Reduction operator for wide-SIMDs reconsidered

Reduction Operator for Wide-SIMDs Reconsidered

Luc Waeijen†, Dongrui She†, Henk Corporaal† and Yifan He†‡

† Eindhoven University of Technology, Den Dolech 2, The Netherlands
‡ Recore Systems B.V., 7500 AB Enschede, The Netherlands
{l.j.w.waeijen, d.she, h.corporaal, y.he}@tue.nl

ABSTRACT

It has been shown that wide Single Instruction Multiple Data architectures (wide-SIMDs) can achieve high energy efficiency, especially in domains such as image and vision processing. In these and various other application domains, *reduction* is a frequently encountered operation, where multiple input elements need to be combined into a single element by an associative operation, e.g. addition or multiplication. There are many applications that require reduction such as: partial histogram merging, matrix multiplication and min/max-finding. Wide-SIMDs contain a large number of processing elements (PEs) which in general are connected by a minimal form of interconnect for scalability reasons. To efficiently support reduction operations on wide-SIMDs with such a minimal interconnect, we introduce two novel reduction algorithms which do not rely on complex communication networks or any dedicated hardware. The proposed approaches are compared with both dedicated hardware and other software solutions in terms of performance, area, and energy consumption. A practical case study demonstrates that the proposed software approach has much better generality, flexibility and no additional hardware cost. Compared to a dedicated hardware adder tree, the proposed software approach saves 6.8% in area with a performance penalty of only 7.1%.

1. INTRODUCTION

Reduction is a higher order function which combines a given list of input elements through the use of an associative operation, construction a single return value. Examples of reduction are calculating the sum of the elements of a vector, finding the maximum or minimum element in a list and logic operations such as `and`, `or` and `xor` over a vector. Reduction is encountered so frequently that many programming languages such as C++, python, perl and ruby, have built in support for it, although often under different names including `accumulate`, `fold`, `aggregate`, `compress` and `inject`.

Reduction is also often encountered in the video, image and signal processing domains, which are the target domains of wide-SIMDs. Amongst others, reduction is required for kernels such as Partial Histogram Merging, Convolution, Sum of Absolute Differences, Row Projection, Min/Max-finding and Matrix Multiplication.

Because the operator used in reduction is associative, the different combine operations can be performed independently. Thus, reduction inherently possesses a large amount of DLP. This DLP can be exploited by wide-SIMDs. Given that reduction is such an important part of the target domains of wide-SIMDs, it is imperative to support reduction in an efficient manner on wide-SIMDs.

One of the main difficulties of wide-SIMDs is the interconnect between the PEs. PEs need to be able to communicate in order to synchronize or exchange data. Given that there are hundreds of PEs, any form of complex interconnect soon hits a scalability wall. Therefore wide-SIMDs typically only have a very limited form of interconnect, which puts constraints on the amount and type of communication between the PEs. This complicates the exploitation of the DLP present in reduction.

In this paper two novel reduction algorithms optimized for wide-SIMDs with minimal interconnect are proposed. These algorithms do not rely on any additional hardware and require only local communication with short wires, making this approach extremely scalable. Furthermore this software approach is completely flexible in type of combining operation. To demonstrate the effectiveness of the proposed algorithm, we compare an implementation on a wide-SIMD with limited connectivity, with both a straightforward mapping and a solution with dedicated hardware. Furthermore a practical case study shows that for a practical case dedicated hardware is only 7.1% faster, while it consumes 6.8% more chip area.

The remaining parts of this paper are organized as follows; First the experimental setup including the target platform and data layout are discussed in Section 2. Next a straightforward and two novel reduction algorithms are presented in Section 3. The novel reduction algorithms are analysed and compared with the reference approaches in Section 4, including the results of a practical case study. Finally related work and conclusions can be found in Sections 5 and 6 respectively.

2. EXPERIMENTAL SETUP

This section describes the target platform used to benchmark the novel reduction algorithms. Furthermore the data

.

layout on this platform and a dedicated hardware approach are described.

2.1 Target Architecture

The architecture used to benchmark the novel reduction algorithms is a wide-SIMD with limited interconnect. In particular this SIMD has both an array with N_{PE} RISC-like processing elements to exploit data level parallelism, and in parallel to that a Control Processor (CP). The CP handles the program flow and the PE Array runs in lock-step with the CP. A high level overview of the architecture is shown in Figure 1.

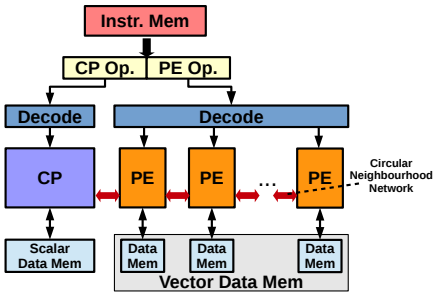


Figure 1: High level overview of the target SIMD

Neighbourhood Network

In order to communicate between PEs and the CP, the architecture has a *neighbourhood network*, which is one of the minimal types of interconnect possible. In this network, all PEs are connected in a circular fashion. To communicate, a PE is able to access one of its neighbouring PEs' operands. The neighbourhood network is illustrated in Figure 2.

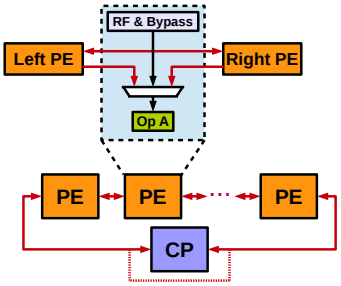


Figure 2: The Neighbourhood Network enables each PE to access it's direct neighbours' operands

The CP can be a part of the loop or not, depending on the configuration of the first and last PE. It is also possible to 'break' the loop and let the boundary PEs read a predefined value. This configuration can be changed at runtime.

All the wires are local and there is no complex network control involved, which is very scalability friendly. This scalability comes at the price of degraded performance for long distance communication. The key concept here is that when a PE needs to exchange data with a PE not directly adjacent to it, that data will have to pass through all PEs in between. Every hop in this chain takes one cycle, hence long distance communication is slow and inefficient. Therefore the challenge of this network is to map algorithms in such a way that communication is kept local as much as possible.

Processing Elements

The Processing Elements are RISC-like architectures with 4 pipeline stages. An instruction can either perform a memory or an arithmetic operation. The memory operations operate on a private data memory (DMEM) with addressing that is independent of the rest of the PE Array. Furthermore each instruction can be predicated to be able differentiate the execution between the PEs.

2.2 Data Layout

The goal of the reduction techniques is to combine the elements of a vector which is distributed over the data memories of the PE Array. In particular we assume N_{Vect} vectors of size V_{size} elements are stored in the N_{PE} data memories of the target SIMD. The N_{Vect} reduced outputs have to end up in the CP. In terms of data layout in the PE Array two cases can be distinguished:

case 1, $V_{size} \leq N_{PE}$:
If the vector size is smaller or equal to the number of PEs, each vector has at most one element in the DMEM of each PE. The vectors are assumed to be stored in rows, and in case $V_{size} < N_{PE}$ the last PEs in the array are assumed to hold no elements and can be left out of consideration. In Figure 3 the position of 4 vectors in the DMEM of the target architecture is illustrated.

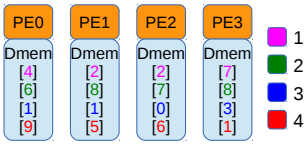
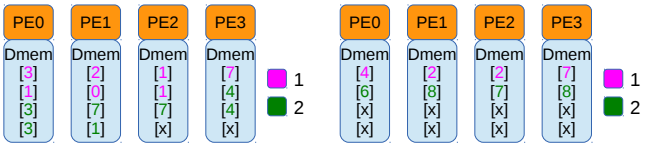


Figure 3: Case 1: $V_{size} \leq N_{PE}$

case 2, $V_{size} > N_{PE}$:
If the vector has more elements than there are PEs, a wrap around is required. Therefore the DMEM of a PE will contain at least one element of the vector and possibly more. It is relatively easy to convert this case to case 1, by letting each PE locally reduce all elements associated to the same vector in its private DMEM. This leads to the same layout as in case 1 where each PE has one element per vector. The conversion from case 2 to case 1 is illustrated in Figure 4.



(a) Initial situation **(b)** After column reduction

Figure 4: Conversion from case 2 to case 1 $V_{size} > N_{PE}$. N.B. here *summation* is arbitrarily chosen as the combine operator for illustration purposes. The choice of operation is completely free.

The conversion from case 2 to case 1 is a simple procedure, since there is no communication required between PEs. Given that a PE contains a maximum of $\lceil \frac{V_{size}}{N_{PE}} \rceil$ elements of a single vector, converting case 2 to case 1 would take $\lceil \frac{V_{size}}{N_{PE}} \rceil$ loads, $\lceil \frac{V_{size}}{N_{PE}} \rceil - 1$ combine operations and 1 store operation. This gives a total of $2 \times \lceil \frac{V_{size}}{N_{PE}} \rceil$.

All the algorithms and techniques discussed hereafter assume a data layout as shown in Figure 3. To compensate for the conversion from a layout such as in Figure 4a, an additional $2 \times \lceil \frac{V_{size}}{N_{PE}} \rceil$ cycles should be added to all running times given in this work.

2.3 Dedicated Reduction Hardware

To benchmark the novel reduction algorithms, they are compared with dedicated reduction hardware. Although dedicated hardware is not as scalable as a software approach, and fixes the supported combine operation at design time, it has been used in the past in wide-SIMDs as will be discussed in Section 5. Therefore it is important to compare the novel algorithms with such an approach.

Since dedicated hardware fixes the type of supported combine operations, a choice has to be made on what to support. Calculating the sum of the elements of a vector is one of the most common types of reduction, and can be found in many kernels. Therefore the focus is on this type of reduction and an adder tree is added to the target architecture as dedicated hardware.

The used adder tree is fully pipelined and can start a new computation every cycle. It is as wide as the PE Array and contains $\lceil log_2 N_{PE} \rceil$ stages. The adder tree inputs and output are memory mapped. The PEs can input elements and the sum of those elements can be accessed by the CP.

3. SOFTWARE APPROACHES

This section contains three software approaches to map reduction to the target architecture. Straightforward reduction is an attempt to exploit the DLP within a single reduction operation, and is intended as a reference for the novel algorithms. The *pipelined reduction* and *diagonal access reduction* are the two novel algorithms that map reduction efficiently to the target architecture using no dedicated hardware extensions or complicated interconnect requirements.

3.1 Straightforward Reduction

In typical cases the DLP in a reduction operation is exploited by performing the operations in a tree-like fashion, i.e. all operations in one layer of a binary reduction tree are executed in parallel. The mapping of such a tree to the PE Array is illustrated in 5. As can be seen in Figure 5, directly

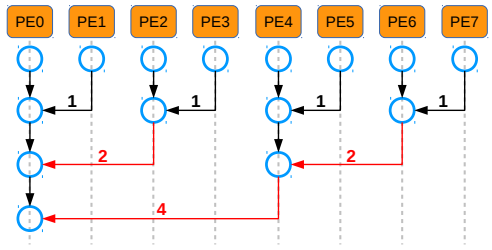


Figure 5: Reduction tree mapped to the PE Array. The numbers indicate the number of required cycles to perform the communication. Red lines require more than one step and severely degrade the performance of the reduction tree.

mapping such a reduction tree onto the target architecture results in a mismatch with the neighbourhood network. Per cycle, data can only be transferred either one PE to the left or to the right. The red arrows in Figure 5 require communication over more than one PE, resulting in additional cycles to perform the communication. Per layer of the tree, the branches become longer and the overhead increases. The number of operations for layer i, consisting of one reduction operation plus communication operations is given in formula 1.

$$OperationsPerLayer(i) = 2^i, \text{with } i = 0, 1, ... \quad (1)$$

The number of layers in a reduction tree for vectors of size V_{size} is given in formula 2.

$$layers(V_{size}) = \lceil log_2 V_{size} \rceil \quad (2)$$

Combining formula 1 and 2, the number of required operations can be calculated, as is shown in inequality 3.

$$\begin{aligned} &Operations(V_{size}) = \\ &= \sum_{i=0}^{layers(V_{size})-1} OperationsPerLayer(i) \\ &= \sum_{i=0}^{\lceil log_2 V_{size} \rceil - 1} 2^i \\ &= 2^{\lceil log_2 V_{size} \rceil} - 1 \\ &\geq V_{size} - 1 \end{aligned} \quad (3)$$

From this inequality it can be concluded that the number of cycles required by the straightforward implemented reduction tree is the same or even more than using a sequential algorithm that simply performs the $V_{size} - 1$ combinations required to reduce one vector.

From inequality 3 it can be concluded that instead of mapping the reduction tree to the SIMD, it would be just as fast, or even faster, to implement a sequential type of algorithm. This is accomplished by shifting the elements to the CP and in parallel combine them one by one as they arrive. The pseudo code for this straightforward method is given in Algorithm 1. In the pseudo code **right**(x) is used to indicate that element x is being read from the right neighbouring PE.

Algorithm 1 Straightforward Approach

```
LoadAddr ← addressFirstVector
for i = 0 to N_Vect do
  v ← load(loadAddr)
  for j = 0 to V_size − 1 do
    v ←right(v)
    CP: combine(right(v))
  end for
  LoadAddr ← LoadAddr + 1
end for
```

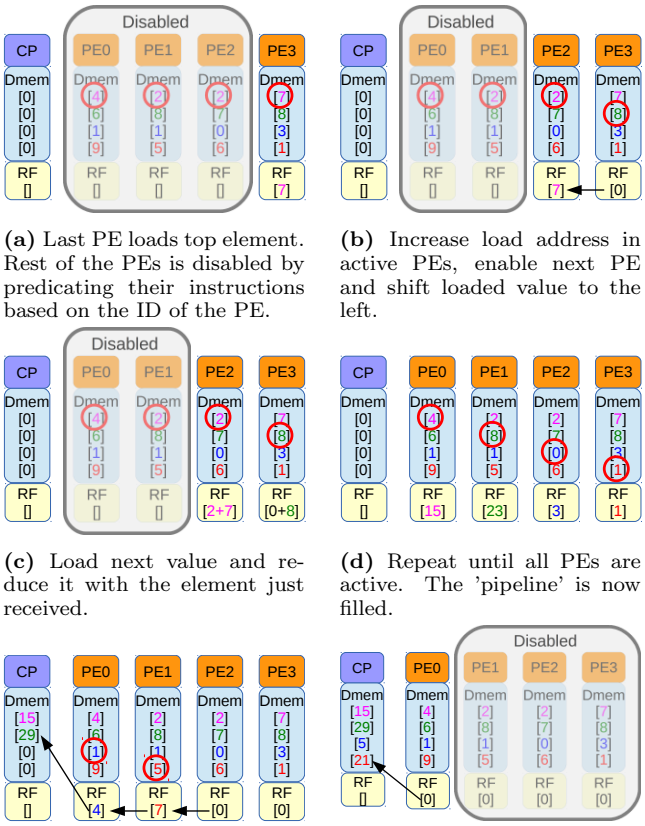
3.2 Pipelined Reduction

Since it is impossible to exploit the DLP within a single vector with a neighbourhood network as shown in the previous section, the parallelism has to be found elsewhere. In this section the novel *pipeline reduction* and *diagonal access reduction* algorithms are introduced that exploit parallelism in the number of vectors that have to be reduced. Using this parallelism the communication pattern is transformed such that only local transactions are required, and the whole PE Array can perform combine operations on the input data.

The pseudo code for the pipelined reduction algorithm is given in Algorithm 2. The key of this algorithm is that it operates on multiple vectors in parallel, i.e. at any given moment in time, all the PEs perform combine operations for different vectors. After a PE has performed a combine operation, the result is passed to the next PE. This PE will then load the element from its DMEM that corresponds to the vector of the received data, and repeat the process. For clarity a visualisation is given in Figure 6. In this pipelined

Algorithm 2 Pipelined Reduction

```
LoadAddr ← addressFirstVector
for i = 0 to N_Vect + V_size − 1 do
  if (PEid ≥ V_size − i) and (loadAddr < endAddr) then
    v ← load(loadAddr)
    s ← combine(left(s), v)
    CP: store(left(s)) {if i > V_size − 1}
    loadAddr ← loadAddr + 1
  end if
end for
```



(a) Last PE loads top element. Rest of the PEs is disabled by predicating their instructions based on the ID of the PE.

(b) Increase load address in active PEs, enable next PE and shift loaded value to the left.

(c) Load next value and reduce it with the element just received.

(d) Repeat until all PEs are active. The 'pipeline' is now filled.

(e) When a PE is done with all vectors in its DMEM, disable it again.

(f) Repeat until all sums have ended up in the CP.

Figure 6: Visualisation of the pipelined reduction Algorithm. For this Figure *summation* is used as the combine operator.

reduction algorithm, three phases can be recognized:

1. Filling the pipeline:
In this phase not all PEs are active. It takes N_{PE} steps before PE0 receives its first element. This phase corresponds with Figure 6a to 6c.

2. Maximum occupancy:
If $N_{Vect} \geq V_{size}$, then there will be a point where all the PEs are active. In this phase V_{size} PEs will perform a useful combine operation per step in the algorithm. See Figure 6d.

3. Emptying the pipeline:
Once the last PE in the array has processed the last vector, it can be disabled. From this point on the remaining PEs will finish one by one until the first PE in the array completes. This corresponds with Figure 6e to 6f.

3.3 Diagonal Access Reduction

If $N_{Vect} < V_{size}$, the pipelined reduction algorithm never enters the most efficient phase (phase 2). Therefore, if N_{Vect} is much smaller than V_{size} it is better to take a different approach. By accessing the elements in a diagonal pattern from the start and using wrap-around, efficient reduction is possible for all situations where $N_{Vect} \leq V_{size}$. The pseudo code for the diagonal access reduction algorithm is given in Algorithm 3. A visualization is given in Figure 7.

Algorithm 3 Diagonal Access Reduction ($N_{Vect} < N_{PE}$)

```
LoadAddr ← addressFirstVector + (PE_ID mod N_Vect)
s ← load(loadAddr)
for i = 0 to N_Vect − 1 do
  loadAddr ← wrap(loadAddr+1) {no modulo required!}
  v ← load(loadAddr)
  s ← combine( v + right(s))
end for
for i = 0 to V_size do
  s ← right(s)
  CP: combine(Result[i mod N_Vect],right(s))
end for
```

4. ANALYSIS AND EVALUATION

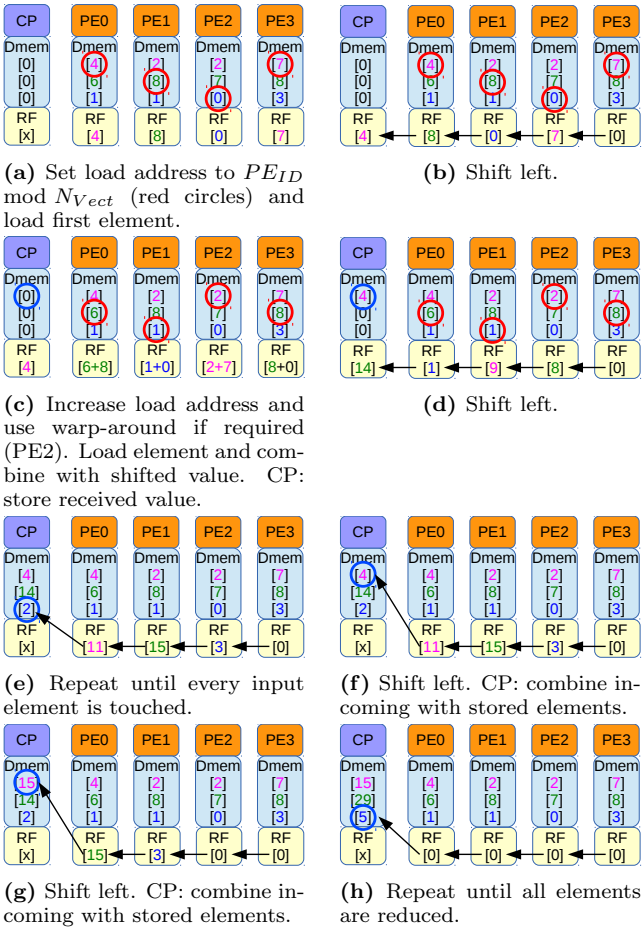
In this section the two novel reduction methods and the reference methods are analysed and evaluated in terms of running time, chip area and energy consumption.

First running times of the various approaches are obtained by using a cycle accurate simulator which is verified against RTL code. The *measured* running times are plotted as *continuous* lines in Figure 8. The vector size (V_{size}) is fixed at 128 elements. Apart from the measured values, the specific constants for the complexity formulas of the algorithms are derived from the source code to approximate the running times for any combination of N_{Vect} and V_{size} (equations: 4, 5, 6 and 7). To demonstrate the accuracy of the formulas, the approximated lines also plotted in Figure 8.

$$\begin{aligned} &\mathit{Straigthforward}(V_{size}, N_{Vect}) = \\ &\quad 10 + 12 \times N_{Vect} + \frac{11}{8} \times V_{size} \times N_{Vect} \end{aligned} \tag{4}$$

$$\mathit{Pipelined}(V_{size}, N_{Vect}) = 26 + V_{size} \times 4 + N_{Vect} \times \frac{19}{8} \tag{5}$$

$$\begin{aligned} &\mathit{DiagonalAccess}(V_{size}, N_{Vect}) = \\ &\quad 12 + 11 \times N_{Vect} + \log_2 \frac{V_{size}}{N_{Vect}} \times (37.5 + N_{Vect}) \\ &\quad + V_{size} \times 0.5 \end{aligned} \tag{6}$$



(a) Set load address to PE_{ID} mod N_{Vect} (red circles) and load first element.

(b) Shift left.

(c) Increase load address and use warp-around if required (PE2). Load element and combine with shifted value. CP: store received value.

(d) Shift left.

(e) Repeat until every input element is touched.

(f) Shift left. CP: combine incoming with stored elements.

(g) Shift left. CP: combine incoming with stored elements.

(h) Repeat until all elements are reduced.

Figure 7: Visualisation of Diagonal Access Reduction, again *summation* is chosen as the combine operation.

$$adderTree(N_{Vect}, N_{PE}) = 14 + \alpha + N_{Vect} \times \left(\frac{6}{\alpha} + 2\right) \quad (7)$$
$$\text{with } \alpha = nextPowerOfTwo(\lceil \log_2 N_{PE} \rceil)$$

Since the adder tree requires exactly the same amount of cycles for $64 < V_{size} \leq 128$, the same line would hold for $V_{size} = 65$. The software approaches would however need to do less work and would thus finish faster. To illustrate this a purple line is added for the pipelined algorithm for $V_{size} = 65$. This line can thus be compared directly to line of the adder tree, indicating how much the performance difference can vary if V_{size} is between two consecutive powers of two.

As can be seen in Figure 8 the *pipelined* and *Diagonal Access* algorithms provide an enormous speed up compared to the straightforward method for more than a couple of vectors. The pipelined approach has a high initial cost and is slow for a small N_{vect}. This effect is partly mitigated by using the Diagonal Access algorithm in this region.

The interesting part however, is that the running time of the adder tree grows at about the same rate as the pipelined reduction algorithm. In fact, as can be derived from the running time formulas, in the current implementation the pipelined reduction algorithm grows at about 2.38 cycles per vector while the adder tree grows at 2.75. At some

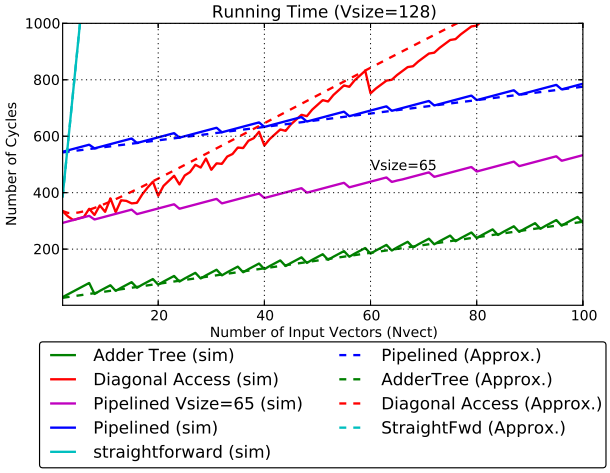


Figure 8: Measured and approximated running times of the various methods for varying number of vectors (N_{Vect}) and fixed vector size (V_{size})

point the software reduction would thus actually be *faster* than the dedicated adder tree.

This effect though is dependent back to the target architecture. Both algorithms have the same complexity and are theoretically able to grow at a rate of one cycle per vector. In the current target architecture however, one cycle is required to load the vector from memory, one to do the actual reduction and the rest is control overhead shared over a number of vectors. An expansion of the target architecture with zero overhead loop support and dual issue PEs would enable a growth of only one cycle per vector for both the dedicated hardware, and the pipelined reduction method.

Table 1 shows the area overhead, and energy results for a fixed input size. These numbers are obtained by synthesizing the SIMD for 400MHz with a 40nm TSMC library. Post synthesis simulation is used to obtain the power and energy results. As can be seen in the table, the adder tree

$N_{vect} = 100$ **and** $V_{size} = N_{PE} = 128$				
Approach	**Area Overh.**	**Speed (cycles)**	**Power (μW)**	**Energy (pJ)**
Straightfwd.	0%	18814	51	2398
Diagonal	0%	1377	63	217
Pipelined	0%	786	56	110
Adder Tree	6.5%	294	82	60

Table 1: Area Overhead, Running Time, Power and Energy comparison for the various approaches obtained by post-synthesis simulation.

consumes less energy, but these numbers are excluding memories. If the data memories are chosen to be 16 bit wide, 1KB large and also built in 40nm technology, the cacti memory tool [6] calculates an access energy of $0.7564pJ$. For the tested configurations this would result in an additional energy of $9758pJ$, making the energy difference between dedicated hardware and the novel algorithms negligible.

Case Study - Fast Focus on Structures

To evaluate the effectiveness of the novel reduction algorithms in a practical application, the Fast Focus on Struc-

Application	Adder Tree	Novel Algorithm
CH/CIAcalculation	2580	2540
Row Projection	379	970

Table 2: Cycles times for both the adder tree and the novel reduction algorithms for FFoS on a 120x45 input image

tures application [2] was mapped to the target platform as a case study.

In the FFoS algorithm the centres of OLEDs have to be detected from an image. In order to do so, reduction is used in two parts of the algorithm. Once to merge partial histograms and convert them to a Cumulative Histogram (CH) and Cumulative Intensive Area, and once to obtain the sum of the rows of the image and detect peaks in that projection.

The cycle counts for the various parts of the application with both the novel software techniques and a dedicated adder tree are given in Table 2. As is shown in the table, for this practical example, the software reduction technique is even faster for the CH/CIA calculation. This is due to the flexibility of the software approach. Where the adder tree always gives its result directly to the CP, the software approach is able to do some post processing in parallel with the CP, reducing the running time. For row-projection, the detection of peaks in the row projection on the CP takes so much time that the reduction operations on the PEs are completely hidden. It is only the initial start up cost that makes the software reduction technique slower here. Overall the FFoS application with dedicated hardware is only 7.1% faster than with the software reduction techniques.

5. RELATED WORK

Reduction is encountered frequently in the target domains of wide-SIMDs and multiple solutions to support reduction have been proposed in the past. The most common approach is to implement dedicated hardware to support a fixed type of reduction. For example S.Seo et al. [5] suggest a dedicated adder tree as an extension to AnySP [7] in order to support the H.264 video codec efficiently. Other examples of SIMDs optimized for video processing that include dedicated hardware include SIMD-2D [3] and the work by Don-Xiao Li et al. [4].

In the SLiM-II [1] a dedicated interconnect is used to support reduction. Essentially the red lines in Figure 5 are implemented as direct, one cycle latency, connections between PEs. To perform one reduction operation with this network $\lceil \log_2 V_{size} \rceil$ communication steps are required. This approach is flexible in type of operation, but a single reduction operation takes $O(\lceil \log_2 V_{size} \rceil)$ operations, as consecutive operations cannot be pipelined. Furthermore implementing the red lines as connections would result in PE0 having $\lceil \log_2 V_{size} \rceil$ additional connections, which the instruction set must support selecting from.

It is clear that efficient reduction support for wide-SIMDs is a relevant topic for many applications. The proposed solutions in the related works all use additional hardware to support reduction causing them to either lose generality, or end up with an inherently slower and more complex design. The novel reduction algorithms introduced in this paper avoid the downsides of dedicated hardware and offer an interesting trade-off between pure performance, flexibility, scalability and chip area.

6. CONCLUSIONS

In this paper we have introduced 2 novel reduction algorithms optimized for highly scalable, low-power interconnects that provide only minimal connectivity. It has been shown that the algorithms are much more effective than a straightforward approach and can even compete with dedicated hardware solutions. The added flexibility of the algorithms can in practical cases give an edge over hardware solutions. Since there is no additional hardware involved and only short local wires for communication are required, these software approaches are cheaper in area and can scale virtually unlimited. As almost all types of interconnect provide the required connectivity, these algorithms could be mapped to existing processors that lack hardware support and for future designs it should be a reason to reconsider adding hardware support at all.

For future work, it would be an interesting topic to see if a slightly more complex network with a few long connections could help to minimize the start up cost of the software approaches. For example a network that allows blocks of 8 PEs to be skipped with a single hop.

7. REFERENCES

[1] H. Chang, S. Ong, C. Lee, M. Sunwoo, and T. Cho. A general purpose SliM-II image processor. In *Computer Architecture for Machine Perception, 1997. CAMP 97. Proceedings. 1997 Fourth IEEE International Workshop on*, pages 253–259, 1997.

[2] Y. He, Z. Ye, D. She, R. Pieters, B. Mesman, and H. Corporaal. 1000 fps visual servoing on the reconfigurable wide SIMD processor. In *Proceedings of the 16th Annual Conference of the Advanced School for Computing and Imgaging 2010*, pages 302–309, 2010.

[3] S. Perri, M. Lanuzza, P. Corsonello, and G. Cocorullo. SIMD 2D convolver for fast FPGA-based image and video processors. In *Military Aerospace Programmalbe Logic Devices, 2003 (MAPLD'2003)*, 2003.

[4] D. -X. Li, W. Zheng, and M. Zhang. Architecture design for H.264/AVC integer motion estimation with minimum memory bandwidth. *Consumer Electronics, IEEE Transactions on*, 53(3):1053–1060, 2007.

[5] S. Seo, M. Woh, S. Mahlke, T. Mudge, S. Vijay, and C. Chakrabarti. Customizing wide-SIMD architectures for H.264. In *Systems, Architectures, Modeling, and Simulation, 2009. SAMOS '09. International Symposium on*, pages 172–179, 2009.

[6] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. Jouppi. cacti 5.3, rev 174, March 2014. Available at `http://quid.hpl.hp.com:9081/cacti/`.

[7] M. Woh, S. Seo, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. AnySP: Anytime Anywhere Anyway Signal Processing. *Micro, IEEE*, 30(1):81 –91, jan.-feb. 2010.