Compositional Compilation for Sparse, Irregular Data Parallelism

Adam Harries
University of Edinburgh
adam.harries@ed.ac.uk

Michel Steuwer
University of Edinburgh
michel.steuwer@ed.ac.uk

Murray Cole
University of Edinburgh
m.cole@ed.ac.uk

Alan Gray
EPCC, University of Edinburgh
a.gray@epcc.ed.ac.uk

Christophe Dubach
University of Edinburgh
christophe.dubach@ed.ac.uk

Abstract

While contemporary GPU architectures are heavily biased towards the execution of predictably regular data parallelism, many real application domains are based around data structures which are naturally sparse and irregular. In this paper we demonstrate that high level programming and high performance GPU execution for sparse, irregular problems are not mutually exclusive. Our insight is that this can be achieved by capturing sparsity and irregularity friendly implementations within the target space of a pattern-oriented, high-level compilation and transformation system. By working in a language rather than a library, we benefit from the ability to generate implementations by program-specific *composition* of building blocks which capture detailed, low-level implementation choices. Using sparse matrix-vector multiplication as a case study, we show that the resulting system produces implementations for which the performance is competitive with, and sometimes outperforms that obtained with leading ad-hoc approaches. We show that there are correlations between good implementation choices and simple measurable properties of the irregularity present in problem instances. These can be used to design heuristics which navigate the implementation space effectively.

In a case study, we implement a number of versions of sparse matrix-vector multiplication, and achieve promising preliminary performance results. On very regular sparse matrices we are able to achieve up to 1.8x the performance of the state-of-the-art sparse matrix-vector implementation from the clSPARSE libray, and up to 0.7x the performance on very irregular applications.

1. Introduction

Modern high performance parallel architectures, such as GPUs, are well suited to executing applications containing high levels of dense, regular data parallelism. Existing high level programming frameworks and languages, such as Accelerate [3], SkelCL [10] and NOVA [5] aim to make this parallelism easily accessible to non-specialist programmers. By keeping the underlying language semantics close to the capabilities of the hardware, the range of programs expressible is limited to those which are likely to be efficiently executable.

Unfortunately, many important problem domains (*e.g.* graph algorithms) give rise to data sets which are sparse and irregular. To address this, and maintain performance, programmers resort to the use of low level languages (for example, CUDA and OpenCL) and ad-hoc optimizations. The resulting code tends to suffer from a lack of portability, and its production is labour-intensive. While encapsulating the resulting code in libraries benefits application programmers in the short term, the monolithic nature of their implementation makes maintenance and improvement increasingly complex.

In this paper we present promising initial results for a new approach to the challenge of combining ease of programming with the ability to make complex, data-dependent optimization choices. Our insight is that this can be achieved by representing both the high level application and the low level optimizations within a uniform language framework, based around parallel patterns, and traversable through a system of compiler-implemented transformations. In such a system applications can be recompiled to account for new system and data characteristics, and the whole framework can be easily extended by the addition of new patterns and transformations.

Our paper is structured as follows: section 2 presents background material on sparsity and irregularity in matrix computation, and its implications for efficient GPU programming, section 3 describes the compilation and transformation framework within which we have conducted our study, section 4 describes our case study in sparse-matrix dense-vector multiplication, focusing on the optimising transformations we were able to capture. Section 5 explains the experimental setup and presents our experimental results. We complete the paper in sections 6 and 7 with discussion of related work and conclusions.

2. Background

In this section we introduce the phenomena of sparsity and irregularity in matrices, and the key characteristics of GPUs and the OpenCL programming model. We focus on the challenges which sparsity and irregularity pose for efficient GPU programming.

2.1 Sparsity and Irregularity in Matrices

Sparsity refers to the phenomenon by which significant portions of a matrix do not contribute data which is “interesting”, except through its position. Typically this involves values which are zeros in the underlying algebra. Sparsity is conventionally addressed by storing only the interesting entries in some compressed format. This saves space, but can complicate access (and hence add computation time) depending upon the requirements of each algorithm.

Irregularity refers to the extent of structure present in sparse matrices. For example, the tri-diagonal matrices common in many applications of linear algebra are highly regular, and lend themselves to specialized formats and algorithms which exploit this. In contrast, graphs arising from web applications are much less regular [7]. When attacked with data parallel algorithms, this irregularity results in significant variation in granularity of computation associated with the individual operations. For example, the amount of work involved in mapping some operation across the rows of a matrix will depend upon the sparsity and layout of the data in each row. Irregularity therefore generates load-balancing problems, which require good heuristics for the mapping of work to processing elements.

In this paper, we will see that the degree and forms of sparsity and irregularity present have a large impact on the correct choice of implementation strategy for sparse-matrix dense-vector multiplication (SMV), which is itself is a fundamental building block for a large range of applications. In our experiments we quantify sparsity by counting the number of non-zero elements present in each row of the input matrix, and irregularity by measuring the variance in sparsity across the whole matrix.

2.2 GPU programming and the OpenCL model

Both our implementation and the framework we compare against are implemented using OpenCL [12]. OpenCL is a cross platform language designed to allow programmers to program heterogeneous *compute devices* (such as GPUs, CPUs, FPGAs etc) in a data parallel manner.

OpenCL programs are described in terms of compute *kernels*, written in a limited subset of the C programming language, which are executed in parallel on a device. The executing threads (called *work items*) are grouped into *workgroups* which are scheduled by the execution device across a set of *compute units* which are typically lightweight SIMT (single instruction, multiple thread) cores.

For execution, modern GPUs group multiple work items of a single workgroup into *warps*. Work items within a warp run in lockstep with each other, with a single instruction applied across all work items each cycle. If two work items within an individual warp *diverge*, i. e. attempting to execute two different instructions, the work items executing the one instruction will halt, while the other work items execute their instruction, in effect, serialising the execution. This is termed *work-item divergence* and should be avoided for achieving high performance.

OpenCL defines a memory model, which separates memory into three distinct address spaces: *global, local* and *private* memory. On a GPU, global memory is typically implemented as a large, relatively low-bandwidth and high-latency region of memory shared across all work items on the GPU. Local memory is a much faster but usually smaller memory. It is shared between work items in a given workgroup. Every work item has its own private memory and is typically provided using very fast on-chip registers.

In order to effectively access memory on the GPU, programmers should ensure that multiple work items read consecutive memory elements when accessing the global memory. These type of accesses are *coalesced* by the hardware enabling a higher percentage of peak memory bandwidth performance.

Both work-item divergence and the need for coalescing present a significant challenge for the efficient implementation of sparse and irregular algorithms on GPUs. *Control flow irregularity* presents workloads where processing of individual data items requires specialized handling leading to control flow in the kernel which easily triggers work-item divergence. *Data irregularity* presents a corresponding problem for memory accesses, as the required data for each work item may be spread across global memory precluding opportunities for coalesced accesses and thus limiting the utilized memory bandwidth.

We will discuss in section 4 how we apply optimisations to achieve memory coalescing and avoid work-item divergence.

3. High level functional GPU programming

In this section, we present our approach to high level GPU programming. We will describe how we extended this approach in the next section for targeting irregular and sparse problems.

In [11] we introduced a novel code generation approach capable of generating highly optimized OpenCL code. This approach is based on a set of high-level functional parallel primitives that are used to express algorithms. Our compiler, turns these high-level primitives into low-level ones that directly corresponds to the OpenCL concepts.

The parallel patterns exposed in our approach for high level programming are well known in the community and have proven to be a valuable abstraction for parallel programming in numerous *algorithmic skeleton* frameworks [3, 4, 10]. In our work we take a different approach from the usual library implementation of skeletons. We have developed a pattern based compiler using a novel compilation technique: starting from a high-level program, provably correct rewrite rules are applied to transform the high level program into an semantically equivalent low level program from which eventually OpenCL code is generated. This process is described in detail in [11]. Choosing this rewrite-based compilation approach has the advantage over fixed library implementations that different OpenCL code can be generated for the same high level expression in different circumstances. In [11] we show that this allows us to achieve performance portability across different parallel processors. Later in this paper we will see that it is beneficial to choose between different OpenCL kernels depending on the irregularity of the input matrix.

3.1 Dot-product

The code listing in figure 1 shows how the computation of the dot product of two vectors is expressed using our parallel patterns. The computation is specified in the lines 4–5: first, the two input vectors `v1` and `v2` are combined pairwise with the `Zip` pattern; then, the `Map`

```
1  val denseDotProduct = fun(
2    ArrayType(ElemT, N)),
3    ArrayType(ElemT, N),
4    (v1, v2) => Reduce(add, 0, Map(
5        mult, Zip(v1,v2)))
6    )
```

Figure 1: The dense dot product computation expressed with functional parallel patterns

pattern is used to compute the product of every combined element; and finally, the final result is computed by summing up all elements using the `Reduce` pattern.

The code in figure 1 shows the usage of our current implementation as a domain specific language embedded in the functional programming language *Scala*. Besides the actual computation, the programmer also specifies the types of the input vectors including their sizes. In the example, the first two lines specify that both vectors must have the same, but statically unknown length (N). The ElemT type represent the element type of the vector, which is typically *integer* or *float*.

3.2 Dense matrix vector multiplication

A key idea of parallel patterns is to reuse them as fundamental building blocks when expressing larger problems. Figure 2 shows how we reuse the `denseDotProduct` defined in figure 1 to express dense matrix vector multiplication.

```
1  val denseMatrixVector = fun(
2    ArrayType(ArrayType(ElemT, M), N),
3    ArrayType(ElemT, M),
4    (matrix, vector) =>
5      Map(fun(row =>
6              denseDotProduct(row, vector)),
7          matrix) )
```

Figure 2: The high-level functional implementation of dense matrix vector multiplication.

The expression is straightforward; matrix vector multiplication can be seen as the application of the dot product computation for every row combined with the vector. This is directly expressed in line 5–7, where the `denseDotProduct` function is applied to each row of the matrix using the `Map` pattern.

When investigating the types we can see that the matrix is represented as an array of nested arrays. Therefore, mapping over the outer array naturally gives access to each individual row.

3.3 Summary

Our compiler approach makes it is easy to compose and reuse existing programs, like the dense dot product. In addition the programmer does not have to commit to a particular OpenCL implementation but instead let the compiler combine and explore different choices. For instance, the compilation process based on rewrite rules can make different optimisation choices when the dot product is on its own or as part of the matrix vector computation.

4. Sparse Matrix Vector Multiplication

This section presents the implementation of sparse matrix vector multiplication using our high-level functional programming model. This includes examples of optimisations that can be expressed with our framework.

4.1 High-level functional representation

Figure 3 presents the high-level functional expression which implements sparse matrix vector multiplication. As can be seen, it is very similar to the expression corresponding to the dense case seen in figure 2. Using our Map primitive, a sparse dot product is performed between each row of the matrix and the input vector. In the dense scheme the subarrays represent dense rows of the matrix, while in the sparse scheme the subarrays represent sparse rows of the matrix. The sparse rows are encoded as arrays of tuples of indices and values, which is similar to the `ELLPACK` format. For instance, the sparse row [0,0,13,54,0,0,75,0] is encoded as [(2,13),(3,54),(6,75)] .

```
val sparseMatrixVector = fun(
  ArrayType(ArrayType(TupleType(Int, ElemT)),N),
  ArrayType(ElemT, M),
  (matrix, vector) =>
    Map(fun(row =>
            sparseDotProduct(row, vector)),
        matrix) )
```

Figure 3: The high-level functional implementation of sparse matrix vector multiplication.

Figure 4 shows the implementation of the dot product between a sparse vector (the matrix row) and a dense vector. Similarly to the dense version, each element of the sparse vector is multiplied with the value of the corresponding elements in the dense vector. Note that the number of elements in v1 (the sparse vector) might be different from the number of elements in v2 (the dense vector). Instead of using the Zip primitive, we need to lookup, for each element of v1, the corresponding value in v2. The ArrayAccess function performs the indirect array access to the dense vector v2 given the index stored in the sparse vector v1. Finally, after all the matching elements have been multiplied together, a reduction sums up all the value computed similarly to the dense dot product.

```
val sparseDotProduct = fun(
  ArrayType(TupleType(Int, ElemT)),
  ArrayType(ElemT, N),
  (v1, v2) => Reduce(add, 0, Map(
     fun((index, value) =>
       mult(value, ArrayAccess(index, v2))),
     v1))
  )
```

Figure 4: The high level expression implementing sparse vector - dense vector dot product

This correspondence shows that sparse operations can be implemented by reusing the same primitives developed for the dense case. We can, therefore, reuse the same mechanism presented in [11] to explore the implementation space of sparse matrix operations. This allows us to both fundamentally change algorithms through simple rewrite rules, as well as implement complex optimisations at a higher level. Instead of having one monolithic kernel, we have a flexible abstract algorithm which we can tune and optimise for given classes of sparsity and irregularity. The next section illustrates some of the implementation choices we have started to explore in this work which result in promising preliminary results. Our end-goal goal is to fully explore, automatically, all the choices typically encountered by implementers of high performance sparse linear algebra.

4.2 Optimisations

This section now examines how a number of optimisations are implemented and expressed within our framework. It also discusses how some optimisations are best suited to particular classes of matrices.

Array of structures vs structure of arrays The first optimisation we describe is a transformation from an “array of structures” to a “structure of arrays” data representation. Figure 5 shows the corresponding expression after applying this transformation. As can be seen, the main difference from figure 3 is the extra parameters to the Zip on line 8 (indices and values) which decompose the original matrix of rows of tuples into two separate matrices. The first matrix stores all the indices while the second one stores all the corresponding values. Since the dot product function expects one row of tuples, we combine the matrices using

```
1  val sparseMatrixVector = fun(
2    ArrayType(ArrayType(Int),N),
3    ArrayType(ArrayType(ElemT),N),
4    ArrayType(ElemT, M),
5    (indices, values, vector) =>
6      Map(fun(row =>
7              sparseDotProduct(row, vector)),
8          Map(Zip,Zip(indices, values))) )
```

Figure 5: Expression corresponding to the structure of array version.

Map(Zip,Zip(indices, values) on line 8 which is equivalent to the original data representation of the matrix.

The advantage of using a compiler-based approach, as opposed to a library-based approach, is two-fold. First the implementation of the dot product does not need to change since the data input is still in the expected format after the optimisation (the sparse row is an array of tuples). Secondly, the expression on line 8 in figure 5 does not actually produce any code when generating the OpenCL kernel. This is implemented inside the compiler using a *view* mechanisms, similar to the notion of views in a database. The compiler tracks the information on how the data is combined and produces the correct code when the data is actually accessed in the dot product implementation. It generates an access to the first or second array respectively, based on whether the index or the value is being accessed.

This optimisation provides a speedup on the GPU in all cases and brings the core data structure more in line with that used by the state of the art library clSPARSE. This structure of arrays format is in fact similar to the standard "compressed sparse row" format, used by many sparse linear algebra toolkits. This example shows that it is possible to encode more complex data structures by simply composing arrays and tuples and by using a small set of functional primitives.

Fusing maps and reductions in sparse/dense dot product Another common and general optimisation which we apply, is the fusion of map and reduce, expressed using the following fusion rule:

```
Reduce(f, z) o Map(g) => Reduce(f o g)
```

This is easily derivable in our system [11], and presents another simple, yet powerful optimisation, which allows us to generate OpenCL code similar to what an expert programmer might write. Note that this rule is expressed using curried arguments and using the function composition operator $\circ$. This optimisation is particularly useful inside the dot-product for instance where it prevents the need for storing the intermediate result produced by the Map function.

Mitigating irregularity through parallel sparse/dense dot products Our high-level functional programming model easily enables nesting of parallelism. In the kernels we generate, we explore two choices of parallelism:

- performing each dot product in parallel (row level parallelism, or single level parallelism)
- parallelising the map and reduction components of the dot product itself (dot product level parallelism, or two level parallelism).

The exploration of multiple levels of parallelism is motivated by the observation that mapping individual threads to rows (or multiple rows to individual threads) suffered huge performance penalties for highly irregular matrices. With the row level parallel scheme, particularly long rows take significantly longer to process than shorter rows, meaning that some threads take far longer to run than others. When we map these threads to the GPU, we have two choices: either map a small number of threads to each multiprocessor, or map as many as possible, neither option is ideal. In the first case, long running threads tie up the entire multiprocessor, stopping other threads from being processed on it, as well as severely underutilising the parallelism available on each multiprocessor. In the second case, long running threads again tie up the entire multiprocessor, but the lack of parallelism is not mitigated, as faster rows are forced to run as "slowly" as a slow row they are scheduled with.

Our proposed solution to this problem, and a modification easily expressible in our system, is to divide each row of the matrix into even sized chunks, and process each chunk in parallel. This solves much of the irregularity problem, as each chunk presents a consistent and regular workload, allowing the GPU to effectively schedule them, allocating more resources to long rows, and fewer to shorter rows. Figure 6 show how the dot product function can be rewritten in order to implement this choice. The inner

```
1  val sparseDotProduct = fun(
2    ArrayType(TupleType(Int, ElemT), M),
3    ArrayType(ElemT, N),
4    (v1, v2) => Reduce(add, 0,
5      Join ( Map(Reduce(
6        fun((acc, (index, value)) =>
7          add(acc,
8              mult(value, ArrayAccess(index,
9                                      v2)))),
10       Split(chunkSize,
11             v1)
12          )))
13     ))
```

Figure 6: Parallel reduction in the dot-product.

Map(Reduce,...) on line 4 performs several reductions in parallel on different chunks of the sparse vector v1. The outer Reduce on line 5 merges all the partial reduction to produce the final value returned by the dot product function.

Coalescing memory access in parallel sparse/dense dot product The above approach works well in practice, however the default implementation results in uncoalesced memory accesses, as each thread performs a map and reduction over a single chunk. In order to coalesce the memory accesses, we use the ReorderStride primitive to reorder each row with a stride as can be seen on line 11 in figure 7. The vector v1 is reordered based on a stride which

```
1  val sparseDotProduct = fun(
2    ArrayType(TupleType(Int, ElemT), M),
3    ArrayType(ElemT, N),
4    (v1, v2) => Reduce(add, 0,
5      Join ( Map(Reduce(
6        fun((acc, (index, value)) =>
7          add(acc,
8              mult(value, ArrayAccess(index,
9                                      v2)))),
10       Split(chunkSize,
11             ReorderStride(M/v1))
12          )))
13     ))
```

Figure 7: Reordering the elements of a row within a parallel dot product to ensure coalesced memory accesses.

is a function of the chunkSize and the number of rows. This allowseach map over the subsequent split to perform memory accesses in a coalesced manner. This modifies the data access patterns of each thread, such that instead of mapping over and reducing a contiguous set of memory, each thread performs its operations

Matrix name	Width	Height	Nonzeros	Mean	Var	Min	Max	Domain	Subdomain
1138_bus	1138	1138	2596	2.28	2.19	1	17	Graph Theory	Power admittance Network
airfoil1	4253	4253	12289	2.89	0.33	0	6	Finite Element Methods	2D Problem
airfoil1_dual	8034	8034	11813	1.47	0.26	0	2	Finite Element Methods	2D Problem
as-735	7716	7716	13895	1.80	468.01	0	1432	Graph problem	Computer Network
bcsstk18	11948	11948	80519	6.74	20.03	1	31	Finite element Methods	Stiffness matrix
bcsstm25	15439	15439	15439	1.00	0.00	1	1	Finite element Methods	Stiffness matrix
ca-AstroPh	18772	18772	198110	10.55	361.14	0	353	Graph Theory	Collaboration Network
ca-GrQc	5242	5242	14496	2.77	24.26	0	56	Graph Theory	Collaboration Network
ca-HepPh	12008	12008	118521	9.87	764.55	0	438	Graph Theory	Collaboration Network
ca-HepTh	9877	9877	25998	2.63	15.95	0	64	Graph Theory	Collaboration Network
cis-n4c6-b1	21	210	420	2.00	36.17	0	20	Topology	Simplicial complexes from Homology
diag	2559	2559	4092	1.60	0.24	0	2	2D Grid	Tree
DK01R	903	903	11766	13.03	27.92	3	19	Comp. Fluid Dynamics	1D Turbulent Flow
examplemm	5	5	8	1.60	0.30	1	2	An example matrix	
G11	800	800	1600	2.00	0.27	0	4	Torus Matrix	+1/-1 Uniform distribution
G32	2000	2000	4000	2.00	0.12	0	4	Torus Matrix	+1/-1 Uniform Distribution
G48	3000	3000	6000	2.00	0.07	0	4	Torus Matrix	+1/-1 Uniform Distribution
G57	5000	5000	10000	2.00	0.06	0	4	Torus Matrix	+1/-1 Uniform Distribution
G65	8000	8000	16000	2.00	0.05	0	4	Torus Matrix	+1/-1 Uniform Distribution
G1	800	800	19176	23.97	215.54	0	62	Random Matrix	3% Uniform Distribution
G43	1000	1000	9990	9.99	41.76	0	33	Random Matrix	1% Uniform Distribution
G55	5000	5000	12498	2.50	4.62	0	12	Random Matrix	0.05% Uniform
G60	7000	7000	17148	2.45	4.32	0	13	Random Matrix	0.035% Uniform distribution
G14	800	800	4694	5.87	139.36	0	129	Random Matrix	Decreasing Average Degree
G51	1000	1000	5909	5.91	174.02	0	154	Random Matrix	Decreasing Average Degree
G35	2000	2000	11778	5.89	193.69	0	206	Random Matrix	Decreasing Average Degree
G58	5000	5000	29570	5.91	251.87	0	558	Random Matrix	Decreasing Average Degree
G63	7000	7000	41459	5.92	267.24	0	586	Random Matrix	Decreasing Average Degree
geom	7343	7343	11898	1.62	17.01	0	101	Graph Theory	Collaboration Network
lung1	1650	1650	7419	4.50	2.94	2	8	Graph Theory	Transfer Network
lung2	109460	109460	492564	4.50	3.76	2	8	Graph Theory	Transfer Network
mk12-b1	66	1485	2970	2.00	86.06	0	45	Topology	Simplicial complexes from Homology
mk9-b1	36	378	756	2.00	38.10	0	21	Topology	Simplicial complexes from Homology
n2c6-b2	105	455	1365	3.00	30.07	0	13	Topology	Simplicial complexes from Homology
n3c5-b3	120	210	840	4.00	12.06	0	7	Topology	Simplicial complexes from Homology
n3c6-b2	105	455	1365	3.00	30.07	0	13	Topology	Simplicial complexes from Homology
n4c6-b1	21	210	420	2.00	36.17	0	20	Topology	Simplicial complexes from Homology
n4c6-b2	210	1330	3990	3.00	48.04	0	19	Topology	Simplicial complexes from Homology
p2p-Gnutella05	8846	8846	31839	3.60	26.49	0	79	Graph Theory	Peer to Peer Network
Trefethen_20	20	20	89	4.45	1.84	1	6	Numerical Analysis	Diagonal Matrix with Primes
Trefethen_200	200	200	1545	7.73	2.13	1	9	Numerical Analysis	Diagonal Matrix with Primes
Trefethen_2000	2000	2000	21953	10.98	2.01	1	12	Numerical Analysis	Diagonal Matrix with Primes
Trefethen_20000	20000	20000	287233	14.36	2.23	1	16	Numerical Analysis	Diagonal Matrix with Primes
Trefethen_200b	199	199	1536	7.72	2.13	1	9	Numerical Analysis	Diagonal Matrix with Primes
Trefethen_20b	19	19	83	4.37	1.80	1	6	Numerical Analysis	Diagonal Matrix with Primes
Trefethen_300	300	300	2489	8.30	2.16	1	10	Numerical Analysis	Diagonal Matrix with Primes
Trefethen_500	500	500	4489	8.98	1.99	1	10	Numerical Analysis	Diagonal Matrix with Primes
Trefethen_700	700	700	6677	9.54	2.22	1	11	Numerical Analysis	Diagonal Matrix with Primes
wiki-Vote	8297	8297	103689	12.50	889.35	0	457	Graph Theory	Voting Network

Table 1: Matrices used as input in our experimental evaluation.

over a strided set of memory locations. Note that the ReorderStride function does not generate any code in our compiler. Instead, it is implemented using the same *view* mechanisms mentioned earlier. The compiler simply tracks this information and later generates the correct array indexing when the data is accessed.

4.3 Summary

The techniques we present above, although not entirely novel, provide us with a large range of potential optimisations which we may apply to a our sparse matrix vector multiplication benchmark. More importantly, the techniques are independent of each other, and composable, allowing us to apply them in different scenarios. Instead of a small number of hand optimised kernels, we are able to provide the building blocks to adapt a kernel to whatever input it may process, and thus handle a greater range of possible input types than a monolithic set of kernels would allow. We believe that this composable and flexible has the potential to produce high performance code. As we will see in the next section, we have produced some encouraging preliminary results which demonstrate the potential of our approach.

5. Evaluation

This section presents our evaluation which focuses on two main aspects: we first want to confirm that a compositional compiler based approach captures a space of generated kernels which is broad enough to capture the variants necessary to adapt to the irregularity in input matrices. Secondly, we want to investigate the performance of our tuned kernels compared to a highly tuned ad-hoc implementation of sparse matrix operations: *clSparse*.

5.1 Experimental setup and evaluation methodology

OpenCL Kernels We generated 41 OpenCL kernels with different implementation strategies and optimization choices applied.

Datasets We used a set of 50 matrices drawn from the University of Florida sparse matrix collection [7]. Our selection aimed to target a wide range of problem domains, as well as a wide range of irregularity. Table 1 gives a list of the matrices used, along with some characterizing statistics including the domain the matrix is drawn from.

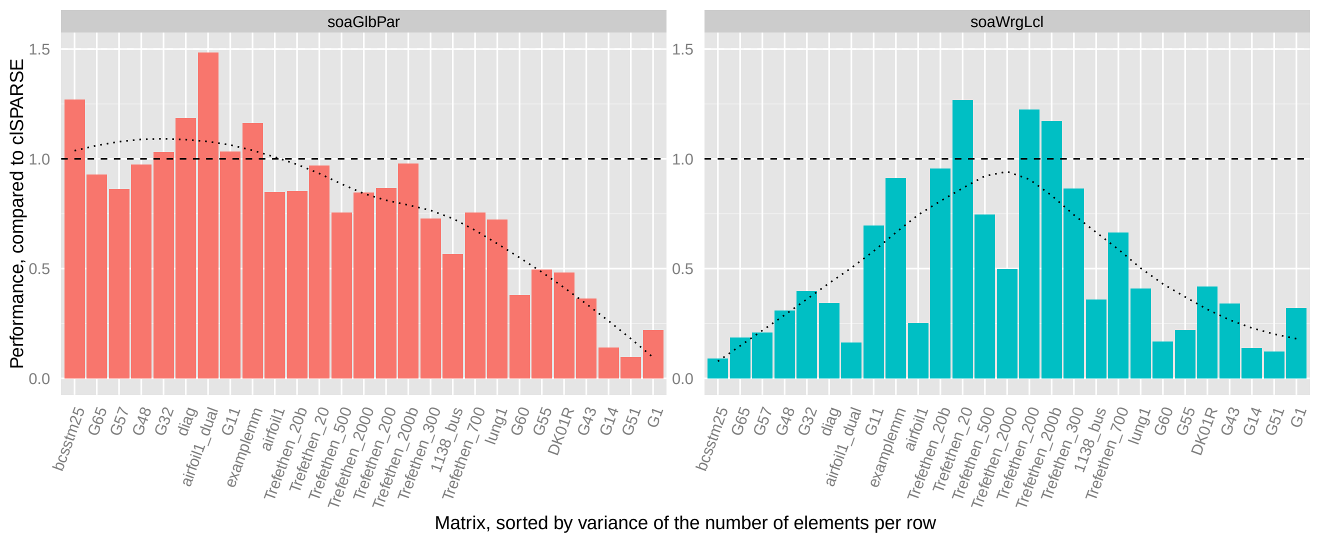

Figure 8: The performance distributions of two kernels, soaGlbPar and soaWrgLcl, showing how each kernel performs best on matrices with different variances. The dotted lines are a smoothed average done with local polynomial regression fitting.

Evaluation platform We measured performance on a single machine, running Scientific Linux 7.1 and equipped with a 32 core Intel Xeon E5-2640 CPU and 64GB main memory. Our machine is equipped with a NVIDIA K40m GPU, with OpenCL version 1.2 and CUDA 7.5.9.

Statistical analysis We refer to a combination of an individual kernel, a single matrix, and a single global and single local size as a *configuration*. For each configuration evaluated, we report the average of at least 30 measurements to minimize the influence of noise.

We focus our measurement on the kernel execution time, excluding any data transfer or host overhead, using the profiling API provided by OpenCL.

Comparing to clSPARSE We compared our results to the open-source sparse linear algebra toolkit clSPARSE framework [1] which is developed by AMD and Vratis Ltd. We selected clSPARSE for comparison for a number of reasons: firstly, as it is open source, we were able to examine the source kernels and execution infrastructure, allowing us to investigate the optimisation techniques applied. Secondly, unlike many other sparse linear algebra toolkits, clSPARSE is written using the OpenCL programming language, allowing for a fair comparison, while a comparison to, e. g. cuSPARSE which is implemented in CUDA, may be influenced by the chosen implementation language.

5.2 Tuning kernels for irregularity

Figure 8 shows how two of our automatically generated kernels have different performance characteristics depending on the irregularity of the input matrices. In this figure, we compare the performance of two generated kernels, `soaGlbPar` and `soaWrgLcl`, across a set of matrices ordered by the variance in elements per row, from low variance on the left to high variance on the right. The kernel `soaGlbPar` performs a sequential dot product over each row and only exploits parallelism across rows, while the kernel `soaWrgLcl` performs a dot product, with a parallel map but a sequential reduction for each row. We can clearly observe from the figure that the first kernel is beneficial for matrices with low variance of the number of elements per row, where the second kernel performs best for a higher, but not too high, level of variance. For example, the kernel `soaGlbPar` clearly performs best on kernels with a very low level of irregularity (e.g. the bcsstm25 matrix), while the kernel `soaWrgLcl` performs best on matrices with a medium degree of irregularity (e.g. the Trefethen 200 matrix).

5.3 Performance comparison against clSPARSE

Figure 9 shows the performance of our best automatically generated kernel selected for each input matrix compared to the performance of the manually tuned clSPARSE. Different selected kernels are shown in different colors.

Overall, the performance of our generated kernels is mixed: in some cases on particular – mostly regular – matrices, we are able to reach a speedup of up to 1.8x compared to clSPARSE. For the majority of the more regular matrices shown in the left half of the plot we are either on par, or faster than clSPARSE. Here mostly the same kernel is selected, but there exists interesting outliers, e.g. for the Trefethen 20000 matrix, where a different kernel shows very good performance.

For the more irregular matrices on the right side the performance is significantly worse. Although we show that different kernels show the highest performance for different input matrices, the best generated kernels for these irregular matrices are still lower performing than clSPARSE.

Figure 9 illustrates this trend very clearly: there is a clear correspondence between the performance of the generated kernels, and the variance of the processed matrices.

5.4 Limitations

Overall, there appears to be good evidence to suggest that our compiler-based and compositional approach is sound and promising. However, as this is work-in-progress there exist a number of limitations with our current implementation which are especially relevant for sparse and irregular problems.

The first limitation is the lack of ragged arrays within our framework. This limits the size of matrices we can process, as we currently allocate the same amount of storage for each row, which is based on the size of largest row. This is especially a problem

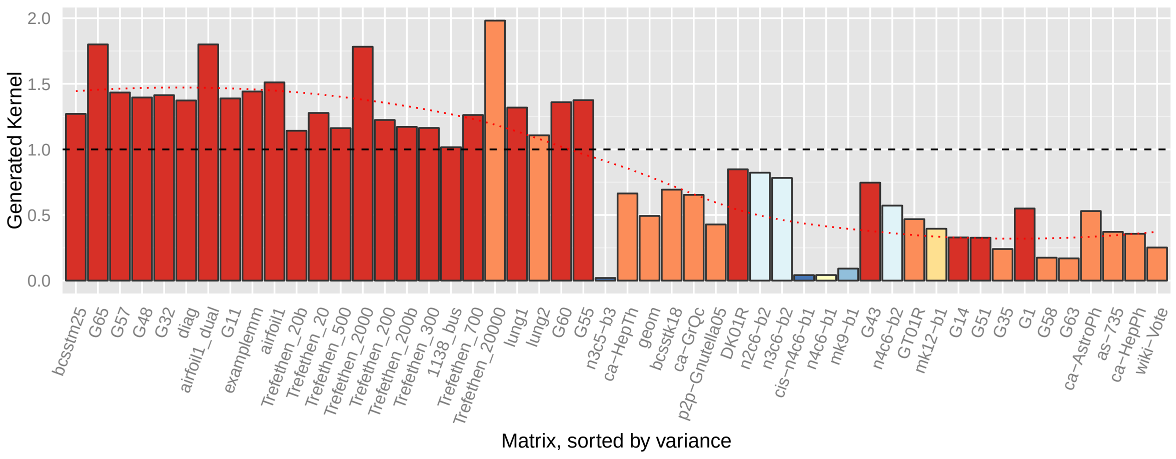

Figure 9: The maximum performance our generated kernels achieve in comparison to clSPARSE. The performance is on par, or exceeds clSPARSE for just under half the matrices evaluated. There is however a clear downward trend in performance as irregularity increases. The dotted lines are a smoothed average done with local polynomial regression fitting.

on highly irregular matrices, as each row is currently allocated the same space as the longest.

The second limitation of our current implementation is that the mapping of parallelism is currently performed statically and there is no support for dynamic load balancing. All parallel patterns assume that each unit of work is roughly equal in difficulty, or time to process. For example, a map pattern assumes that each thread will be presented with a roughly equal workload. This assumption is fairly easily broken by highly irregular workloads, which incurs a significant performance penalty as we have seen. In future work we want to address this by implementing additional primitives capable of mapping the parallelism in a dynamic fashion to achieve load balancing. This would allow our system to choose between different static and dynamic load balancing schemes. For example the parallelism at the row level could be expressed statically while the processing of the row elements could be done dynamically.

5.5 Analysis of Results

We believe that our initial results are already promising, and give us a very solid base for future development as our system evolves.

With respect to our first goal, we have clearly demonstrated our ability to generate kernels whose relative behaviour is sensitive to identifiable variations in properties of the input matrix (in this case irregularity). While the 41 kernels explored were selected by intuition, they were generated automatically. The space of potential kernels is much larger, and automated search of such spaces is an orthogonal topic, already widely explored in the compiler world.

With respect to our second goal, we have demonstrated that our generated kernels can sometimes outperform a state of the art library. Where they fail to do so we have an understanding in terms of irregularity of when this is likely to occur, and of the corresponding optimizations which are missing from our current framework. The strength of the compositional nature of our system is that these can be added subsequently and independently, in contrast to the disruptive impact they would have on ad-hoc kernel implementations.

6. Related work

Sparse linear algebra on GPUs has been studied before by numerous projects. Mostly implementing ad-hoc solutions for particular GPUs or architectures.

GPU libraries for sparse linear algebra Bell and Garland [2] explore the design of efficient sparse matrix vector multiplication kernels using CUDA for a GeForce GTX 285 GPU, achieving good absolute performance and peak bandwidth utilization for that architecture. The experiments show that different sparse data formats and correspondingly tuned algorithms have affinity to different classes of sparsity and irregularity. In contrast to our work, all coding and tuning is done manually.

Daga and Greathouse [6] tackle sparse matrix vector multiplication for the compressed spare row format, implementing adaptive schemes which choose between three algorithmic strategies on a row-by-row basis. The algorithms are optimised for short, long and very long rows respectively and incorporate other tweaks (for example, a logarithmic depth reduction strategy) as deemed appropriate. These are reported to outperform previously state-of-the-art GPU SMV implementations. As with Bell and Garland, the implementation, and now also the online algorithmic choice heuristic, are hand-coded.

Besides clSparse, which we already discussed in the evaluation section, there exist NVIDIAs CUDA Sparse Matrix library (cuSPARSE) [8]. This library is a CUDA specific collection of sparse linear algebra routines, including sparse matrix vector multiplication. It supports a range of dense and sparse formats, including coordinate, compressed sparse row, compressed sparse column, and a hybrid format.

High-level GPU code generation Petabricks [9] is a programming language letting the programmer specify alternative implementations of an algorithm. The compiler and runtime automatically try to choose the most suitable implementation.

Delite [13] is a system that enables the creation of domain-specific languages which are automatically compiled to multi-core CPU and GPU code. Parallelism is expressed using parallel patterns but, unlike our system, neither Delite nor Petabricks offer any support for sparse and irregular problems.

7. Conclusions

The results presented in this paper represent an encouraging validation of our underlying research programme. They demonstrate that a compositional compiler and language based approach to the exploitation of parallel patterns can be, at the very least, competitive with hand-coded libraries, and similarly sensitive to the input characteristics which determine the most appropriate optimizations. This is a strong result, whose benefits will become more pronounced as we expand our range of patterns and optimizations. Essentially, through orthogonality the work required to add each new capability to our system is linear, but through composition its potential impact on the quality of the our results will be multiplicative. This contrasts sharply with the effort required and impact achieved to systematically retrofit new tricks to existing libraries. In the short term, we plan to expand our system to close the performance gaps which remain in certain areas of the SMV case study. In the longer term, we will extend to new patterns and domains, while continuing to benefit from the power of composition.

Acknowledgments

The authors would like to thank Thibaut Lutz and Toomas Remmelg for their assistance with many technical aspects of the code generation framework used in this work. This work was supported by grant EP/L01503X/1 for the University of Edinburgh School of Informatics Centre for Doctoral Training in Pervasive Parallelism (http://pervasiveparallelism.inf.ed.ac.uk/) from the UK Engineering and Physical Sciences Research Council (EPSRC). This work was partially funded by Google and Oracle Labs.

References

[1] AMD. clMathLibraries/clSPARSE, 2015. URL `https://github.com/clMathLibraries/clSPARSE/tree/master`.

[2] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008.

[3] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, DAMP '11, pages 3–14, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0486-3. . URL `http://doi.acm.org/10.1145/1926354.1926358`.

[4] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press & Pitman, 1989.

[5] A. Collins, D. Grewe, V. Grover, S. Lee, and A. Susnea. Nova: A functional language for data parallelism. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'14, pages 8:8–8:13, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2937-8. . URL `http://doi.acm.org/10.1145/2627373.2627375`.

[6] M. Daga and J. L. Greathouse. Structural agnostic SpMV: Adapting CSR-adaptive for irregular matrices. In *International Conference on High Performance Computing, HiPC 2015*. to appear.

[7] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, Dec. 2011. ISSN 0098-3500. . URL `http://doi.acm.org/10.1145/2049662.2049663`.

[8] Nvidia. cuSPARSE, 2015. URL `https://developer.nvidia.com/cusparse`.

[9] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. Amarasinghe. Portable performance on heterogeneous architectures. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 431–444, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1870-9. . URL `http://doi.acm.org/10.1145/2451116.2451162`.

[10] M. Steuwer, P. Kegel, and S. Gorlatch. Skelcl - A portable skeleton library for high-level GPU programming. In *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May 2011 - Workshop Proceedings*, pages 1176–1182. IEEE, 2011. . URL `http://dx.doi.org/10.1109/IPDPS.2011.269`.

[11] M. Steuwer, C. Fensch, S. Lindley, and C. Dubach. Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance opencl code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 205–217, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3669-7. . URL `http://doi.acm.org/10.1145/2784731.2784754`.

[12] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *IEEE Des. Test*, 12 (3):66–73, May 2010. ISSN 0740-7475. . URL `http://dx.doi.org/10.1109/MCSE.2010.69`.

[13] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans. Embedded Comput. Syst.*, 13(4s):134:1–134:25, 2014. . URL `http://doi.acm.org/10.1145/2584665`.