Evaluation of autoparallelization toolkits for commodity graphics hardware

David Williams[1], Valeriu Codreanu[1], Po Yang[2], Baoquan Liu[2], Feng Dong[2], Burhan Yasar[3], Babak Mahdian[4], Alessandro Chiarini[5], Xia Zhao[6], and Jos B.T.M. Roerdink[1]

[1] University of Groningen, The Netherlands
[2] University of Bedfordshire, United Kingdom
[3] RotaSoft, Turkey
[4] ImageMetry, Czech Republic
[5] Super Computing Solutions, Italy
[6] AnSmart, United Kingdom

Abstract. In this paper we evaluate the performance of the OpenACC and Mint toolkits against C and CUDA implementations of the standard PolyBench test suite. Our analysis reveals that performance is similar in many cases, but that a certain set of code constructs impede the ability of Mint to generate optimal code. We then present some small improvements which we integrate into our own GPSME toolkit (which is derived from Mint) and show that our toolkit now out-performs OpenACC in the majority of tests.

1 Introduction

The last ten years have seen the widespread adoption of parallel computing hardware in the form of Graphics Processing Units (GPUs). These are commonly found in all consumer grade systems from high-end desktop machines down to mobile devices. Although originally designed for accelerating highly parallelizable graphics operations, these GPUs have steadily increased in programmability and have found widespread application in a number of specialist fields [1].

Writing such general purpose applications on the GPU typically requires a developer to be experienced with OpenCL or CUDA, and to have a strong understanding of the GPU's parallel architecture. This acts as a barrier to adoption in environments where such specialised knowledge is not readily available, such as many small and medium sized enterprises (SMEs) which do not have access to the needed domain experts.

The development of semi-automatic parallelization tools [2, 3] has the potential to shift this balance and encourage more wide-spread adoption of GPU acceleration. These tools typically work by augmenting the input C/C++ code with compiler directives which mark regions to be parallelized, and then automatically generating the required OpenCL/CUDA code for device initialization, memory transfers and kernel implementation. The principle is similar to that

adopted by OpenMP [4] and has proved to be a simple and practical approach to improving utilisation without requiring a high level of expertise.

In this paper we provide an evaluation of the OpenACC [3] and Mint [2] tools, with the particular aim of identifying and implementing improvements to Mint. We perform this evaluation against the PolyBench [7] test suite after adding OpenACC, Mint, and OpenMP directives. We are then able to identify the areas in which Mint is not competitive, and in the second half of the paper we present some changes which we include in our enhanced version of Mint (known as the GPSME toolkit [5]).

2 Related Work

OpenACC is a relatively new technology, with the first version of the standard being finalised in 2011 and the latest version expected during 2013. Coupled with the lack of freely available and mature implementations, this has meant the technology has not been widely evaluated by the academic community. The small number of available evaluations have focused primarily on small test cases [15] though some application to real-world code has also been performed [16] [17]. In all cases significant speedup was observed on sections of parallelizable code, and it should be noted that the OpenACC compilers are still undergoing rapid development due to the standard being so new.

To our knowledge, work evaluating Mint has been limited to that undertaken by its authors. The original Mint paper [2] claimed its performance was twice that of the PGI Accelerator model when using 3D heat simulation as a test case. The PGI Accelerator model was the predecessor to PGI's OpenACC compiler, so it is interesting to see how the performance has changed. A later paper presented an application of Mint to earthquake simulation and demonstrated an order of magnitude performance increase over the CPU reference implementation [10].

We have chosen to use an existing benchmark suite rather than to design our own, in order to minimise the bias which would be implicit in such a process. The PolyBench polyhedral benchmark suite [7] was designed to test the performance of a number of kernels from various application domains, and was recently extended with GPU implementations of most of the tests [8]. This provides an ideal basis for evaluating the performance of autoparallelization tools.

3 Methodology

The PolyBench test suite contains 15 test programs in the domains of convolution, linear algebra, data mining and stencil operations. The original implementations [7] were in C but GPU implementations in OpenCL and CUDA were added later [8] by Grauer-Gray *et al.* We do not consider these GPU implementations to be optimal as we were able to obtain equal or better results than most of them using our autoparallelization tools (see Section 4), but they still provide a useful reference point in the performance analysis. We consider only

the CUDA implementation as there are known differences between OpenCL and CUDA performance [11, 12], and our tools use CUDA as a backend anyway.

Within this work we will sometimes need to refer to individual tests within PolyBench. We do this by using the name which PolyBench assigns to each test, written in upper case letters. For example, the 'ATAX' test uses matrix transpose and vector multiplications while the 'SYRK' test contains symmetric rank-k operations. A full list of test names and descriptions is provided with PolyBench [7].

We are primarily concerned with measuring the performance of Mint and OpenACC with respect to each other, but a comparison to CPU performance is also useful as a baseline. In the interests of fairness this CPU implementation should take advantage of all available cores and threading opportunities. We have therefore added OpenMP directives to each of the PolyBench tests.

The performance of CUDA programs (and, by extension, the output of our tools) is often highly dependent on the way in which the problem space is partitioned into thread blocks within the application. The optimal size for these thread blocks depends on a number of factors including the nature of the work done in the kernel, the need for synchronisation between threads, the data access pattern, and the target hardware. Prior to performing the evaluation we wrote an automated system to compile each test in a large number of thread block configurations, and then chose to perform our tests using the configuration which showed the greatest performance. A similar benchmarking approach is described in [14].

3.1 Modifications to PolyBench

A few of the benchmarks required minor modifications in order for their computational pattern to be successfully captured by the Mint and OpenACC programming models. These modifications affected the ATAX, BICG, and GRAMSCHMIDT tests as they contained two or three-level nested loops with inter-loop dependencies. These were resolved by splitting the loop into two successive nested loops. The modifications affected all the parallelization tools equally as each test is only implemented once but with multiple sets of pragmas applied.

Additionally, we adjusted the timing code for the CUDA manual implementations to be inclusive of the data transfer time as this was previously omitted. The time taken to transfer data to and from the GPU is often significant and can dominate algorithm runtime in some cases [13]. By accounting for this we help ensure a fair comparison.

3.2 Test configuration

The test machine is comprised of a quad-core Intel CPU and an NVidia GPU as detailed in Table 1. All components are from the same generation hardware to make the comparison as fair as possible. Table 1 also specifies the operating system and compiler setup, with all tests being compiled on maximum optimization levels.

Test configuration	
CPU Setup:	Intel Core i7-2600K 3.4 GHz
GPU Setup:	NVidia GTX680
Operating System:	Ubuntu 12.04 LTS
C/OpenMP Compiler:	GCC 4.6
CUDA Compiler:	NVCC (From CUDA 5.0 SDK)
OpenACC Compiler:	PGI Compiler 13.1

Table 1: System details

4 Initial Results

Initial performance measurements for the CUDA, Mint and OpenACC versions of each of the 15 tests are shown in Figure 1. These measurements are all shown relative to the baseline set by the OpenMP implementation running on the CPU and using all cores. Note that the tests are performed using the default dataset size provided by the PolyBench suite.

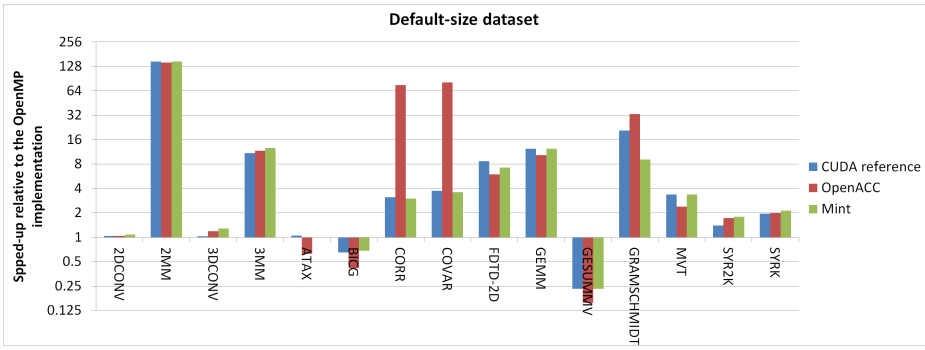

Fig. 1: Speed-up of GPU implementations compared to OpenMP on the default dataset. The values are in logarithmic scale.

It can be seen that for many tests the performance of the three implementations is similar. Among these cases there is usually a small amount of variation in the exact performance distribution. When the CUDA version is *faster* it can be attributed to missed optimization opportunities in the autogenerated code, and when the CUDA version is *slower* we found that autogenerated code was using pitched memory allocations to reduce the memory coalescing penalties. This penalty is heavily incurred in the case of SYR2K, the manual version being more than 30% slower than the automatic ones, with the kernel code being similar.

Furthermore we can observe that the performance of Mint and OpenACC is generally very similar, but that Mint has a slight edge in the majority of cases.

An analysis of the output code has suggested that this is due to better register usage in the Mint code, as well as some redundant instructions in the OpenACC version (probably reflecting its more general-purpose nature).

More interestingly, there are a few tests which are notably different from the generalizations described above. Most striking is the large performance difference between OpenACC and Mint in the CORR, COVAR and GRAMSCHMIDT tests. This can be explained by the *triangular* nature of the nested loops in these tests, and is further discussed in Section 5.1 where we address this problem.

Three of the tests (ATAX, BICG, GESUMMV) actually showed reduced performance when running on the GPU. These tests made use of a one-dimensional thread block which did not contain enough work to benefit from offloading to the GPU, and the additional communication overhead caused an overall slowdown. However, it can be observed that Mint performed significantly better than OpenACC in these cases, and this is due to its support of the *tile* and *chunksize* parameters. OpenACC could not have the same degree of control with the *vector*, *worker* and *gang* parameters, and will include a tile parameter in OpenACC version 2.0.

There are other differences arising from the organization of the parallel loops. The 3DCONV benchmark is composed of a three-level nested for-loop. In the manual CUDA version the outer for-loop is iterated on the CPU, and only the inner two levels of the loop nest are offloaded as a 2D thread block. The automatic approaches use a 3D thread block, reducing thus the CPU−GPU communication. The same happens with the CORR, COVAR and GRAMSCHMIDT examples.

Before moving on to make improvements to Mint in the next section, we first ran the tests again with all the dataset sizes doubled in each dimension. All other parameters were left the same and the results can be seen in Figure 2.

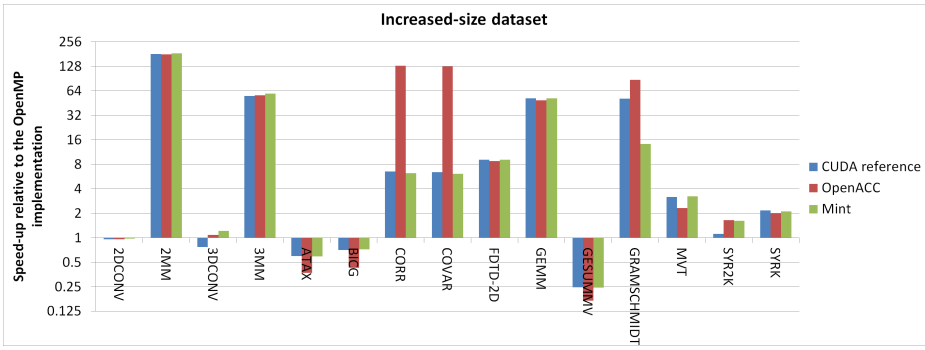

Fig. 2: Speed-up of GPU implementations compared to OpenMP on the enlarged dataset. The values are in logarithmic scale.

In these results for the enlarged dataset we can see that the speed difference between the GPU approaches and the CPU implementation is increasing when compared to the default dataset. This happens because most of the benchmarked

problems have a complexity which is quadratic with respect to the dataset size and this increases the amount of work performed by each thread. Even at this dataset size, the ATAX, BICG and GESUMMV tests perform better on the CPU, but the Mint model continues to provide faster code than OpenACC.

5 Mint Enhancements

As part of our GPSME project [5] we have developed a number of extensions to Mint to create a new tool known as the 'GPSME toolkit'. We have added significant functionality (C++ and multi-file support, preliminary OpenCL output, etc.) but this is not used for the PolyBench tests and is not the focus of this paper. Instead, we wish to use the insight we have gained in Section 4 to improve the performance characteristics of our toolkit relative to OpenACC.

5.1 Supporting triangular loops

In Section 4 it was stated that three of the tests (CORR, COVAR and GRAM-SCHMIDT) suffered from poor performance in Mint due to the usage of *triangular loops*. A problematic section of code from the COVAR test is shown in Algorithm 1, complete with the Mint directives which were added.

Algorithm 1 The covariance code from PolyBench with Mint directives added.

```
#pragma mint copy(data,toDevice, M, N)
#pragma mint copy(mean,toDevice, M)
#pragma mint copy(symmat,toDevice, M, N)
#pragma mint parallel
{
  ... //Some code omitted for brevity

  /* Calculate the m * m covariance matrix. */
  #pragma mint for nest(2) tile(16, 16)
  for (j1 = 0; j1 < M; j1++)
  {
    for (j2 = j1; j2 < M; j2++)
    {
      ... //Some code omitted for brevity
    }
  }
}
#pragma mint copy(symmat,fromDevice, M, N)
```

The key issue is the dependency of the initial value of $j2$ in the inner loop on the current value of $j1$ in the outer loop. This was enough to prevent Mint from generating valid CUDA code when attempting to specify that both loop levels

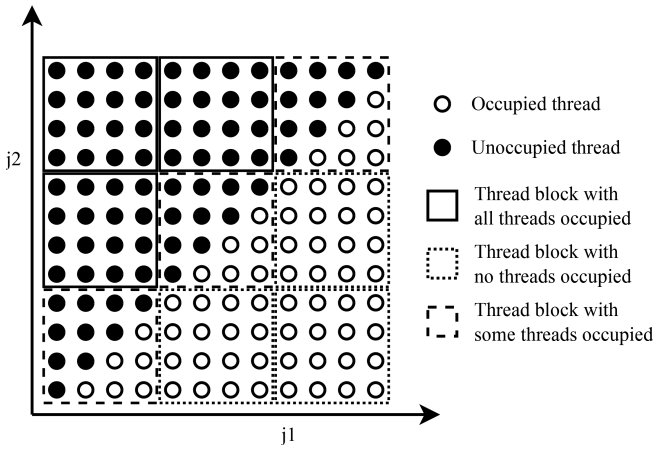

Fig. 3: Iteration space of the two-level covariance loop

should be parallelized. Replacing 'nest(2)' with 'nest(1)' allowed parallelization of only the outermost loop to proceed as expected, but the performance was significantly less than that obtained from OpenACC (see Section 4).

An extension in our GPSME toolkit has allowed this situation to be handled naturally. A rectangular iteration space is defined by the full range of values which $j1$ and $j2$ can assume, and just over half of these points fall within the triangular region processed by the test (see Figure 3). A grid of CUDA thread blocks is overlaid on the rectangular iteration, and the CUDA kernel contains a test to determine whether a given set of thread indices actually form part of the triangular region. With this in mind, a thread block can be categorized as being in one of three states with respect to the number of threads which need to execute:

- **Full:** All threads are part of the triangular iteration space and must be executed. No processing capability is wasted in this scenario.
- **Empty:** None of the threads are part of the triangular iteration space. All threads will fail the membership test implemented in the kernel and return immediately.
- **Half-full:** In this case the running time of the thread block is determined by the threads which do need to run. Threads which do not need to run must still wait upon those that do, and this represents some wasted processing capability.

With this new addition, the performance of the output code is greater than the one generated with OpenACC, and is more than 30 times faster than the one generated by the base Mint (full results presented later in Section 6).

The number of idle processing elements in the 'half-full' category is dependant upon the size of the thread block, and so a smaller thread block size results

in better utilization. However, CUDA applications in general benefit from making thread blocks rather large (in the absence of synchronization concerns) and this outweighs the benefits of better utilization for the CORR, COVAR and GRAMSCHMIDT examples.

5.2 Single-dimensional vs. multi-dimensional arrays

Another advantage to the GPSME model is that it finds more optimization opportunities when applied to code that uses multi-dimensional arrays. The optimizations are in terms of better register reuse, as well as better shared memory usage. We've tested this assumption on some of the tests. For the 2MM and SYR2K tests, a further 25% performance increase is obtained when using two-dimensional addressing instead of the default flattened array addressing.

The changes from single-dimensional to multi-dimensional array accesses were done in a manual manner, as in Polybench all tests are written with flattened array accesses. However, with extra hints from the programmer the GPSME toolkit should be able to treat the single dimensional arrays as multi-dimensional ones.

An interesting observation is that when faced with the same two-dimensional arrays in the 2MM and SYR2K tests, the OpenACC compiler reports more than two times worse performance, as can be observed in Table 2. The reasons for this are not currently clear and will be the subject of some future investigation.

	2MM-1D [s]	2MM-2D [s]	SYR2K-1D [s]	SYR2K-2D [s]
OpenACC	3.921	8.927	16.671	32.272
GPSME	3.814	2.812	17.01	12.08

Table 2: Timing improvement for the 2MM benchmark

6 Final Results

The results obtained after implementing the proposed enhancements are presented in Figure 4. The tests which benefited from our enhancements are shown in strong colours, with other tests faded out to indicated that they have not changed since the initial results. The improvements of GPSME over Mint is shown by the hatched bars. These examples rely on triangular loop support, and our improvement has enhanced their performance dramatically.

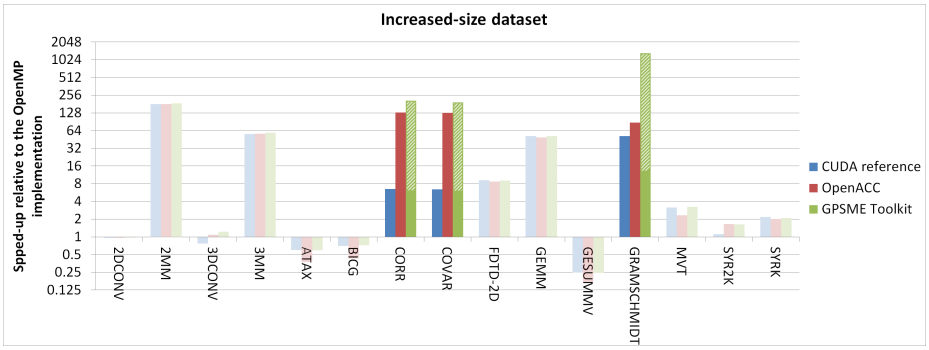

Fig. 4: Speed-up of GPU implementations compared to OpenMP on the enlarged dataset. The faded bars correspond to tests which have not changed since Figure 2, and for the three tests which have changed the hatched bars show how GPSME has improved over Mint.

We note that the improvements observed from multi-dimensional addressing are not included in this table, as it is not implemented at this point as an automatic transformation.

7 Conclusions

Automatic parallelization through compiler directives is proving to be an effective method of maximising computing resources, and we expect that the coming years will see the approaches achieving the kind of widespread adoption that we currently see with OpenMP. We have shown that both OpenACC and also Mint/GPSME are capable of delivering code with a performance to meet or exceed that provided by the hand-written code supplied with PolyBench, and that the modifications presented in this paper have been enough to push it into the lead on the PolyBench tests.

Performance between OpenACC and the GPSME toolkit is very close in most cases, which is to be expected since they perform a similar set of operations and are running on the same hardware. While we do show slightly greater performance for our toolkit it should also be noted that OpenACC is more generic, and so may perform better on other applications. The triangular loop support added in this paper is of better quality than the one offered within OpenACC, with the three tests affected exhibiting the largest performance difference.

Future work will revolve around automating some changes which were made manually for the purpose of this paper (such as the enhancements in Section 5.2), as well as identifying further opportunities for optimisation. We will also continue working with a number of small and medium enterprises to apply our toolkit to some real-world problems.

References

1. Owens, J.D., Luekbe, D., Govindaraju, N., Harris, M., Krger, J., Lefohn, A.E., Purcell, T.J.: A Survey of General-Purpose Computation on Graphics Hardware, Computer Graphics Forum, Volume 26, Issue 1, pp. 80–113, 2007
2. Unat, D., Cai, X., Baden, S.B.: Mint: Realizing CUDA Performance in 3D Stencil Methods with Annotated C Proc. Int'l Conf. Supercomputing, pp. 214–224, 2011
3. The OpenACC Application Programming Interface, Version 1.0, 2011
4. OpenMP Application Program Interface, Version 3.1, 2011
5. `http://www.gp-sme.eu/`
6. The OpenACC Application Programming Interface, Version 2.0, March 2013
7. Pouchet, L-N.: PolyBench: The Polyhedral Benchmark suite (2011), Version 3.2, `http://www.cs.ucla.edu/~pouchet/software/polybench/`, 2011
8. Grauer-Gray, S., Xu, L., Searles, R., Ayalasomayajula, S., Cavazos, J.: Auto-tuning a High-Level Language Targeted to GPU Codes, Proc. Innovative Parallel Computing, pp. 1–10, 2012
9. Nugteren, C., van den Braak, G., Corporaal, H.: Future of GPGPU Micro-Architectural Parameters, Design, Automation and Test in Europe (DATE 13), 2013
10. Zhou, J., Unat, D., Choi, D. J., Guest, C. C., Cui, Y.: Hands-on Performance Tuning of 3D Finite Difference Earthquake Simulation on GPU Fermi Chipset, Procedia Computer Science, 9, pp. 976-985, 2012
11. Fang, J., Varbanescu, A.L., Sips, H.: A Comprehensive Performance Comparison of CUDA and OpenCL, Proc. Parallel Processing, pp. 216–225 2011
12. Komatsu, K., Sato, K., Arai, Y., Koyama, K., Takizawa, H., Kobayashi, H.: Evaluating performance and portability of OpenCL programs, Proc. Automatic Performance tuning, 2010
13. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Skadron, K.: A performance study of general-purpose applications on graphics processors using CUDA, Journal of Parallel and Distributed Computing, Vol 68, Issue 10, pp. 1370–1380, 2008
14. Magni, A., Grewe, D., Johnson, N.: Input-Aware Auto-Tuning for Directive-based GPU Programming, Proceedings of the 6th Workshop on General Purpose Processor Using Graphic Processing Units, pp. 66–75, 2013
15. Reyes, R. N., Lopez, I., Fumero, J. J., de Sande, F.: Directive-based Programming for GPUs: A Comparative Study. IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012
16. Wienke, S., Springer, P., Terboven, C., Mey, D.: OpenACC - First Experiences with Real-World Applications. In Euro-Par 2012 Parallel Processing pp. 859–870, 2012
17. Herdman, J. A., Gaudin, W. P., McIntosh-Smith, S., Boulton, M., Beckingsale, D. A., Mallinson, A. C., Jarvis, S. A. : Accelerating Hydrocodes with OpenACC, OpeCL and CUDA. High Performance Computing, Networking, Storage and Analysis (SCC), pp. 465–471, 2012