i;
                x:= SQRT(x);
                x := i*i;
                x:= SQRT(x);
                x := i*i;
        ENDFOR;
ENDPARFOR;
```

The results for this new version:



Quite interestingly, the results with shared memory accesses provide a very similar graph, even though the number of iterations was changed to 10,000,000 to provide relative results. During the execution of the second test, the only resource in high contention was register use. Up to 7 temporarily allocated float registers were in use per thread, in addition to a predicate register (logical expression evaluation). Finally, I manually removed many threads barriers and memory synchronization instructions from the generated PTX code to also rule out their overheads, but the results were nearly identical for all conditions. At this point my conclusion is that the performance gains diminish due to register pressure. When the register pressure becomes too high, registers are swapped out to local memory which has a high access time latency.

Closing Remarks

This project has been a fun divergence into the world of GPUs and assembly code generation. While I did not have enough time to implement even a fraction of the numerous optimizations techniques that were presented in class, I believe that this project helped me understand the core concepts stressed while allowing me to gauge the difficultly and importance of the more advanced compiler optimizations. Even on a small scale such as this project, the results clearly show that the need for intelligent compilers is absolutely critical to tune code if you expect to get anywhere near the theoretical maximum performance that our modern CPUs and GPUs provide.

For Fun: Definite Integral Approximations using PARFOR

The following code shows the appropriate usage of a PARFOR block performing an integral approximation from a technique I learned way too long ago…

It works great for relatively small integrals, but the 32bit floats prevent it from being effective for anything function with a degree or very small numbers such as tiny strips.

```
//Calculates the closed integral of a function using
//the Riemann Sum method using the middle sum
//technique for approximation.
//
//Demonstrates the use of a PARFOR block to perform
//each strip's partial sum in parallel.
//
//Sample input to calculate the closed integral:
// f(y) = 12x^4 + 5x^3 + 2x^2 + 20x + 15
// over x=10 to x=200 with 20000 strips:
//     10 200 4 20000 12 5 2 20 15
//
//Warning: Can't be validated on large integrals
//due to the poor precision of 32bit floats. :(

DECLARE
start      //where to begin the definite integral
end        //where to stop the definite integral
strips     //how many strips to take
degree     //the degree of the function we are integrating
coef[32]   //the coefficients of each degree
stripsize  //how wide each strip will be
i j        //multiuse temporary variables
x y sum
ENDDECLARE
READ(start);
READ(end);
READ(degree);
READ(strips);
//Read in each coefficient and a constant
i := degree;
WHILE i >= 0 DO
       READ(j);
       coef[i] := j;
       i := i - 1;
ENDWHILE;
stripsize := end - start;
stripsize := stripsize / strips;
PARFOR i := 0 TO strips PRIVATE(i j x y sum) REDUCE ADD sum DO
  //calculate the x offset for the middle of this strips
  x := stripsize / 2 + stripsize * i;
  //evaluate f(x) at this position
  y := coef[0];

  sum := 0; //sum will temporarily hold x, then x^2, then x^3, ...
  FOR j:= 1 TO degree + 1 DO
    sum := sum + x;
    y := y + sum * coef[j];
  ENDFOR;
  sum := y * stripsize;
ENDPARFOR;
WRITE(sum); //Output final integral approximation after parallel reduction
```