A Map Reduce Framework for Programming Graphics Processors

Bryan Catanzaro, Narayanan Sundaram and Kurt Keutzer
University of California, Berkeley
545Q Cory Hall
Berkeley, California 94720
{catanzar, narayans, keutzer}@eecs.berkeley.edu

ABSTRACT

Recent developments in programmable, highly parallel Graphics Processing Units (GPUs) have enabled high performance general purpose computation. We describe a framework designed for high performance GPU programming, built on Nvidia's Compute Unified Device Architecture (CUDA) platform. The framework is built around the Map Reduce abstraction, which allows application developers to focus on their application, while enabling high performance GPU implementation. We show the utility of our framework by implementing Support Vector Machine training as well as classification, achieving speedups of up to 32× and 150× respectively over commonly used SVM software running on a CPU.

1. INTRODUCTION

Driven by the capabilities and limitations of modern semiconductor manufacturing, the computing industry is currently undergoing a massive shift towards parallel computing [1]. This shift brings dramatically enhanced performance to applications which capitalize on parallelism.

However, applications must be reengineered to express parallelism in a way which maps well to the architecture of these new platforms, which is a significant barrier to the future success of parallel computing. In order to map a given application to a parallel computer, the application must be first re-examined in order to discover parallelism. Then, the application must be reimplemented on a parallel platform. Finally, when porting the application to a future, different parallel system, the whole process may need to be restarted, if there is significant mismatch between the scope and types of parallelism supported by the old system and the new. To address these problems, tools and methodologies for programming highly parallel systems must be created which automate the programming process.

Programming GPUs remains a complicated task despite the recent advances made to make GPU programming simpler. We propose to alleviate this complexity by making use of the Map Reduce programming abstraction. The Map Reduce abstraction helps application developers focus on their application, rather than optimizing GPU code.

In this paper, we present preliminary details of a Map Reduce code generation framework for programming GPUs. Our framework generates very efficient code for the GPU, without inserting any abstractions or indirections. It improves programmer productivity by keeping the programmer's attention on the algorithm, rather than the specific details needed to implement the algorithm on the GPU. The focus on efficiency is critical to the scope of applications addressed by the framework, since applications which need to perform many Map Reduce computations can't afford overhead in the framework.

We demonstrate the utility of our approach by implementing two different applications using the framework, namely Support Vector Machine training as well as classification. We detail how our framework yields high productivity and high performance.

The organization of the paper is as follows. Section 2 presents the Map Reduce pattern in detail. Section 3 gives an overview of GPU architecture and the programming environment on which our framework is built. Section 4 explains our implementation of the Map Reduce pattern for the GPU. Section 5 details the applications with which we demonstrate the usefulness of our implementation, namely Support Vector Machine training and classification. We show our results in Section 6 and conclude in Section 7. Appendix A presents a sampling of the syntax used by our framework.

2. MAP REDUCE

Many applications feature large amounts of independent computation, followed by a global summarization of the computation. We call the independent computations a "Map" operation, and the summarization, a "Reduce" operation.

The idea of Map Reduce dates back at least to the Lisp programming language, which allowed programmers to map independent computations onto data sets, using reduce operations to summarize the results [19]. Recently, Google proposed a Map Reduce variant for processing large datasets on compute clusters [7], where programmers specify a map function that operates on a set of (key,value) pairs to produce a set of intermediate (key,value) pairs, as well as a reduce function that merges all the intermediate results with the same key values. The Map Reduce pattern has been shown to be useful for many applications, e.g. those from Machine Learning [5].

The Map Reduce pattern has been successful because it is a natural way to express many computations. However, it is not just syntactic sugar. Expressing computations in terms of maps and reductions preserves parallelism, which enables efficient mapping onto parallel machines.

Parallelism in the Map stage is evident and requires no explanation. However, the Reduce stage by its nature has diminished parallelism, and requires some synchronization. This makes implementation of the Reduce stage complicated and can lead to performance bottlenecks, especially on very parallel architectures which have limited synchro-

nization and communication abilities between threads, such as the GPU.

For each Map Reduce computation in an application, our framework asks the programmer to provide a map function, a set of reduce operators, and a cleanup function which operates on the result of the reduction. The framework then restructures the code to produce one function which implements the map function combined with a local reduction, and another which provides a global reductions, combined with the cleanup computation. Because the programmer doesn't specify *how* the reduction is accomplished, but instead gives only an atomic description of each reduce operator, the framework is free to implement many different styles of reduction, allowing the highest performance reduction technique for a given problem to be used. Additionally, the programmer does not write the complicated and error prone code which deals with synchronization and communication between threads. Furthermore, the programmer does not have to restructure their algorithm around the synchronization abilities of the GPU.

Other groups are also creating Map Reduce frameworks for multiprocessor systems. Ranger presents a runtime system for doing efficient Map Reduce computations on multicore CPUs [16]. This system closely follows Google's Map Reduce framework, even providing fault tolerance. Linderman presents a Map Reduce programming model for heterogeneous systems [13], which includes a runtime for dynamically decomposing and executing tasks tasks on Intel CPUs and GPUs. However, all these runtime systems incur significant overhead in order to provide flexibility and dynamic task partitioning. When Map Reduce computations are composed inside loops or other structures, this overhead can become very substantial. Our code generation framework keeps overhead to a minimum by integrating the map phase and the reduce phase of the computation, which keeps CPU/GPU synchronization to a minimum and avoids the need to stage datain global memory for the reduction. This makes it possible to achieve meaningful performance speedups, even on computations where the map reduce is relatively small and inside an iterative loop.

3. GRAPHICS PROCESSORS

GPUs are currently transitioning from their initial role as specialized accelerators for triangle rasterization to general purpose engines for high throughput floating-point computation. State of the art GPUs provide up to an order of magnitude more peak IEEE single-precision floating-point than their CPU counterparts. Additionally, GPUs have much more aggressive memory subsystems, typically endowed with more than 10× higher memory bandwidth than a CPU.

However, programming the GPU to extract such performance is still difficult, since GPU performance is dependent on large amounts of parallelism. A typical computation running on the GPU must express thousands of threads in order to effectively use the hardware capabilities. Additionally, writing code for the GPU which requires global communication or synchronization is complicated, error prone, and very specific to the exact GPU which is being targeted.

Map Reduce is an ideal abstraction for programming general purpose computations on the graphics processor. Structuring a computation as stages of Map Reduce operations ensures that maximal parallelism is expressed. The global summarization provided by the Reduce operation adds generality, enabling the implementation of diverse computations.

3.1 CUDA

Our framework is built on top of the Compute Unified Device Architecture (CUDA) programming environment which Nvidia provides as a programming environment for its GPUs [14]. The programmer codes in annotated CUDA, accelerating compute intensive portions of the application by executing them on the GPU.

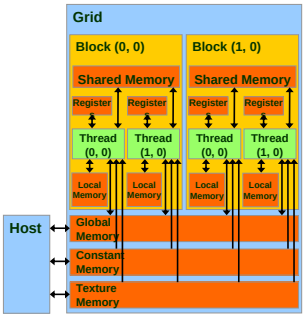

Figure 1: Logical organization of the GeForce 8800

Figure 1 illustrates how the GPU appears to the programmer. The programmer organizes the computation into grids, which are organized as a set of thread blocks. The grids run sequentially on the GPU, meaning that all computation in the grid must finish before another grid is invoked. As mentioned, grids contain thread blocks, which are batches of threads that execute together, sharing local memories and synchronizing at programmer specified barriers. A maximum of 512 threads can comprise a thread block, which puts a limit on the scope of synchronization and communication in the computation. However, enormous numbers of blocks can be launched in parallel in the grid, so that the total number of threads that can be launched in parallel is very high. In practice, we need a large number of thread blocks to ensure that the compute power of the GPU is efficiently utilized.

4. MAP REDUCE FRAMEWORK

Our framework takes advantage of the local synchronization capabilities provided by CUDA to restructure the Map Reduce into two stages. As mentioned earlier, our framework asks the programmer to provide a map function, a set of reduce operators, and a cleanup function which operates on the result of the reduction. The framework then synthesizes a function which combines the map operation with a local reduction, as well as another function which combines a global reduction with the cleanup function. The programmer writes annotated CUDA code, indicating important features of the Map Reduce to the framework, and the framework then generates CUDA code.

4.1 Predication

Before going into the details about the implementation of the framework, we need to describe how we use predication. The Google version of Map Reduce is very general, with each map function producing (key, value) pairs, and each reduction function working on sets of (key, value) pairs.

To provide a limited version of this flexibility, while still achieving high performance on the GPU, we use predica-

tion. The Map function is required to produce a set of output values, as well as a set of predicates for each thread. The predicates control how each output from the Map function participates in each reduction operation. In our current implementation, these predicates are contained in one integer per thread. Each bit of the predicate controls whether the Map function output represented by the predicate will participate in a particular reduction. In a sense, we are restricting the Google style Map Reduce by assuming that the keys are known *a priori*, and that each map function produces only one output. Since the current implementation uses only one integer per thread for predication, we are also assuming that there are less than 32 keys active in any given Map Reduce call, although this restriction is not inherent to our approach and could be lifted, if necessary. Although this functionality is somewhat restricted compared to the Google Map Reduce, it is well suited to the GPU architecture, and provides enough flexibility to be useful.

4.2 Assumptions about Reductions

It is also important to be clear about what kinds of assumptions our framework makes about reduction operators. Firstly, we assume that each reduction operator is a binary operator. We allow each input to each reduction function to be a set of data, meaning that the input and output of the operator may be essentially a structure. A reduction operator can perform reductions on that structure in arbitrary ways, but the reduction must depend on exactly two input structures, and produce exactly one output structure.

Secondly, we assume that the reduction functions are associative[1]. This assumption enables us to restructure the reduction into a binary tree, in order to extract maximal parallelism from the reduction. Commutativity is not required.

Thirdly, we assume that the programmer can specify identities for each reduction operator. Performing the reduction operator on two inputs, one of which is an identity for the reduction operator, should pass the other input through to the output of the reduction unchanged.

4.3 Map Implementation

In our current implementation, the Map function is arbitrary CUDA code that produces a set of outputs and predicates. The goal of our framework is not to hide GPU programming from the programmers completely, but to help the programmers achieve good performance with lower effort.

The Map function must produce the outputs and predicates in local memory, and indicate them to our code generator. This is illustrated in appendix A. As mentioned earlier, the predicates are handled using an array of 32-bit integers, with one entry in the array per map thread. Each thread sets the i^{th} bit in its corresponding predicate to denote that the data generated by the thread must take part in the i^{th} reduce function.

4.4 Reduce Implementation

Generally, parallel reductions are implemented in logarithmic stages. In each stage the number of threads taking part in the reduction halves until just one thread is active, which contains the result of the reduction. In our framework, reduce functions act on the output of the map functions in local memory, and using the predicate array, locally reduce the data within each block of threads. When more data participates in the reduction than can fit in a single block of threads, multiple reduction stages must be performed, as illustrated in figure 2. The framework automatically generates the reduction code, including intermediate data structures to connect the different stages of the reduction.

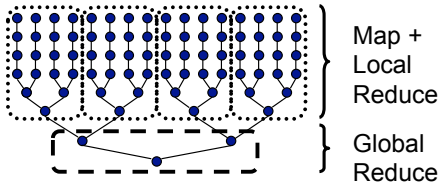

Figure 2: Structuring the Map Reduce

There are myriad ways of implementing reductions, using different varieties of loop unrolling, serializing the reduction to differing degrees, etc. At present, our framework can generate two different reduction methods, which vary in loop unrolling and number of loads per thread. The programmer can choose which one is executed using a template parameter.

To understand the complexities involved in reduction, see table 1. For illustrative purposes, we considered three different reduction methods, and then instantiated them with three different numbers of threads per thread block. It should be noted that this is not an exhaustive search on all possible parallel reduce algorithms possible on a GPU/CPU system, but an example to point out the complexity involved. A brief description of the methods is given below:

Method 1 does no loop unrolling, uses sequential addressing and does one memory load per thread.

Method 2 does full loop unrolling and two memory loads per thread.

Method 3 does full loop unrolling and partially serializes the reductions.

Table 1: GPU Reduction Complexity

Array Size	Best case		Worst case	
	Kernel (blocksize)	Time taken (ms)	Kernel (blocksize)	Time taken (ms)
65,536	3 (128)	0.02316	1 (128)	0.08373
8,388,608	1 (128)	0.08992	2 (256)	0.8453
33,554,432	2 (256)	0.03140	3 (128)	1.8689

As shown in table 1, choosing a single reduction method *a priori*, that is optimal for a particular data set size, can yield performance that is up to 60× worse than the optimal method for a different data set size. The optimization space is complex and it is not easy to get the best performance without tuning. This motivates the need for a Map Reduce framework: implementing many different reduction methods in the framework keeps the programmer from having to understand and implement these low-level, yet performance critical details.

Other work has been done on how to perform global operations such as reductions efficiently on the GPU, e.g. [4], which explains how map, reduce, scan and sort can be implemented on a GPU using the Graphics API. Other recent work includes efficient scan primitives [18], which details new approaches for implementing sort and scan. Our approach

[1]Or at least pseudo-associative, like floating point addition

differs from these by making use of a code generator to integrate these primitive operations tightly into the code, reducing overhead and increasing the scope of applications to which these operations can be profitably applied.

4.5 Autotuning

Programming the GPU involves fixing values for numerous parameters which must be tuned to get the best performance. Parameter tuning is an established technique for CPUs in High Performance Computing. For example, tools like PHiPAC [3] do autotuning for Linear Algebra operations according to cache sizes on different processors. A similar tool for optimizing thread block sizes and reduction kernels is needed for our framework.

GPU performance can be strongly affected by the number of registers utilized in a function, the amount of local memory which is used by a function, the number of threads in a block, as well as the algorithm used to perform the computation, among other factors. Because these parameters affect performance differently, for different data set sizes, the answer to the performance tuning question depends on the data set size. We allow the programmer to expose these choices through template parameters, and are developing an integrated autotuning framework to identify the best choices for different data set sizes. This work is in progress and we expect to have automated support for parameter tuning soon. The framework should be able to generate a switch matrix which chooses the best tuning parameters for a particular data set size, thus creating a single binary which performs well across a wide range of problems.

5. EXAMPLE APPLICATIONS

To demonstrate our framework, we have implemented two applications: Support Vector Machine training and classification. Support Vector Machine training is an iterative Quadratic Programming solver, which has many tight loops with relatively small Map Reduce computations in each loop. Support Vector Machine classification is performed using BLAS3 operations followed by a Map Reduce computation to finish the classification.

5.1 Support Vector Machine Training

We consider the standard two-class soft-margin SVM classification problem, which classifies a given data point $x \in \mathbb{R}^n$ by assigning a label $y \in \{-1, 1\}$ [6]. This problem has found widespread use in diverse fields, such as image recognition, bioinformatics, text processing, and network security, among others.

The SVM training problem is a Quadratic Programming optimization problem, which can be solved by many methods, each with different parallelism implications. Given a labeled training set, the goal is to find an optimal weight α_i for each training point x_i. The weights and training set then constitute a classifier. We have implemented the Sequential Minimal Optimization (SMO) algorithm, first proposed by Platt [15], with the improved first-order variable selection heuristic proposed by Keerthi [11], and kernel caching as proposed by Joachims [10]. The SMO algorithm is a specialized optimization approach for the SVM training Quadratic Program, which takes advantage of the sparse nature of the α weights and the simple nature of the constraints in the SVM QP to reduce each optimization step to its minimum form: updating two α_i weights. The bulk of the computation is then to update the Karush-Kuhn-Tucker optimality conditions for the remaining set of data points (map) and then find the two maximally violating weights (reduce), which are then optimized. It is important to note that the reduction to find the two maximally violating weights is done over dynamic, data-dependent subsets of the points, which is accomplished via predication. When all points satisfy the optimality conditions to a given tolerance, the algorithm terminates.

Algorithm 1 SVM Training: SMO

Input: training data x_i, labels y_i
Initialize: $\alpha_i = 0$, $f_i = -y_i$
Find maximally violating pair I_{low}, I_{high}
Optimize $\alpha_{I_{high}}$ and $\alpha_{I_{low}}$
repeat
  Map: update f_i, $\forall i \in \{1..l\}$
  Reduce: compute b_{high}, I_{high}, b_{low}, I_{low}
  Cleanup: Optimize $\alpha_{I_{high}}$ and $\alpha_{I_{low}}$
until $b_{low} \leq b_{high} + 2\tau$

5.2 Support Vector Machine Classification

The SVM classification problem evaluates an unknown point z_j with respect to the decision surface constructed in the training process, in order to classify z_j into one of the two classes. More specifically, the SVM classification problem is the following: for each data point $z_j \in \mathbb{R}^n$ which should be classified, compute

$$\hat{z}_j = \text{sgn}\left\{ b + \sum_{i=1}^{l} y_i \alpha_i \Phi(x_i, z_j) \right\} \quad (1)$$

where b is an offset derived from the solution to the SVM training problem shown earlier, $\Phi()$ is a kernel function which varies according to the problem[2], and all other variables remain as previously defined in the SVM training problem.

We approached the SVM classification problem by making use of Map Reduce as well as Nvidia's matrix matrix multiplication routine (SGEMM). We recast the kernel function evaluations between the set of unknown points and the training points into dot products, which are then computed via SGEMM. The Map Reduce framework is then used to finish the classification: the map function completes the kernel evaluations $\Phi(x_i, z_j)$, and multiplies them by $y_i \alpha_i$. The reduce function computes the sum for each z_j, after which b is added to obtain the final classification as given by equation (1).

[2]we use the most commonly used kernel function, the RBF kernel: $\Phi(x_i, z_j) = e^{-\gamma ||x_i - z_j||^2}$

6. RESULTS

We compare the results of the SVM training and classification applications on a GPU with those on a CPU using LibSVM. LibSVM is a popular software package for SVM training and classification problems [8]. It uses the SMO algorithm with several additional optimizations which we have not yet implemented.

Experiments were run on an Intel Core 2 Duo 2.66 GHz processor, and we gave LibSVM a cache size of 650 MB, which is slightly larger than our GPU implementation was allowed. The GPU used in the experiments was the Nvidia 8800 GTX, featuring 768MB of memory and 128 stream processors, running at 1.35 GHz. CPU-GPU communication overhead is included in runtimes.

6.1 SVM Training

Table 2 contains performance results for the two solvers. We see speedups in all cases from 5× to 32×.

Table 2: Comparison of GPU vs LibSVM solve times

Dataset	GPU (sec)	LibSVM (sec)	Speedup
Adult [2]	36.312	550.178	15.1
Web [15]	181.334	2422.469	13.4
MNIST [12]	525.783	16965.794	32.3
USPS [9]	0.733	5.092	6.9
Forest [2]	13360.785	66523.538	5.0
Face [17]	2.57	27.61	10.7

6.2 Classification

Results for our classifier are presented in table 3. We achieve 120-150x speedup over LibSVM on the datasets shown.

Table 3: Performance and accuracy of GPU SVM classification vs. LibSVM

Dataset	GPU (sec)	LibSVM (sec)	Speedup
Adult	0.570	75.65	132.5
Web	1.069	144.53	135.2
MNIST	1.98	258.751	130.7
USPS	0.0097	1.194	123.2
Face	0.706	109.259	154.8

6.3 Productivity

The SVM training code requires 380 lines of kernel code to write in our framework, whereas the output of our code generator is 574 lines. The SVM classification code requires 72 lines of kernel code using our framework, whereas the output of our code generator is 201 lines. The output of the code generator is close to a hand-coded version in terms of amount of code generated. From our experience, this represents a significant reduction in the amount of code to be written by the user, especially since writing the reductions is so difficult on the GPU.

7. CONCLUSION

Using the Map Reduce abstraction, we have created a code generation framework to help application developers program Graphics Processors. The Map Reduce abstraction captures parallelism at a high level, while usefully constraining the application, enabling high performance parallel implementations. Using our framework, productivity is improved, since the programmer does not need to restructure their algorithm around the synchronization and communication limitations of the GPU. Additionally, the programmer is not required to understand the details of how the reduction is performed, which can be complicated and performance critical. Code created by our framework is efficient, providing up to 32× and 150× speedup on two different applications, compared to widely used CPU implementations. Future work includes completing the autotuner, and broadening the framework to include parallel prefix operations, which will make it still more general.

8. REFERENCES

[1] K. Asanović, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] A. Asuncion and D. Newman. UCI machine learning repository, 2007.

[3] J. Bilmes, K. Asanović, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. In *Proceedings of International Conference on Supercomputing*, Vienna, Austria, July 1997.

[4] I. Buck and T. Purcell. A toolkit for computation on GPUs. In R. Fernando, editor, *GPU Gems*, chapter 37, pages 621–636. Addison Wesley, 2004.

[5] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In B. Schölkopf, J. Platt, and T. Hoffman, editors, *Advances in Neural Information Processing Systems 19*, pages 281–288. MIT Press, Cambridge, MA, 2007.

[6] C. Cortes and V. Vapnik. Support-vector networks. *Mach. Learn.*, 20(3):273–297, 1995.

[7] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04*, Berkeley, CA, USA, 2004. USENIX Association.

[8] R.-E. Fan, P.-H. Chen, and C.-J. Lin. Working set selection using second order information for training support vector machines. *J. Mach. Learn. Res.*, 6:1889–1918, 2005.

[9] J. J. Hull. A database for handwritten text recognition research. *IEEE Trans. Pattern Anal. Mach. Intell.*, 16(5):550–554, 1994.

[10] T. Joachims. Making large-scale support vector machine learning practical. In *Advances in kernel methods: support vector learning*. MIT Press, Cambridge, MA, USA, 1999.

[11] S. S. Keerthi, S. K. Shevade, C. Bhattacharyya, and K. R. K. Murthy. Improvements to Platt's SMO Algorithm for SVM Classifier Design. *Neural Comput.*, 13(3):637–649, 2001.

[12] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[13] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: A programming model for heterogeneous multi-core systems. *ASPLOS 2008*, 2008.

[14] Nvidia. Nvidia CUDA, 2007. `http://nvidia.com/cuda`.

[15] J. C. Platt. Fast training of support vector machines using sequential minimal optimization. In *Advances in kernel methods: support vector learning*, pages 185–208. MIT Press, Cambridge, MA, USA, 1999.

[16] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. *HPCA 2007*, pages 13–24, 10-14 Feb. 2007.

[17] H. A. Rowley, S. Baluja, and T. Kanade. Neural network-based face detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(1):23–38, 1998.

[18] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *GH '07: ACM SIGGRAPH Symposium on Graphics Hardware*, pages 97–106, 2007.

[19] G. L. Steele, Jr. Parallelism in Lisp. *SIGPLAN Lisp Pointers*, VIII(2):1–14, 1995.

APPENDIX

A. SYNTAX

To make the framework more concrete, we illustrate how it is used with a few code fragments.

```
MapReduce(

  mapFunction
    <...template parameters...>
    <<<...CUDA parameters...>>>
    (...function parameters...)

  reduceFunction0, ..., reduceFunctionN,

  cleanupFunction
    <...template parameters...>
    <<<...CUDA parameters...>>>
    (...function parameters...)

);
```

Figure 3: Instantiating a Map Reduce

When instantiating a Map Reduce, the programmer provides a list of the functions which are to be used by the framework. For the map function and the cleanup function, the programmer also fills out the various parameter lists with the values needed for the instantiation. As mentioned earlier, we use template parameters for some code generation, which allows the same source code to generate several different versions. If the programmer notifies the framework of the set of values each template parameter can take, the framework will then instantiate all possible versions of the template parameters in the code automatically. This essentially does limited constant propagation and dead code elimination for different paths through the code, which can provide significant performance boosts on a register constrained architecture such as the GPU.

The programmer also provides the CUDA parameters, specifying the dimensions of parallelism in the map and cleanup functions. Runtime dynamic local memory allocation is taken care of by the framework, and appended to the CUDA parameter list.

```
__shared__ float xILow[nDimension];
    /*dynamic (iLowCompute)*/
__shared__ int localIndices[blockDim.x];
    /*output 0 dynamic*/
__shared__ float localFs[blockDim.x];
    /*output 1 dynamic*/
__shared__ int localFlags[blockDim.x];
    /*predicate dynamic*/
```

Figure 4: Map Declarations

The programmer indicates important details to the framework via some pre-specified comments. For example, the map function is assumed to put its outputs in local memory, but the framework needs to know which array in the map function corresponds to which output. Also, the runtime dynamic memory allocation needs to be indicated to the framework. We allow runtime dynamic memory allocation to be predicated on runtime variables, as shown by xILow in figure 4, which keeps local memory usage to a minimum, allowing greater effective parallelism.

```
void maxArgmax(
    int in0Index, float in0Value, //input0
    int in1Index, float in1Value, //input1
    int* outIndex, float* outValue) //output
{
  *outValue = NINFTY; /*identity*/
  if (in1Value > in0Value) {
    *outIndex = in1Index;
    *outValue = in1Value;
  } else {
    *outIndex = in0Index;
    *outValue = in0Value;
  }
}
```

Figure 5: An Example Reduction Operator

Figure 5 illustrates how the programmer writes a reduction operator. Each input to the reduction is composed of a set of data, in this case an int and a float. The ordering of this set is determined by the declarations in the map function: output 0 from the map function is the first item in the set, etc. The programmer specifies the identity for the reduction operator by assigning the outputs to their identity value, and annotating the assignment. The rest of the operator is self explanatory.

```
float bLow; /*reduce 0 result 1*/
float bHigh; /*reduce 1 result 1*/
int iLow; /*reduce 0 result 0*/
int iHigh; /*reduce 1 result 0*/
```

Figure 6: Accessing Reduction Results

Figure 6 shows how the cleanup function accesses the results of the reduction. Variables are annotated to indicate exactly which output from which reduction they should receive.