CASK - Open-Source Custom Architectures for Sparse Kernels

Paul Grigoras
Department of Computing
Imperial College London
paul.grigoras09@imperial.ac.uk

Pavel Burovskiy
Department of Computing
Imperial College London
p.burovskiy@imperial.ac.uk

Wayne Luk
Department of Computing
Imperial College London
w.luk@imperial.ac.uk

ABSTRACT

Sparse matrix vector multiplication (SpMV) is an important kernel in many scientific applications. To improve the performance and applicability of FPGA based SpMV, we propose an approach for exploiting properties of the input matrix to generate optimised custom architectures. The architectures generated by our approach are between 3.8 to 48 times faster than the worst case architectures for each matrix, showing the benefits of instance specific design for SpMV.

1. INTRODUCTION

Sparse matrix vector multiplication (SpMV) is an important kernel in many large scale scientific applications where it is a typical component of linear solver algorithms for large sparse systems of equations [1]. For the HPC community to use FPGAs effectively, SpMV kernels need to have good performance. However, the dynamic nature of the data flow in SpMV [2, 3] requires expensive and complex circuitry and it becomes a challenge to achieve effective use of arithmetic and logic resources and on-chip and off-chip memory bandwidth. Furthermore, performance varies greatly based on the matrix instance [4–6].

While many general purpose solutions have been proposed for reconfigurable architectures, they do not provide a viable alternative to CPU or GPU based computing [4, 5, 7, 8]. We believe that an instance specific approach, in which properties of the sparse matrix such as its sparsity pattern and numerical value range are adequately exploited, is the key to generating high performance sparse matrix kernels on FPGAs. This work takes the first step towards instance specific design methods for SpMV kernels. Our contributions are: *1)* a framework for generating customised *iterative double precision floating point sparse matrix vector multiplication* designs based on sparse matrix instances, taking into account the matrix order and sparsity pattern, *2)* an open-source implementation of the proposed framework for the Maxeler Vectis (Virtex 6) and Maia (Stratix V) platforms[1] and *3)* evaluation on a set of matrices from the University of Florida sparse matrix collection [9] to demonstrate the scope for instance specific design on SpMV.

FPGA'16, February 21-23, 2016, Monterey, CA, USA

 ISBN 978-1-4503-3856-1/16/02…$15.00
DOI: http://dx.doi.org/10.1145/2847263.2847338

2. BACKGROUND

SpMV refers to the multiplication of a *sparse* matrix A by a vector x to produce a result vector b. A matrix is considered *sparse* if sufficient entries are zero to allow adequate representation and algorithms to reduce the storage size or execution time of various operations [1]. In this work, sparse matrices are represented using the Compressed Sparse Row (CSR) format [1]. CSR encodes a sparse matrix of dimension n with N_{nnz} nonzero elements using three arrays: the `values` and `col_ind` arrays have one element for each nonzero value representing its value and column index. The `row_ptr` array encodes the start and end of each row as indices in the `values` and `col_ind` arrays. The notation introduced in this paragraph will be used throughout this work.

Early papers on reconfigurable architectures cover optimisations to reduce resource usage [6, 8], but have not studied instance specific optimisations. [7] proposes a method to process multiple CSR rows in parallel by using independent channels. A methodology for improving memory bandwidth utilisation is described in [4]. However, the proposed format (equivalent to a block CSR of width 16) is not parametric so it is not possible to optimise it based on the matrix sparsity pattern. In addition [4] also studies the use of compression to reduce memory traffic. This has also been proposed in other work [10, 11] to improve performance when memory bound matrices are involved and remains an important point for future development in this work. However, for matrices on which SpMV is not memory bound, careful architectural tuning is more effective than compression.

The cost of pre-processing is acceptable for iterative SpMV implementations. Although early implementations introduced too much overhead [12], recent preprocessing, scheduling and partitioning techniques have been employed successfully [2, 7]. Pre-processing has become essential for modern SpMV based applications and it is also used in our approach to enable partitioning and blocking.

To quantify, understand and correctly apply the numerous optimisation opportunities for SpMV, a systematic method is required to explore the various implementation trade-offs on particular sparse matrices. We present such a method in this work and start by providing an overview in the next section.

[1] caskorg.github.io/cask

3. OVERVIEW

To support instance specific design for SpMV problems we introduce **CASK**, an open-source framework for exploring custom architectures for sparse kernels. The key novelty of **CASK** is the capability of adapting parts of the flow to better suit the problem instance which may lead to more efficient architectures.

At the application level, **CASK** is designed for *iterative double precision floating point SpMV*. We assume the input is a CSR encoded sparse matrix, with double precision floating point values and 32 bit index pointers (`col_ind` and `row_ptr`). SpMV kernels are most commonly part of iterative algorithms, such as linear or non-linear solvers where the structure and possibly values of the sparse matrices will not change for the duration of the algorithm. We assume the resulting optimised SpMV architecture is to be used as part of an iterative algorithm. In such applications the pre-processing time (up to a linear bound) is regularly ignored [2, 4, 7, 10].

At the system level, we assume an accelerator model: an FPGA co-processor is used in conjunction with a multi-CPU host system. Therefore data are initially stored in the CPU DRAM where they are generated as part of a wider application (e.g. Finite Element Method [13]). Data are transferred from CPU DRAM to accelerator DRAM via an interconnect such as PCIe or Infiniband. An iterative computation is implemented on-chip: matrix and vector data are transferred once over the slow interconnect, many iterations are performed on-chip and the output is transferred to the CPU. A large on-board memory is assumed (24 – 48 GB) and the bandwidth between on-board DRAM and the FPGA is assumed to be significantly larger than over the interconnect and significantly smaller than from the on-chip storage.

The three main steps of the **CASK** workflow are *analysis*, *generation* and *execution*.

The *analysis* step identifies the best architecture based on the input sparse matrix by using a performance model which is a function of the architectural parameters shown in Section 4. The performance model provides an accurate estimation of the execution time and resource usage of an SpMV operation on a particular architecture for a specific matrix instance. To support customisation during this step, **CASK** exposes important aspects of the optimisation process ranging from data layout strategies (partitioning and blocking) to architectural characteristics (such as datapath replication and cache structure). In addition, the framework can be extended with new architectures and execution models entirely.

The *generation* step compiles the optimised architecture identified previously. It then generates an x86 executable that configures the FPGA with this architecture and performs the SpMV.

The *execution* step uses the resulting implementation to perform the SpMV as shown in Figure 1 and consists of three main phases: *1) pre-processing*, *2) accelerator execution*, *3) post-processing*.

First, *pre-processing* is performed on the CPU. The input matrix is *partitioned* in work items which can be parallelised and operated on independently. Since the CSR format is used, partitioning can be achieved by row slicing: splitting the matrix into disjoint sets of adjacent rows. In accelerator DRAM, partitions are stored at different addresses and a number of independent memory streams are used for

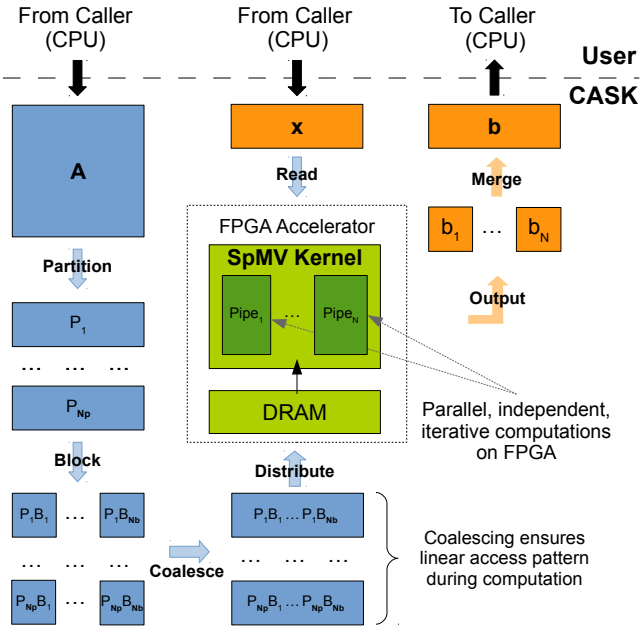

Figure 1: Overview of SpMV steps in CASK

each partition. While this allows better workload distribution, the number of streams is constrained which means that only a small number of partitions can be processed simultaneously. Each partition is then *blocked*. Blocking splits a partition in independent sets of adjacent columns, each set smaller than the size of the on-chip cache, to allow fully storing the required elements of x in the limited on-chip memory. Blocks are *coalesced* to merge the CSR representation of each block into a single input stream. This is required to ensure a linear access pattern of maximal length for accelerator DRAM which is an important factor in achieving good performance. As an optimisation, this process also merges the `values` and `col_ind` arrays, to reduce the number of required memory streams and therefore increasing the maximum number of partitions supported.

Second, during the *accelerator execution* phase, partitions are distributed to accelerator DRAM. The distribution strategy is exposed, so users can experiment with various schemes to improve effective bandwidth, for example by using a better distribution of data across DRAM banks. Then partitions are processed independently in parallel processing units which we refer to as *pipes*; each *pipe* can process multiple matrix nonzero entries from the same row per cycle. This step is described in more detail in Section 4.

Finally, *post-processing* is performed on the CPU to merge the resulting outputs. We note that in many iterative algorithms the order of partitions is irrelevant (due to commutativity of operations) and an iterative algorithm could operate on the current data distribution without additional merging steps.

4. ARCHITECTURE

A flexible, parametric architecture is a prerequisite of the analysis and generation steps. Therefore, the architecture introduced in this section exposes a number of parameters, resulting in more effective customisation. Our architecture is also generic, since it supports any problem up to the size

of the on-board memory of the accelerator. Supporting large problems through blocking has long been a subject for future work [6, 7] since it introduces significant optimisation challenges. For example, the inter-block reduction strategy and efficient handling of empty rows (inevitably introduced by blocking) can have significant impact on performance as shown in Section 5. In [4] a model which effectively constrains the block size to 16 is introduced, but as we find in Section 5, the optimal block size may be as large as 15K on some of the matrices in our benchmark, for our architecture. This is why in our architecture, the blocking strategy is flexible.

An overview of the proposed architecture is shown in Figure 2. It can evaluate k nonzero values per clock cycle of one row and N_p rows in parallel (from different partitions). Intuitively larger values of k improve performance on denser matrices, while larger values of N_p improve performance on sparser matrices. To achieve this design the components shown in Figure 2 are required. Depending on the properties of the input matrix, the generated design may be composed of multiple replicated processing pipelines as described above. As shown in Figure 2, these pipelines are fully replicated, without sharing resources.

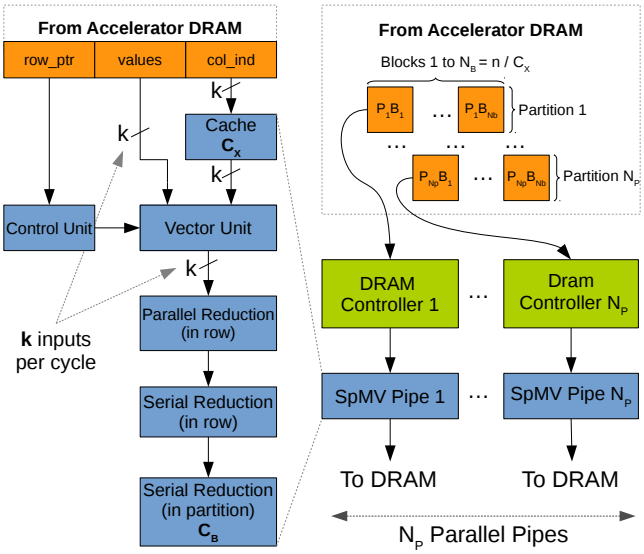

Figure 2: Parameterised SpMV architecture with number of pipes (N_p), reduction cache size (C_b), multiplicand cache size (C_x), vector input width (k)

The control unit implements the sequence of operations in the proposed architecture. Following the steps described in the previous Section, the matrix partitions have been loaded in the accelerator DRAM. From there the execution continues as outlined in Algorithm 1 for each iteration and for each partition. This logic is implemented in a state machine which can decode one CSR entry per clock cycle. Since each parallel pipeline has one such control unit, a total of N_p CSR entries (from different rows) can be decoded per cycle.

The on-chip cache is used to store elements of the multiplicand x. Since the arithmetic unit in each pipe may work on k nonzero entries of a row, the vector element cache should be able to produce k elements per cycle at peak. This is achieved by replicating the vector storage for each arithmetic unit, in total of k times per pipe. During the load stage, all copies can be updated in parallel; during the SpMV arithmetic stage all copies can produce data in parallel.

Algorithm 1 Architecture operation on a single iteration.

```
for all p in partitions do
  for all b in p.blocks do
    for i in 1, size(b) do                ▷ Load vector block on chip
      v[i] ← vectorValue[blockNumber * size(b) + i]
    end for
    part_p ← SpMV(b, v, part_{p-1})
  end for
  output(part_p)
end for
```

To ensure efficient retrieval of elements from the cache (at the rate of exactly k elements per clock cycle) we introduce the Arbitrary Length Burst Proxy (ALBP). The ALBP consists of k FIFOs to store the bursts retrieved from DRAM. When a burst is retrieved (we request a multiple of k elements in each burst from DRAM to improve transfer efficiency), data are pushed in the FIFOs in a circular pattern such that element i of a burst is assigned to the FIFO $o+i \bmod k$, where o is the position after processing the previous burst. $m_t \leq k$ data items may be pulled from the FIFOs, in parallel, by the compute kernel. m_t is determined at runtime, and if fewer than k items are requested, $k - m_t$ zeros are padded to match with the regular k width architecture of the SpMV pipe. Since the FIFOs are implemented in BRAMs, both push and pull can be processed in a single cycle. The ALBP also maintains the count of nonzero elements in a row left to process in order to deliver less than k matrix elements to the arithmetic pipe at the end of matrix row at one cycle, and continues processing k matrix entries of a new matrix row at the next cycle (provided the matrix row has more than k nonzeros to process).

The SpMV arithmetic pipeline processes the current block once the vector elements have been loaded in the on-chip cache. It operates on k elements of the current block as shown in Algorithm 2. Once a block is computed, its results are accumulated with previous results from the current partition. This is done, in parallel, for each partition in the design.

Algorithm 2 Operation of arithmetic pipeline

```
for all r in b.rows do                    ▷ Compute new block
  t_r ← 0
  for all (value, idx) in r.subrows do    ▷ Compute new row
    t_r ← t_r + value × v[idx]
  end for
end for
output(part_{p-1} + t_r)                  ▷ Inter-partition accumulation
```

At the first stage of the arithmetic pipeline a vectorized multiplication is performed on up to k elements of the current row with corresponding vector elements. The outputs are sum-reduced using a balanced adder tree composed of deeply pipelined floating point units, so at every cycle, k values are reduced to a single output. For rows of more than k elements, additional reduction stages are required. Therefore the outputs of the adder tree, one per clock cycle, are reduced using a feedback adder. Due to the latency of double precision floating point addition, a sequence of at least $F_{AddLatency}$ partial sums will be generated in this stage, one per clock cycle. These sums are reduced using a variation

of a partially compacted binary reduction tree (PCBT) [14]. Our modification to the standard PCBT supports reducing an arbitrary number of inputs (up to a design parameter maximum) without stall before processing the next reduction set. It is also capable of skipping *inactive* cycles: data does not shift through the tree levels, thus not modifying its internal data storage, in the case where no input is available. This modification is motivated by the fact that sparse matrix rows may also be shorter than $F_{AddLatency}$. In this case the third stage reduction circuit reduces a number of terms equal to the row length, and the PCBT should produce a correct result for the whole matrix row. This case is even more likely in the presence of blocking, which may reduce the row length per block even further.

Finally, the results from each block must be accumulated with previous results within the same partition. This is done in the *inter-block reduction unit*. For large matrices, this accumulation is performed using DRAM. For small matrices, the number of cycles between writing the result of row i in block b and requiring this value to compute the new value of row i in block $b+1$ may not be high enough to cover the large latency of the write and read to DRAM. Therefore, for small matrices an on-chip buffer (C_b) is employed to perform the accumulation. Of course, it may be preferable to use this on-chip buffer to perform the accumulation for large memory bound matrices to reduce memory traffic.

The proposed architecture also supports empty rows efficiently. Although empty rows are not common in initial sparse matrices since they would correspond to ill-formed systems of equations, they can appear frequently as a result of the blocking strategy required to deal with larger matrices. For example, in banded matrices which are common in practice, a large number of cycles will be wasted on handling empty rows. In fact we observe that for banded matrices, for every C_x columns there will be only at most $2 \times C_x$ rows containing nonzeros on those columns as long as the matrix band is smaller than C_x. Since we should maximise the size of C_x to reduce DRAM transfer overhead due to inter-block accumulation, in practice the matrix band is very likely to be smaller than C_x. Therefore, the rest of $n-2\times C_x$ rows on every group of C_x columns are empty. Assuming one cycle of processing for each empty row (as in the current architecture, with one partition) gives a total quadratic workload overhead of $(n - 2 \times C_x) \times n/C_x = O(n^2)$. We propose a simple approach to reduce the number of clock cycles for processing a sequence of empty rows to 1, provided that the inter-block accumulation results can be buffered on chip, that is $n < C_b$ for some architectures.[2] First, we modify the current CSR format to support a run length encoding of empty rows; we note that without this modification it would not be possible to deal with sequences of empty rows, since the decoding process is serial, requiring at least on cycle per row; every sequence of empty rows will be encoded as an unsigned 32 bit integer for which the most significant bit is the encoding bit[3] and the least significant 31 bits are the length of the empty sequence. Second, we modify the inter-block reduction circuit to allow skipping over empty rows, by providing arbitrary increments to the address counter which controls the write address for the accumulated sum; the address skip is passed from the control unit based on the length of the decoded sequence. In practice the ability to deal with empty rows efficiently can result in substantial speedup.

In summary, the proposed architecture can be used to perform a blocked sparse matrix vector multiplication up to the limit of the on-board DRAM. The architecture is parametric and its most important parameters are k the width of vector pipelines; C_x the multiplicand cache size; C_b the partial result cache size; and N_p the number of parallel pipelines. Tuning the values of these parameters for a specific input matrix can have a strong impact on performance.

5. EVALUATION

The evaluation is performed on a set of matrices from the University of Florida sparse matrix collection [9]. This set is chosen to match the matrices that were used as benchmarks in some of the previous SpMV work on FPGAs [4,5,7]. There is no inherent restriction on the properties of the sparse matrices which are supported, other than they must fit into accelerator DRAM. The benchmark is summarised in Table 1. We note that matrices `mc2depi` and `conf5_4-8x8-05` are not real-valued matrices, so are omitted from our analysis.

We compare our model and implementation on the Maxeler Vectis and Maia platforms with recent implementations targeting the Convey HC-1 platform [4, 5] and a Stratix V development board [7]. The main parameters of these systems are summarised in Table 2. It is difficult to perform an accurate comparison since the DRAM bandwidth, number of FPGAs, type of FPGAs and arithmetic precision differ. To aggregate these results we report a double precision GFLOPs value per FPGA. This is computed for our design as $2 \times N_{nnz}/T$, where the execution time T is measured as explained below. For [7] we optimistically halve the performance of the design, although the resource cost of floating point units increases quadratically with word length and using double precision storage for x reduces the applicability of the design from supporting $16K$ order matrices to $8K$ order matrices. This is considerably smaller than those supported in all other works (including our own, which supports up to main DRAM limit).

Table 2: System properties for previous implementations and this work (Maia and Vectis)

Implementation	[4, 5]	[7]	Maia	Vectis
FPGA	4 x LX330	SV D5	SV D8	V6
Freq. (MHz)	150	150	120	100
Bwidth. (GB/s)	80	21.3	58	38.4
Precision	Double	Single	Double	Double

Figure 3 shows a comparison of the *best* architecture proposed by our framework as shown in Table 1 with implementations of previous work. The execution time on the Maxeler systems is measured using a high resolution clock in the `chrono` library of C++11. It includes the time to load scalar values, to send the compute request from the CPU to the FPGA, to queue initial memory commands and to perform the entire SpMV (including memory transfers on the FPGA). It does not include the time to pre-process and transfer the matrix, the input vector x and the output vector y from CPU to FPGA (and back).

[2] a memory controller extension could be provided to support this optimisation for larger problem sizes but is beyond the scope of this work

[3] when the *encoding bit* is high the number represents an encoded sequence of empty rows

Table 1: Required architectures for each matrix, produced in our approach (for Maxeler Vectis)

Matrix				Architecture				Place & Route	Peak Performance	
Name	Order	Nonzeros	Nnz/row	Cx	k	N_p	Cb	Logic/DSP/BRAM %	GB/s	GFLOPs
dense	2048	4194304	2048.00	2048	16	2	2048	42.63 / 23.41 / 43.14	38.4	6.30
psmigr_2	3140	540022	171.98	4096	16	2	3584	42.02 / 23.41 / 54.23	38.4	4.76
raefsky1	3242	294276	90.77	4096	16	2	3584	42.02 / 23.41 / 54.23	38.4	3.99
rma10	46835	2374001	50.69	7168	16	2	47104	42.91 / 23.41 / 84.87	38.4	1.63
consph	83334	3046907	36.56	9216	8	2	83456	37.46 / 12.30 / 82.61	19.2	1.37
cant	62451	2034917	32.58	11264	8	2	62464	37.15 / 12.30 / 80.92	19.2	1.60
shipsec1	140874	3977139	28.23	14336	16	1	141312	30.24 / 11.71 / 79.65	19.2	0.78
torso2	115967	1033473	8.91	15360	16	1	116224	30.97 / 11.71 / 78.62	19.2	0.19
t2d_q9_A_01	9801	87025	8.88	10240	8	2	10240	36.24 / 12.30 / 60.81	19.2	0.87
epb1	14734	95053	6.45	15360	8	2	14848	37.06 / 12.30 / 75.94	19.2	0.69
mac_econ	206500	1273389	6.17	15360	8	1	206848	27.31 / 6.10 / 73.07	9.6	0.08
scircuit	170998	958936	5.61	14336	16	1	171008	30.39 / 11.71 / 84.54	19.2	0.08
dw8192	8192	41746	5.10	8192	8	3	8192	45.59 / 18.45 / 78.57	28.8	0.68

First, we observe that on the denser matrices (`dense, psmigr_2, consph, cant`) we outperform other implementations. This is due to the ability to select the correct values of k, N_p, C_x and C_b to maximise the achieved bandwidth while allowing sufficient spare resources to place and route the design. We note that when increasing the memory controller frequency to maximise bandwidth, the additional buffering and pipelining logic for the memory streams occupies as many resources as the SpMV kernel itself. So finding the configuration that enables the design to place and route and run correctly is an achievement in itself.

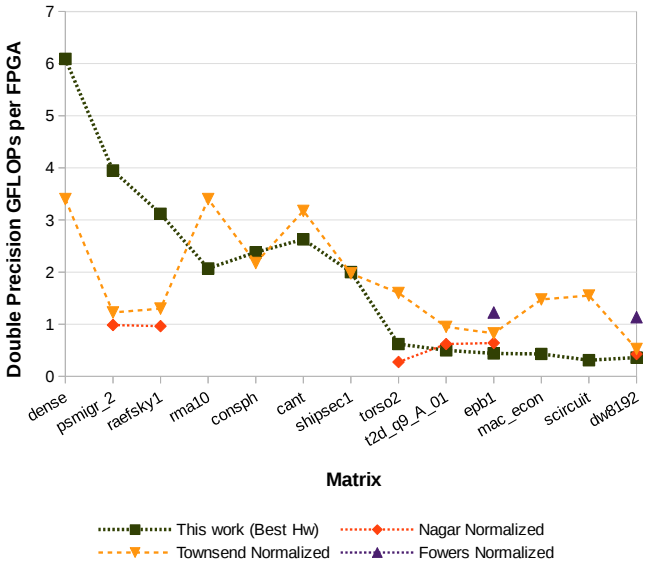

Figure 3: Comparison of the best Maia or Vectis design with prior work for matrices in our benchmark

For sparse matrices (`torso2` to `dw8192`), the platform used in [4] is better suited: with a monolithic memory controller and a large burst size, the Maxeler platforms (both Maia and Vectis) are better suited to linear access patterns with a large number of consecutive bursts; with multiple controllers, and high effective bandwidth for random access, the Convey platform is a better fit to the sparser matrices.

On matrices with few unique values, which therefore compress well (`rma10`, `cant`) our implementation is outperformed by [4] which uses compression techniques to improve effective memory bandwidth while we do not. On `epb1` and `dw8192` the flexible cache structure results in better utilisation of available memory bandwidth, but the approach is not directly applicable to large matrices in our benchmark.

Table 1 shows that on many matrices the proposed architectures are not bound by memory throughput on the Maxeler Vectis (and similarly Maia) platforms. Figure 4 shows projected results for the proposed architecture on the Vectis platform assuming increased memory bandwidth (100GB/s) and increased logic resources and hard blocks such as BRAM. Architectures for denser matrices such as `dense`, `psmigr_2`, `raefsky1` benefit substantially from larger memory bandwidth and are hence memory bound. Architectures for sparser matrices could benefit from more resources, by deploying more independent processing pipes to achieve a better utilisation of on-board DRAM bandwidth.

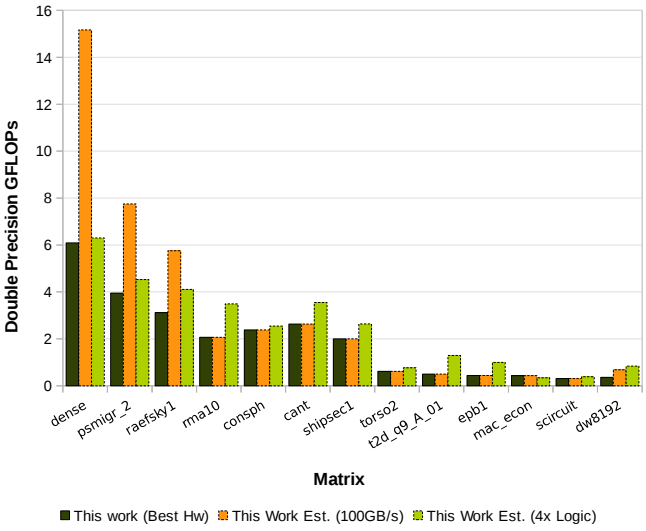

Figure 4: Projections for increased bandwidth or resources

We note that all implementations on FPGA so far are outperformed by CPU and GPU implementations. For example scaling the results of [15] for a modern dual CPU Xeon server would indicate performance on the order of 4.4 – 12 GFLOPs and [4] observes performance in the range of 12

– 30 GFLOPs for an Intel MKL implementation running on a dual Intel Xeon E5-2690 server. Furthermore NVIDIA cuSPARSE [16] claims speedup in the range of 2 - 5x using a modern Tesla K40 GPU over Intel MKL running on one E5-2649. A more conservative estimate based on scaling the results in [17] for the bandwidth of a K40 GPU suggests performance in the range of 9 – 24 GFLOPs could be achieved.

Nevertheless, we believe that through the use of systematic instance specific optimisations, FPGAs would be able to compare favourably with these platforms in the near future. Table 1 also shows the need for an automated, instance specific design method: to maximise performance on the 13 matrices in our benchmark, no fewer than 12 distinct configurations for the Vectis platform have been identified, with various sizes of C_x, k and N_p found to be most effective based on the problem size (the number of rows, which also constrains C_b) and matrix sparsity pattern. A similar set of configurations has been built for the Maia platform, but has been omitted for brevity.

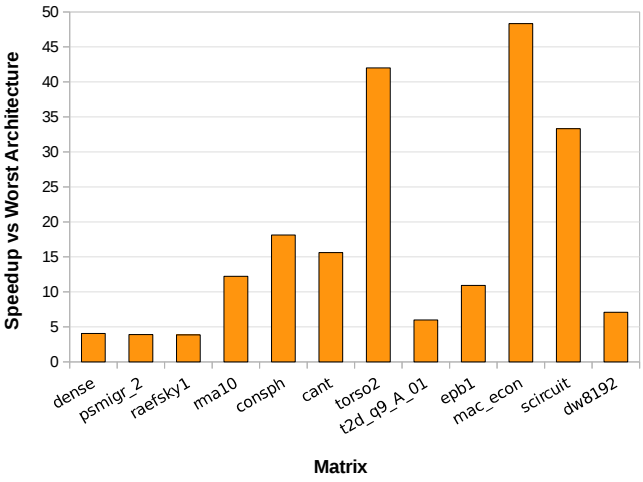

Figure 5: Speedup of best architecture versus worst architecture configuration

Finally, we note that not only do we require a large number of distinct configurations to achieve maximal performance, but also selecting the wrong configuration can reduce performance significantly, by a factor of almost 50 in the worst case, as shown in Figure 5.

6. CONCLUSION

We have introduced **CASK**, an open source tool for exploring custom architectures for sparse kernels. We have shown that **CASK** can use properties of the input sparse matrix such as the sparity pattern to generate customised, instance specific architectures for SpMV. This approach can lead to improved performance and applicability of FPGA based SpMV implementations.

Opportunities for future work include the use of compression techniques [10, 11], instance specific word-length optimisation and support for additional parameters such as clock frequency, memory controller frequency and various low level, FPGA specific optimisations. These improvemets can bring the performance of SpMV kernels on FPGAs closer to that on optimised CPU and GPUs systems, paving the way for wide adoption of FPGAs in HPC.

Acknowledgments

This work is supported in part by the EPSRC under grant agreements `EP/L016796/1`, `EP/L00058X/1` and `EP/I012036/1`, by the Maxeler University Programme, by the HiPEAC NoE, by Altera, and by Xilinx.

7. REFERENCES

[1] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Society for Applied and Industrial Mathematics, 2003.

[2] G. Chow, P. Grigoras, P. Burovskiy, and W. Luk, "An Efficient Sparse Conjugate Gradient Solver Using a Benes Permutation Network," in *Proc. FPL*, 2014.

[3] X. Niu, W. Luk, and Y. Wang, "EURECA: On-Chip Configuration Generation for Effective Dynamic Data Access," in *Proc. FPGA*, 2015.

[4] K. Townsend and J. Zambreno, "Reduce, Reuse, Recycle (R 3): A design methodology for Sparse Matrix Vector Multiplication on reconfigurable platforms," in *Proc. ASAP*, 2013.

[5] K. K. Nagar and J. D. Bakos, "A Sparse Matrix Personality for the Convey HC-1," in *Proc. FCCM*, 2011.

[6] L. Zhuo and V. K. Prasanna, "Sparse Matrix-Vector Multiplication on FPGAs," in *Proc. FPGA*, 2005.

[7] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt, "A High Memory Bandwidth FPGA Accelerator for Sparse Matrix-Vector Multiplication," in *Proc. FPGA*, 2014.

[8] L. Zhuo, G. R. Morris, and V. K. Prasanna, "Designing Scalable FPGA-based Reduction Circuits Using Pipelined Floating-point Cores," in *Proc. ISPDP*, 2005.

[9] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, p. 1, 2011.

[10] S. Kestur, J. D. Davis, and E. S. Chung, "Towards a Universal FPGA Matrix-Vector Multiplication Architecture," in *Proc. FCCM*, 2012.

[11] P. Grigoras, P. Burovskiy, E. Hung, and W. Luk, "Accelerating SpMV on FPGAs by Compressing Nonzero Values," in *Proc. FCCM*, 2015.

[12] M. DeLorimier and A. DeHon, "Floating-point Sparse Matrix-Vector Multiply for FPGAs," in *Proc. FPGA*, 2005.

[13] P. Burovskiy, P. Grigoras, S. J. Sherwin, and W. Luk, "Efficient Assembly for High Order Unstructured FEM Meshes," in *Proc. FPL*, 2015.

[14] L. Zhuo, G. R. Morris, and V. K. Prasanna, "High-performance reduction circuits using deeply pipelined operators on FPGAs," *IEEE Trans. PDS*, vol. 18, no. 10, pp. 1377–1392, 2007.

[15] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of Sparse Matrix–vector Multiplication on Emerging Multicore Platforms," *Parallel Computing*, vol. 35, no. 3, pp. 178–194, 2009.

[16] NVIDIA, "Nvidia cuSPARSE Framework." [Online]. Available: https://developer.nvidia.com/cuSPARSE

[17] N. Bell and M. Garland, "Implementing Sparse Matrix-vector Multiplication on Throughput-oriented Processors," in *Proc. SC*, 2009.