# PTX Backend Extension to Experimental Programming Language

Jeff Miles (jsmiles), https://sites.google.com/a/ncsu.edu/jsmiles_csc766_pr1/

## Final Report

In an exercise to demonstrate and explore the possibilities of integrating GPU technologies into existing compiler frameworks, a simple pseudo language compiler framework has been developed and enhanced to include PTX code generation for simple parallel for loop constructs. As described below this effort demonstrates both the feasibility and profitability of adding data parallelism via GPU support to even to a simple and limited language. The intent of such an exercise is to explore the issues and benefits of integrating such code generation into an existing compiler framework. This report contains a final description of the compiler framework, a basic analysis of some simple performance tests and some concluding remarks concerning the state of the industry.

## Completed Framework

While it may be desirable to utilize several open source compiler tools to manage PTX module generation, the current state of these tools and LLVM add-ons are not such that they can be integrated in a project with a short development cycle. Instead the tools included with the CUDA SDK have been utilized to generate the PTX modules via output from the compiler front end. The compiler front end has been separated into two parts: the first pass generates the host code, and the second generates a .cu module. This .cu source is then used to generate a .ptx module. Figure 1 demonstrates the process of code generation and shows which tools are referenced.

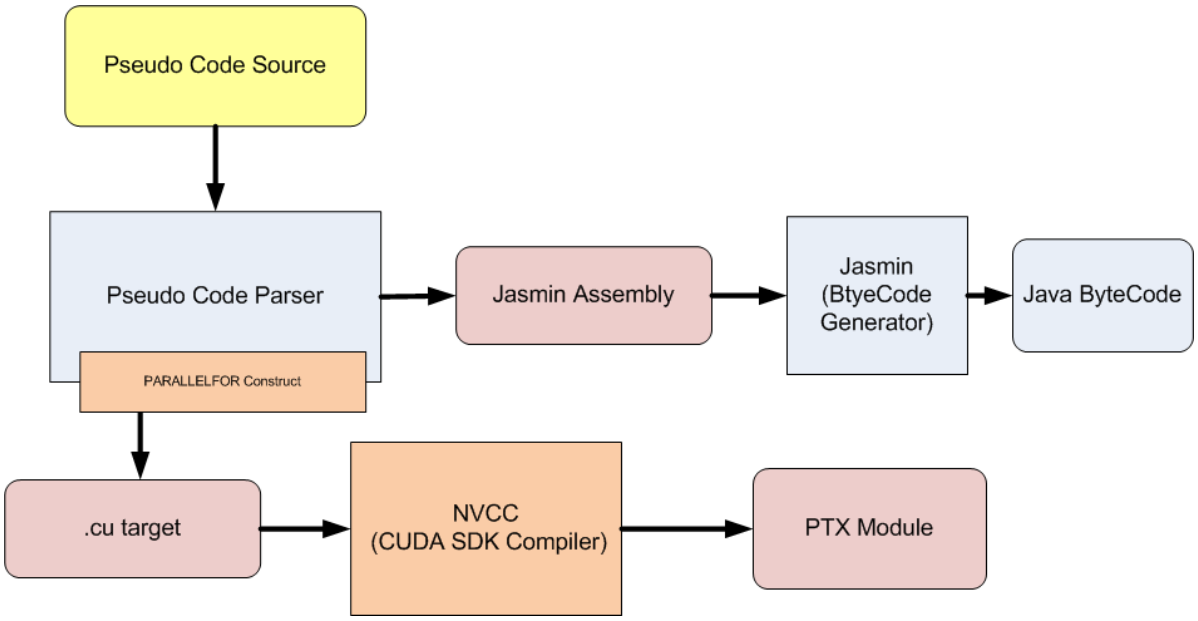


**Figure 1 Pseudo language compilation framework**

This design minimizes the number modules generated for each pseudo language program and also limits the tool requirements to a manageable set. One limitation which should be noted in the current version is that the number of PTX modules (and therefore the number of parallel loops) for each program is limited to one. This limitation is easily resolved by numbering the parallel loops and then generating the corresponding PTX modules; however, not having this functionality does not prevent the validation of these concepts.

At run time, a pre-compiled native library is accessed by the host byte code whenever a parallel for loop is encountered. This native library then loads the ptx module dynamically and invokes the device function. A key feature of this design is that the native method can be implemented with a completely generic interface and thus not recompiled for each pseudo language program. The parameters for this interface include the data, the iteration boundaries and the PTX module name. Figure 1 below shows the run time framework used by the pseudo language programs.

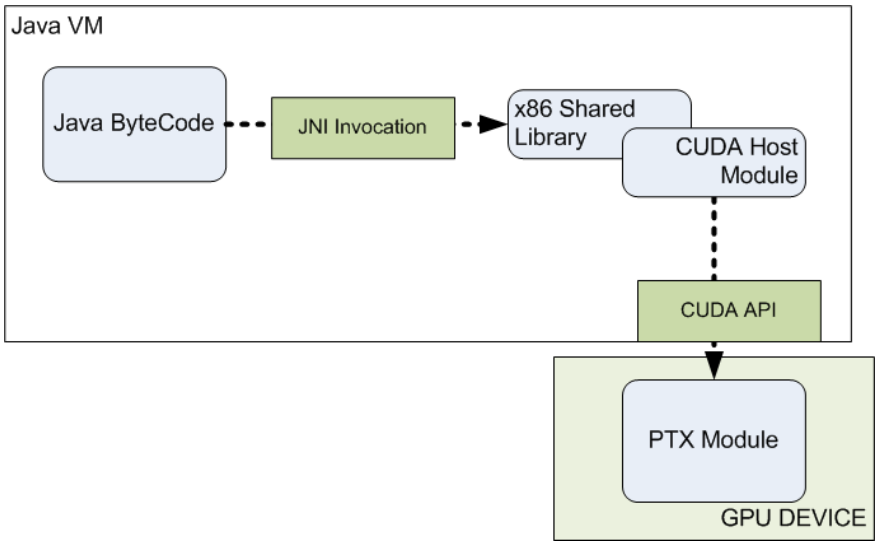


**Figure 2 Runtime framework for pseudo language programs**

In this design, the native library (indicated by "x86 Shared Library") and the CUDA Host module are both pre-compiled and not modified with each pseudo language programs. The only property contained within these modules is the thread block size. The number of blocks is then a function of the thread block size and the number of iterations in the loop (Max/Thread Blocks). Prior to module invocation, the data for each pseudo language program is copied to a 2D Java array which is then copied again to a linear C array. This C array contains the host memory which is then copied to the GPU device prior to launching the device function. The PTX module references these data values using offsets based on the specified size of each array within the pseudo language program. The frontend compiler has been modified to take the maximum array size as a parameter which is used for both copying data as well as specifying the data offsets in the PTX module. After the device function is complete, the data is then copied back in reverse order to the original pseudo language variables.

## Results and Discussion

In order to test and compare the performance of programs compiled using the enhanced pseudo language framework a simple program with a number of computations contained a test for loop is implemented using both the traditional FOR loop and also the PARALLELFOR loop. These programs were then organized and compiled with an increasing number of loop iterations (and data elements) to compare/analyze the execution time as a function of elements/(loop size). The CUDA access library mentioned above was compiled with a hard-coded thread block size of 100 specified in a linear block definition so the loops smaller than 100 load a number of threads greatly exceeding those needed. These tests have been compiled and executed using test sizes ranging from 10 to 100000 and the results are illustrated in the chart below.

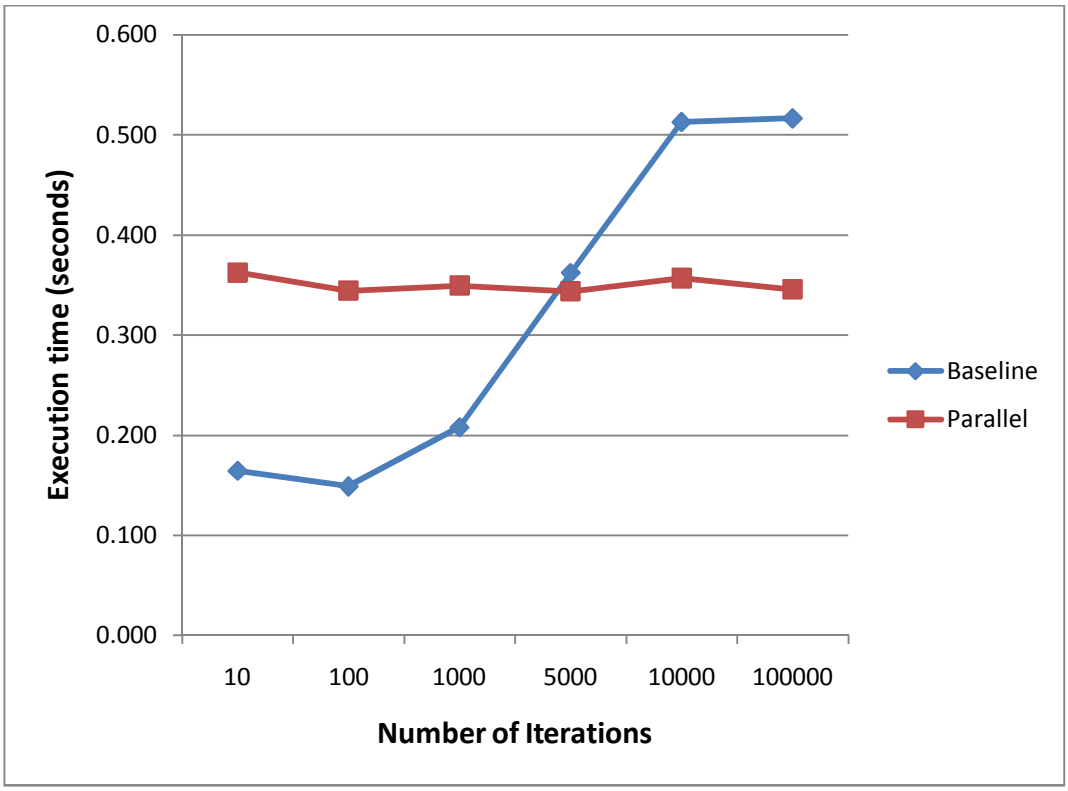


**Figure 1 Comparison of execution times for serial and parallel programs**

The figure above graphically shows the execution time for both the serial (baseline) and parallel execution modes. From the chart, one can easily see that the serial execution time is linearly proportional to the number of data elements/iterations, where as the parallel execution are essentially constant. It is important to note here that the pseudo language has no support for outputting the system time so the time markers used are outside the execution of the java VM within the test script itself. Because of this, the java loading overhead is included in the execution times for both the serial and parallel execution results. Also, the parallel versions incur a one-time overhead for loading the shared libraries. This overhead should not be too significant, but it should be noted in the results. In any event, the parallel execution mode demonstrates good performance compared to t he serial

versions. Considering the lack of optimizations and a somewhat inefficient data movement design this is quite remarkable. Therefore, had the purpose of this effort been to optimize the performance results, these figures could have been drastically improved.

## Conclusion

Once again, the feasibility and profitability of implementing a PTX backend for even the simplest of programming languages has been demonstrated and validated with the results from this development effort. While not all programs require large loops with tightly coupled data parallelism, the focus of this study are the language compilers which are used to generate a variety of programs structures. Likewise, the ever increasing popularity of hybrid programming environments requiring the ability to take advantage of data parallelism shows that this type of effort can be valuable even in commercial environments. Such additions as those described in this report can greatly expand access to many fine grained parallelism constructs without requiring drastic development efforts or changes to existing programs. Further enhancements to the framework developed herein should include improvements in the data movement process, PTX optimizations, and support for multiple parallel loops. If the author's expectations hold true, these and other additions, could only show further speed ups in the performance comparisons, which are only valuable for those readers that are still skeptical.