B629- DOMAIN-SPECIFIC LANGUAGES AND COMPILERS

Performance Benchmarking of Accelerate and Copperhead

Sajith Sasidharan **Devarshi Ghoshal**

12/9/2011

Contents

INTRODUCTION

GPUs are becoming increasingly popular for running parallel programs. But, programming on GPUs is extremely difficult and higher level abstractions to program on GPUs would minimize the effort on writing complex programs. Domain-specific high-level languages provide such abstractions. It is important to analyze the performance of such DSLs. In this work, we compare the performance of two such DSLs for array computations, Copperhead and Accelerate, embedded in Python and Haskell respectively.

COPPERHEAD

Copperhead programs describe parallel computations via composition of data parallel primitives supporting both flat and nested data parallel computation on arrays of data. Copperhead programs are expressed in a subset of Python and interoperate with standard Python modules. Currently, Copperhead targets only NVIDIA GPUs and the runtime system compiles functions annotated by copperhead decorators to CUDA. It also converts simple Python arrays into copperhead arrays which are managed by the runtime. The programmer can specify the execution places using the "with" construct. Currently, Copperhead supports two execution places- one on GPU (places.gpu0) and the Python interpreter (places.here). Copperhead uses PyCUDA and CodePy to manage execution on the GPU and to provide binary caches which mitigate the overhead of runtime compilation.

ACCELERATE

Accelerate defines an embedded language of array computations embedded in Haskell for high performance computing. Computations on multi-dimensional, regular arrays are expressed in the form of parameterized collective operations (such as maps, reductions, and permutations). It includes a code generator based on CUDA skeletons of collective array operations that are instantiated at run-time and an execution engine that caches previously compiled skeleton instances and host-to-device transfers. It also parallelizes the code generation, host-to-device transfers and GPU kernel loading and configuration.

INSTALLATION CHALLENGES

Installing both Copperhead and Accelerate incurred several installation challenges due to dependency conflicts between the required packages. Accelerate uses cabal installation and major challenge was resolving the dependency conflicts due to GHC, cuda bindings to Haskell and the Criterion package for benchmarking. Similarly, for Copperhead major effort was spent resolving issues related to the BOOST library, PyCUDA and CodePy modules.

BENCHMARKS

For benchmarking the two EDSLs, we used three benchmarks which are described below.

Scalar Vector Addition

This benchmark adds the elements of two vectors as: *a*X+Y, where *a* is fixed scalar quantity and X & Y are two vectors. The corresponding Copperhead and Accelerate code is described below.

COPPERHEAD
```
@cu
def axpy(a, x, y):
 return [a * xi + yi for xi, yi in zip(x, y)]
```

ACCELERATE
```
svaAcc :: Float -> Vector Float -> Vector Float -> Acc (Vector Float)
svaAcc  alpha  xs  ys
  = let
      xs' = use xs
      ys' = use ys
    in
    Acc.zipWith (\x y -> constant alpha * x + y) xs' ys'
```

There are three things to notice for the Accelerate version:

i. The result is an Accelerate computation indicated by Acc.
ii. The two plain vectors xs & ys are lifted into Accelerate types with "use".
iii. "fold" is used instead of foldl and hence, it leaves the order in which the elements are combined unspecified and requires an associative binary operation in order to provide a deterministic execution.

Vector Dot Product

This computes the dot product of two vectors, X and Y. The code in Copperhead and Accelerate is shown below.

COPPERHEAD
```
@cu
def dot_product(x, y):
   return sum(map(op_mul, x, y))
```

ACCELERATE
```
dotpAcc :: Vector Float -> Vector Float -> Acc (Scalar Float)
dotpAcc  xs  ys
  = let
      xs' = use xs
      ys' = use ys
    in
    Acc.fold (+) 0 (Acc.zipWith (*) xs' ys')
```

Sum of Absolute Values in a Vector

The source code in both Accelerate and Copperhead is described below for calculating the sum of absolute values in a vector.

COPPERHEAD

```
@cu
def vector_sum(x):
 def el_wise(xi):
   return abs(xi)
 return sum(map(el_wise, x))
```

ACCELERATE

```
sasumAcc :: Vector Float -> Acc (Scalar Float)
sasumAcc  xs
  = Acc.fold (+) 0 . Acc.map abs $ Acc.use xs
```

CONFIGURATION

The machine and software package details on which the benchmarks were executed are described below.

- Intel Xeon CPU X5365 @3.00 GHz, 8GB RAM
- NVIDIA Tesla C1060, GeForce 7300 GT
- Gentoo Linux 3.0.4
- Python 2.7
- GHC 7.0.4
- Accelerate 0.9.0.0
- Copperhead May-2010 release
- CUDA 4.0

RESULTS

The benchmark results using both Copperhead and Accelerate are shown in the following graphs. Figure-1 and Figure-2 show the overheads of transforming and executing the parallelized code on smaller size arrays. Clearly, CUDA device initialization and host-to-device transfer overshadow the effect of parallelization and hence, for smaller arrays the CPU version of the Copperhead and Accelerate programs perform better than the ones running on GPUs.

Figure 1

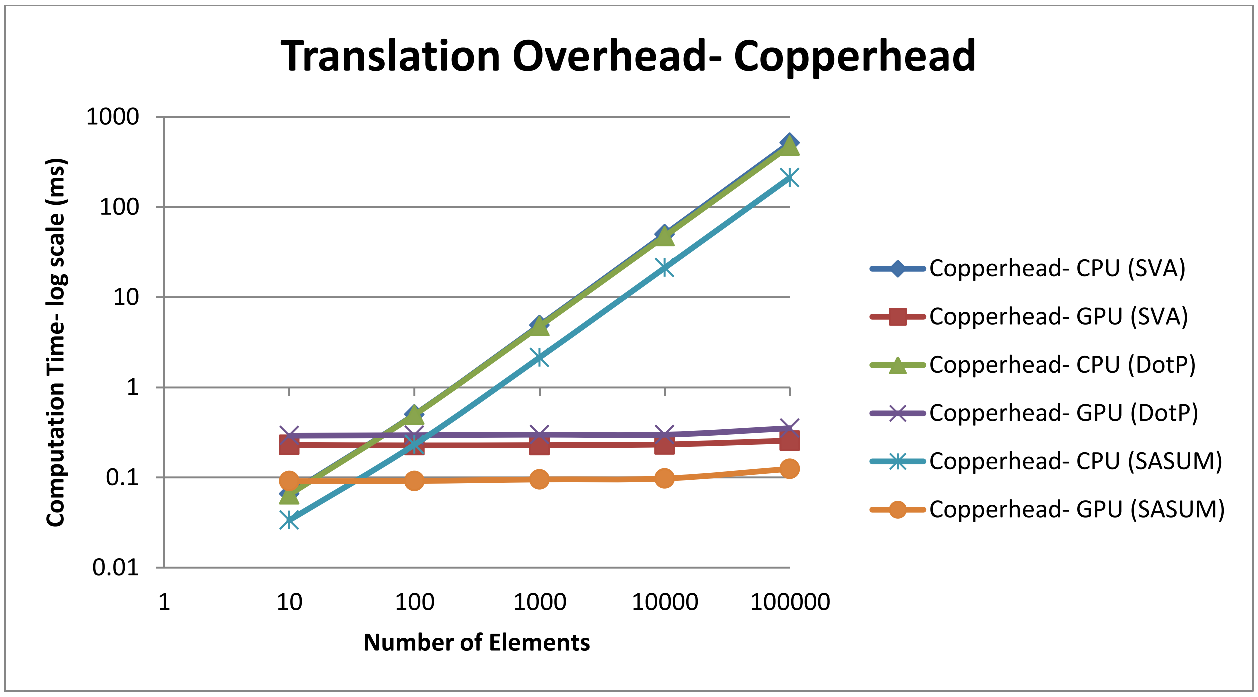


Figure 2

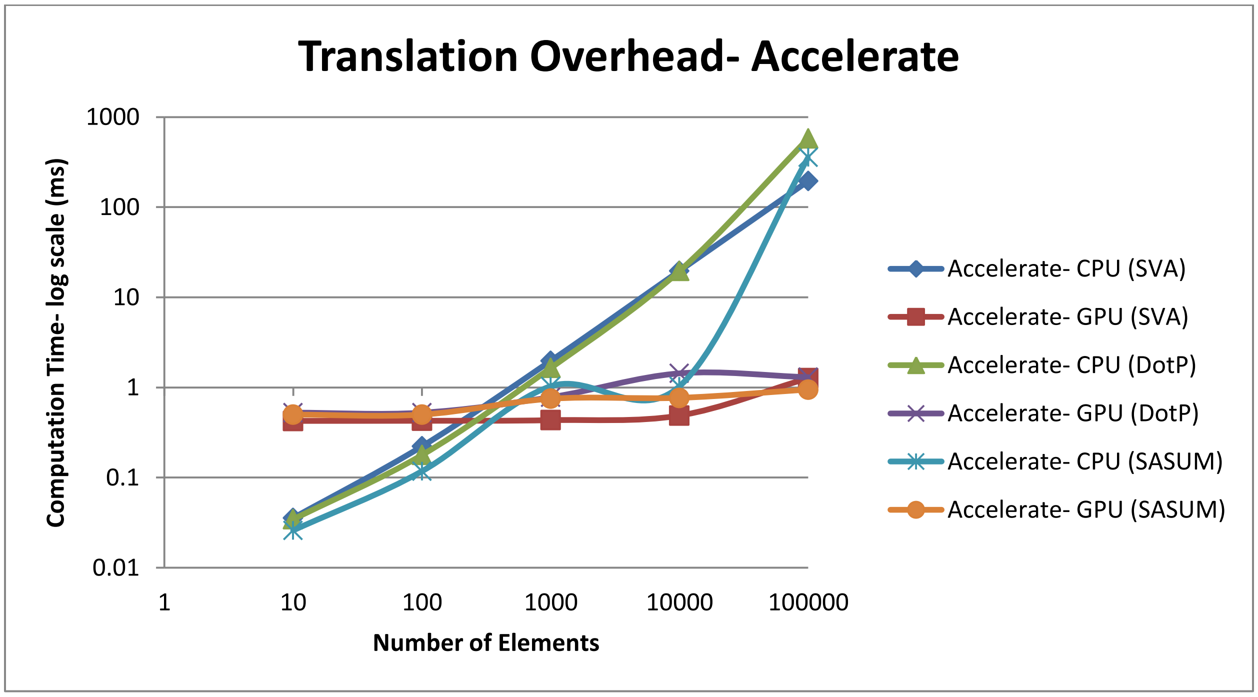


Figure-3 and Figure-4 show the relative comparison between the Copperhead and Accelerate programs running on CPUs and GPUs for larger size arrays. Clearly, the programs running on GPUs are running 10 to 100 times faster than the corresponding versions running on CPUs.

Figure 3

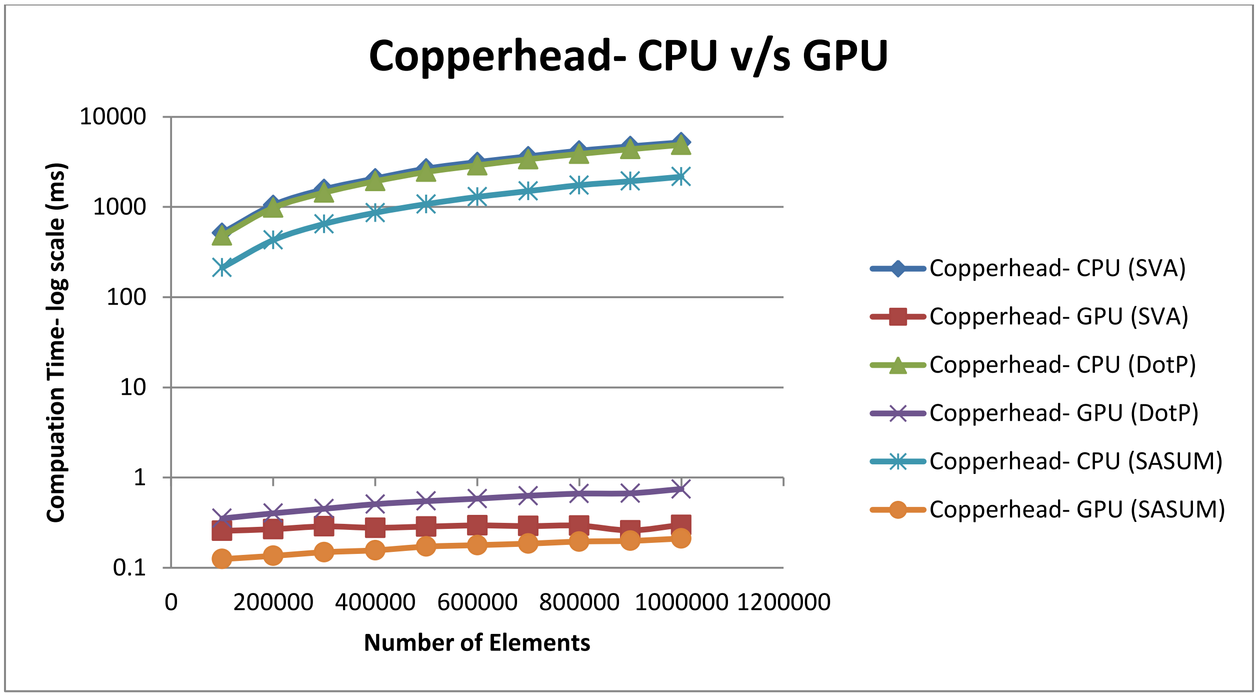
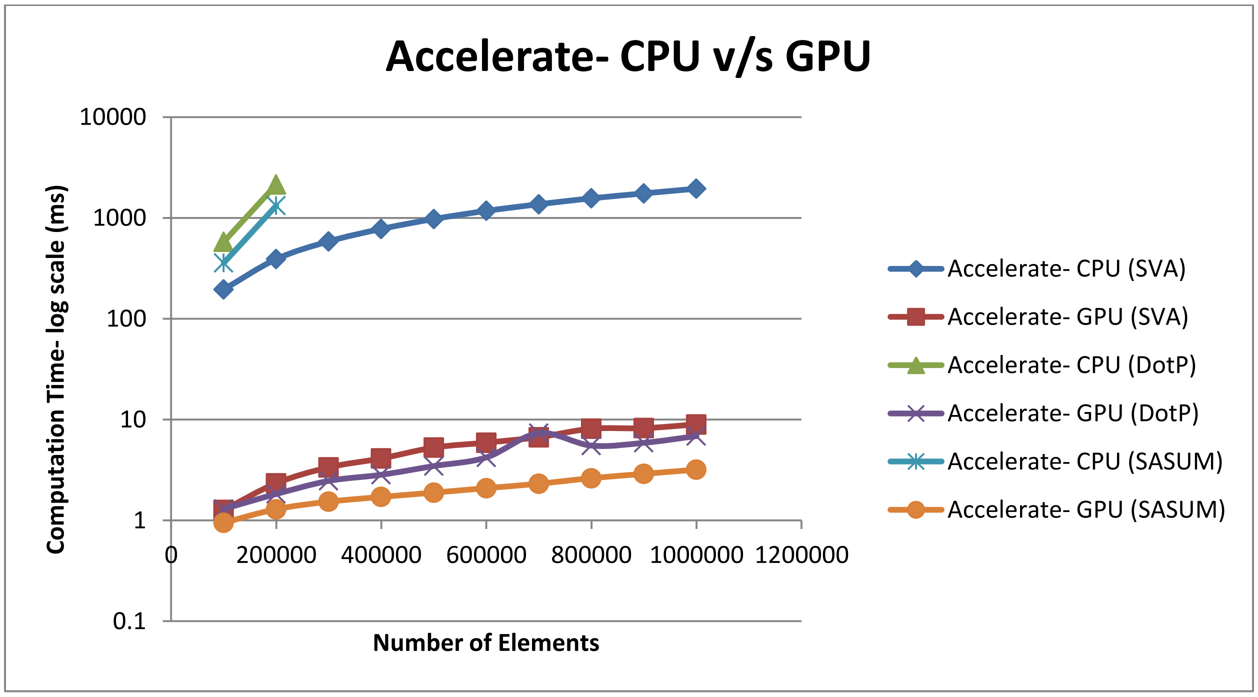


Figure 4



Finally, Figure-5 and Figure-6 show the relative performance of executing the programs using Accelerate and Copperhead. The results show that Copperhead outperforms corresponding Accelerate versions.

Figure 5

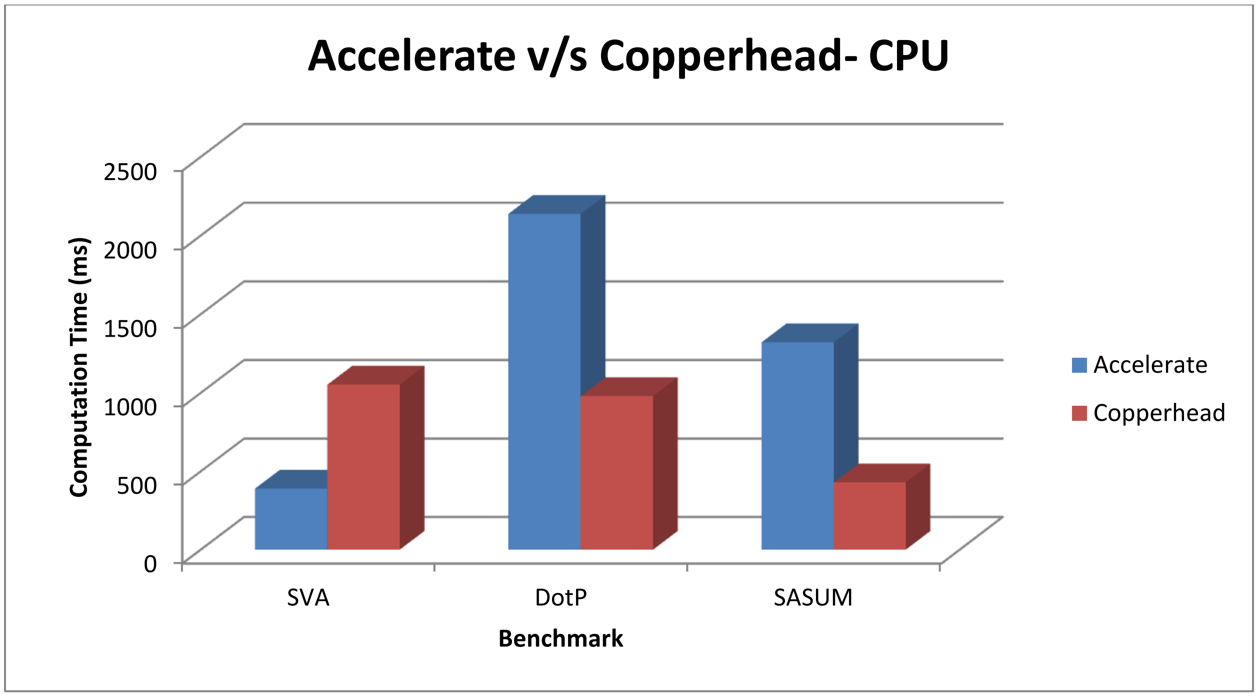
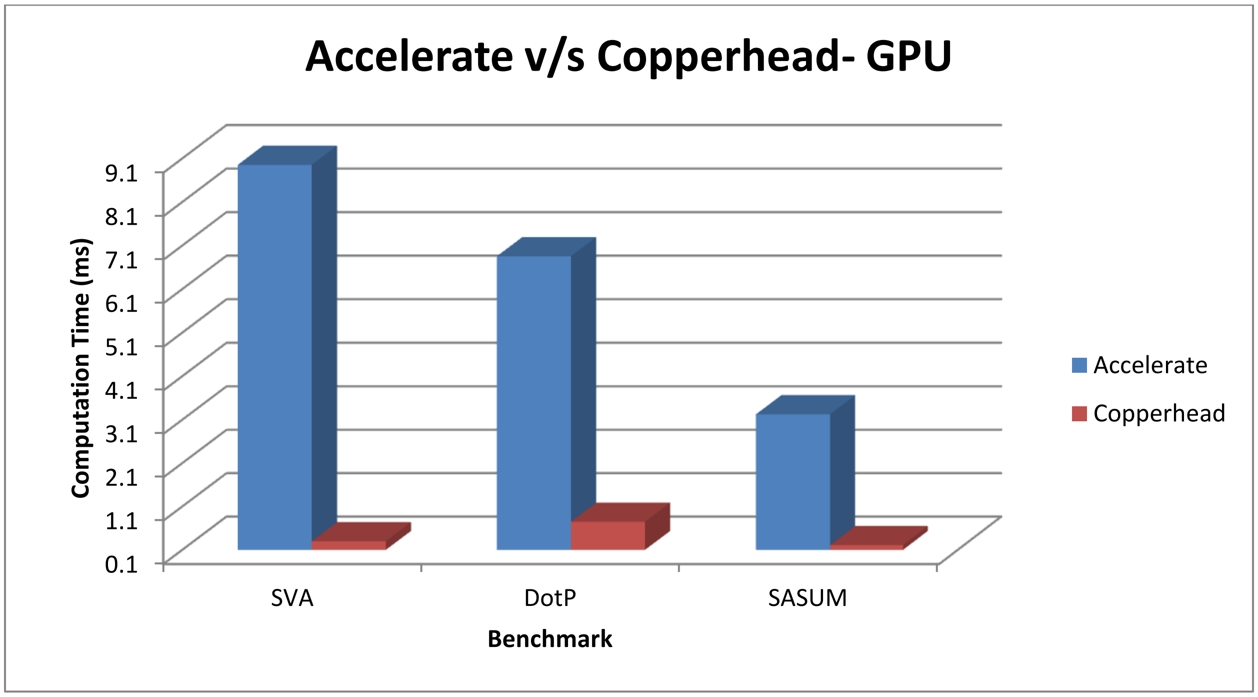
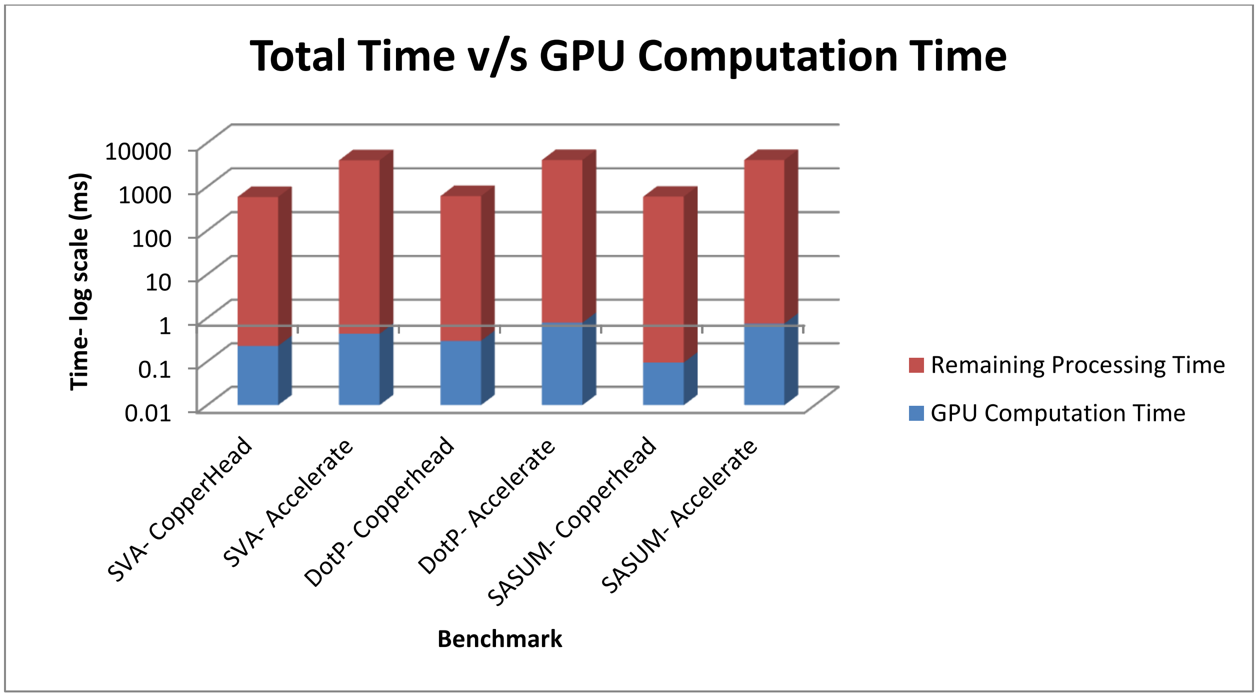


Figure 6



Additionally, Figure-7 shows the computation-to-overall execution time ratio for both Copperhead and Accelerate programs and it is evident from the graphs that both these languages incur a large overhead in transforming the code and running on GPUs.

Figure 7



CONCLUSIONS

By looking at the graphs and results, it looks like Accelerate performs badly compared to Copperhead due to the fact that the code is interpreted. Also, lifting the normal Haskell arrays or vectors to Accelerate vectors result in degrading the performance. Additionally, it has been observed that the pure Haskell and Python version of the programs perform better than both the GPU and CPU version of Accelerate and Copperhead, which again can be attributed to the fact that the translation and data-transfer overheads are high and overshadow the performance gain due to parallelization. But, further investigation and analysis is required to better understand the performance characteristics and variations for both Accelerate and Copperhead. Another important comparison could have been among these DSLs and native C/C++ CUDA code.

ACKNOWLEDGEMENTS

We would like to thank Trevor L. McDonell for helping us resolving the problems related to Accelerate and Bryan Catanzaro for answering questions related to Copperhead. Additionally, we would like to thank Prof. Ryan Newton for guiding us on this project and Prof. Arun Chauhan for letting us use the “timon” machine for running the benchmarks on GPUs.