Pragma based parallelization - Trading hardware efficiency for ease of use?

Tobias Kenter, Henning Schmitz, Christian Plessl
Paderborn Center for Parallel Computing
University of Paderborn
Paderborn, Germany
kenter@uni-paderborn.de

***Abstract*—One major obstacle for a wide spread FPGA usage in general-purpose computing is the development tool flow that requires much higher effort than for pure software solutions. Convey Computer promises a solution to this problem for their HC-1 platform, where the FPGAs are configured to run as a vector processor and the software source code can be annotated with pragmas that guide an automated vectorization process. We investigate this approach for a stereo matching algorithm that has abundant parallelism and a number of different computational patterns. We note that for this case study the automated vectorization in its current state doesn't hold its productivity promise. However, we also show that using the Vector Personality can yield a significant speedups compared to CPU implementations in two of three investigated phases of the algorithm. Those speedups don't match custom FPGA implementations, but can come with much reduced development effort.**

***Keywords*-Parallel architectures; Reconfigurable architectures; Performance analysis; Stereo image processing**

I. Introduction

FPGAs can offer huge gains in performance and energy efficiency over general-purpose CPUs, given the right kind of problem. The means to achieve the performance gains can generally be grouped into three categories. Firstly and foremost one can make use of parallelism at various levels, for example by processing many compute elements in a deep pipeline, where at each pipeline stage one element is in process, or computing them simultaneously by replicating identical functional units or pipelines. Secondly, latencies can be reduced by using datatypes specific to the given problems and by generating specific circuits for operations that need to be executed with many individual instructions on a general-purpose CPU. Thirdly, applications that are memory bound on a CPU can be accelerated by higher raw bandwidth to external DRAM or by explicitly designing clever data prefetch or reuse strategies, also making use of the FPGA's huge bandwidth to the internal block RAM resources.

Despite their potential performance gains and energy savings, FPGAs are not a widely utilized platform in many applicable fields of computing. Probably prime among the obstacles hindering their wider adoption is the development effort involved in accelerating applications on FPGAs. The programmer not only has to identify performance critical parts of his code and identify which parts of it can be parallelized in some form. He also needs to rewrite the identified kernels with unfamiliar programming languages like VHDL and Verilog, has to think in different, hardware centric programming models and needs to utilize unfamiliar tool-flows e.g. for synthesis and debugging, that also have way longer runtimes than those of CPU tool-flows.

Another aspect when porting applications to an FPGA platform is that usually there exist large code parts that are best suited for a CPU, either because they are very sequential, control intense or because they are simply not runtime critical, so it is not worth porting them and letting them use valuable FPGA resources. This requires the programmer to also perform a hardware/software partitioning and orchestrate control and data between CPU and FPGA. When considering ongoing development or updates for applications, it is an extra burden to maintain different code versions for CPU and FPGA implementations.

A new generation of tools now promises a way around these problems by offering a pragma based parallelization like with OpenMP, but targeting FPGA [5] or GPU [4] platforms. The first commercially available FPGA platform with this kind of tool-flow is the Convey HC-1 hybrid core computer. After the programmer annotates his C/C++ or Fortran source with pragmas indicating e.g. desired parts for parallelization, data movements or independence of memory accesses, the compiler handles the hardware/software partitioning and generation of accelerator code automatically, which promises a considerably higher development productivity.

The contribution of this paper is an evaluation of this productivity promise, as well as an analysis of the tradeoffs in terms of performance and hardware usage that come with this approach. For this purpose, we compare pragma based parallelizations with OpenMP and the Convey vectorizing compiler, as well as an assembler based Convey implementation and theoretical performance limits. As a case study, we utilize an implementation of a stereo-matching algorithm, which on the one hand offers significant data level parallelism that can be exploited, but on the other hand contains different memory access patterns, dependencies and some control flow among its kernels.

The remainder of this paper is structured as follows: in

978-1-4673-2921-7/12/$31.00

Section II we give a brief overview of the related work. In Section III we present more aspects and details of the Convey hardware platform, its configurations and tool-flow. Next, in Section IV we present the stereo matching algorithm utilized as case study. In Section V, we discuss for the three main phases of the algorithm, their parallelization using the Convey vectorizing compiler, optimizations of the kernels in assembler for the vector instruction set and aspects of parallelization with OpenMP. Here we also cover the achieved performance, before we draw our conclusions in Section VI.

II. Related work

The fundamentals of the Convey HC-1 system architecture and its capability to implement instruction set extensions and custom personalities have been described in the works of Brewer [5] and Bakos [3]. The work by Augustin et al. [2] studies the suitability of the Convey HC-1 for kernels from linear algebra and compares the performance to CPUs and GPUs. Their work also uses the Vector Personality and compiler infrastructure, as well as the work by Meyer et al. [9]. They port a stencil computation application to the Vector Personality and compare both results and development flow with an OpenMP parallelization. Distinctive features of our work are the comparison to theoretical peak performance and the more diverse kernels of our application. Also in our case study, we discover serious gaps between what can be achieved using the Convey vectorizing compilers and what is possible by making best use of the vector instruction set. Maleki et al. [7] have investigated those effects for compilers targeting vector extensions for current general purpose CPUs.

III. Convey HC-1 hardware and tool-flow

In this section, we introduce the Convey HC-1 hardware platform, give a brief overview of the different ways to configure its FPGAs, specifically with the Vector Personality and introduce the basic tool flow to target the Vector Personality.

A. Hardware Platform

A schematic overview of the Convey HC-1 architecture [5] is presented in Figure 1. At its heart, the Convey HC-1 is a dual socket server system, where one socket is populated with a Intel Xeon CPU while the other socket is connected to a stacked coprocessor board. The two boards communicate using the Intel Front-Side Bus (FSB) protocol. Both processing units have their own dedicated physical memory, which can be transparently accessed by the other unit through a common cache-coherent virtual address space. The coprocessor consists of multiple individually programmable FPGAs. One FPGA, the *Application Engine Hub (AEH)* implements the infrastructure that is shared by all coprocessor configurations. These functions include the physical

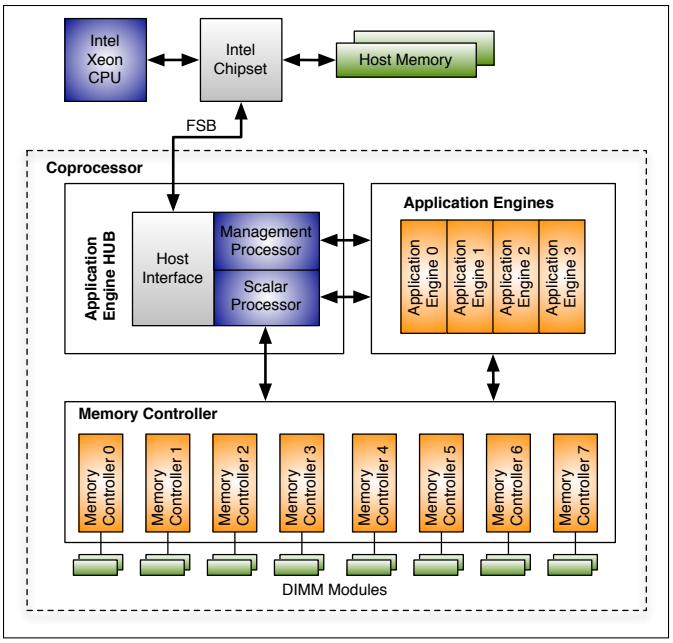


Figure 1. Coprocessor Architecture

FSB interface and cache coherency protocol, configuration and execution management for user programmable FPGAs and command dispatch logic that relays commands received from the host processor to the application-specific logic on the coprocessor. For implementing the application-specific functionality, four high-density Xilinx Virtex-5LX330 FPGAs, denoted as *Application Engines (AE)*, are available.

A distinctive feature of the HC-1 architecture is the availability of a fast multi channel memory interface which provides the application engines with access to 8 independent memory banks through 8 dedicated memory controllers with an aggregated bandwidth of 80 GB/s. Each memory controller accesses two DIMM memory modules. In total 16 DIMMs can be installed resulting in a maximum of 128 GB coprocessor memory. Besides standard memory modules (Std RAM) Convey also offers custom-made scatter-gather modules (SG RAM) which allow accessing memory efficiently in 8-byte data blocks while standard modules are optimized for 64-byte blocks.

The application engines can be configured with so called *Personalities*, that need to implement interfaces to the AEH and the memory controllers and contain the user logic for any specific task. Users can create their own specialized personalities or use those provided by Convey, which exist for a number of specific tasks like graph traversal or local alignment, and as the general purpose *Vector Personality*.

B. Vector Personality

The Vector Personality provides the functionality of a vector processor that executes programs implementing its vector instruction set, in one variant optimized for single

precision floating point and in one supporting also double precision floating point operations. Both also support integer operations. The vector instructions are implemented for up to 1024 elements. A total of 64 vector registers are available and can each store such a set of 1024 elements. Compared to a general-purpose CPU, the total size of this register file is huge and given the right problem and implementation can make up for the non-existing cache in this architecture. Besides the usual element-wise arithmetic vector operations, the vector instruction set contains memory instructions that distinguish it from typical vector instruction set extensions for CPUs. It can load and store vectors where the elements are individually indexed and do not need to be aligned in a continuous memory location. It also supports the generation of vector masks to handle diverging control flow, however currently those masks can only be applied to memory instructions.

C. Vectorizing Compiler

For programming the Vector Personality, Convey offers a vectorizing compiler for the source languages C, C++ and Fortran. There are two versions of the vectorizer component of the compiler, which we both tested. The two vectorizers differ a bit in what source code constructs actually work for automated vectorization and how the generated assembly code looks. Most of the presented results are obtained using the older internal vectorizer.

The programmer can specify via pragmas, which functions are to be executed on the Vector Personality and which loops are to be vectorized. This workflow is supposed to be very similar to OpenMP programming, solving productivity issues by having only a single source code to maintain, which can still be compiled also without the Convey tools. The vectorizing compiler tries to generate code according to the pragmas, using the x86 instruction set for the host CPU and the vector instruction set for the FPGAs, and it orchestrates all interactions between both parts. This is facilitated by the common memory space, which allows the CPU to access the FPGAs memory location and vice versa. However for maximal performance data has to be transferred to the right location, which can be guided by pragmas. A simple example for this use of pragmas is given with Listing 1 for a *saxpy* kernel that adds two vectors, after one vector is multiplied by a scalar factor. The utilized pragmas indicate the desired memory transfers, start and end of the code region that is supposed to run on the coprocessor. The pragma indicating that there are no loop dependencies of the y vector in the inner loop is not required here for vectorization itself, but it prevents a fence operation at the end of each iteration, which could also be achieved with other pragmas.

```
#pragma cny migrate_coproc(x, xByteSize);
#pragma cny migrate_coproc(y, yByteSize);
#pragma cny begin_coproc
    unsigned long i;
#pragma cny no_loop_dep(y)
    for (i = 0; i < size; i++) {
        y[i] = a * x[i] + y[i];
    }
#pragma cny end_coproc
```

Listing 1. Saxpy Kernel with Convey compiler directives

Platform	Virtex-5LX330	Vector Personality
Units	4 FPGAs	4 FPGAs
Cores/Unit	192 DSP48E	8 Function Pipes
SP Throughput/Cycle	1/2 mult[1]	4 fmac
Clock Frequency	550 MHz	300 MHz
Theoretical Peak	211.2 GFLOPS[1]	38.4 GFLOPS
Measured[2] Std RAM	-	5.42 GFLOPS
Measured[2] SG RAM	-	7.36 GFLOPS
Theoretical BW	80 GB/s	80 GB/s
Measured[2] BW Std RAM	-	32.5 GB/s
Measured[2] BW SG RAM	-	44.2 GB/s

Table I
OVERVIEW OF THEORETICAL AND MEASURED PERFORMANCE

D. Achievable performance

We analyze the theoretical peak performance of the Vector Personality and compare it to the raw performance of the FPGAs DSP slices in Table I. According to the Xilinx DSP user guide [11], two DSP48E slices can be combined to form an unsigned 24 x 24 bit multiplier, which has a latency of 4 cycles, but can be pipelined. This covers the multiplication of the mantissas of two single precision floating point numbers, however handling of the exponents and specific characteristics of the IEEE floating point standard must be implemented in fabric or using additional DSP slices. Thus the theoretical peak performance of the FPGA platform in Table I is not full-fledged floating point performance, but covers only the core of the floating point multiplications. On the other hand, it disregards any function units that can be formed from the FPGAs logic cells.

Using these FPGA resources, the Convey Vector Personality implements full floating point units. The functional units are grouped into 8 function pipes per FPGA, where each function pipe can execute up to four fused multiply add operations per cycle. An additional *misc* functional unit can provide two additions per cycle, which are not counted in the table. With a micro benchmark whose inner kernel loop was manually designed in assembler code, we were able to measure 35.98 GFLOPS for independent multiplications, which is close to the predicted peak performance. Consider-

[1]No full floating point multiplications
[2]Automatically vectorized kernel from Listing 1

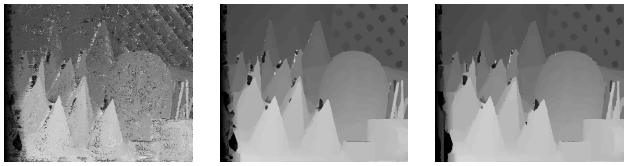
ing the lower clock frequency, which is hard to avoid when combining DSP slices with logic slices, and considering the increase in functionality, from core features of a single multiply to fused multiply add floating point operations, the fraction of $2/11$ of the FPGAs theoretical peak performance that the Vector Personality offers is significant.

However, with actual kernels with memory accesses, written in C/C++ and compiled with the vectorizing compiler, it is hard to reach that performance level. For example for the kernel shown in Listing 1, we measured a performance of 5.42 GFLOPS and a utilized bandwidth of 32.5 GB/s with standard RAM modules and $1/3$ higher values on a machine with scatter-gather RAM modules (Table I). Even though the memory accesses are sequential in this example, the performance is improved by lower memory access latencies of the scatter-gather modules. We see that bandwidth and latencies of the memory accesses significantly limit the performance, when the ratio of computation to memory accesses is rather low as in this kernel.

IV. Stereo Matching

Stereo matching is the task of computing depth information from a pair of two images taken by two cameras with a horizontal offset, mimicking human 3D vision with two eyes. The key concept is, that distant objects have almost parallel incoming light rays for the left and right image and thus appear in both images at the same position, when the cameras are aligned properly. However, the rays of close objects form an angle between the two cameras and appear more to the left in the right image and vice versa. The distance between the position of an object in the left image and its corresponding position in the right image is called disparity. It can be used to compute the distance of an object to the viewer or camera.

We found the stereo matching problem to be an interesting target for FPGA acceleration, because it contains abundant parallelism for example on the pixel level, but on the other hand it is not trivially parallizable because it can be split into different phases with different computational patterns. In our work we follow the algorithm of Mei et al. [8], which is up to now the stereo matching algorithm with the best matching quality in the popular Middlebury Stereo Benchmark [10], [1]. Our current implementation doesn't achieve the matching quality reported by Mei et al., partly because of some unclear details in the first or second phase of the algorithm, partly because we didn't implement all final refinement steps. However, with an average of 6% bad pixels, it would still be in the top third of all benchmark entries. Also, the software runtimes of our implementation match the reported software runtimes well. This is possible because all refinement steps only take up a small fraction of the total runtime due to the fact that they are only applied once to relatively few selected pixels.



(a) (b) (c)

Figure 2. Disparity images after (a) cost initialization, (b) cost aggregation and (c) scanline optimization

In the reminder of this section, we present the general ideas of the three major phases of the implemented stereo matching algorithm and outline their computational patterns. Figure 2 illustrates the disparity images of the *Cones* image pair from the Middlebury benchmark [1], generated after those three phases. The final refinement phase is omitted due to space limitations, for its methods as well as for more details of the three covered phases, we refer to the original paper.

A. Cost Initialization

The cost initialization investigates for each pixel in one image to all possibly corresponding pixels in the other image if they could match each other. The more unlikely the pixels match at a given disparity, the higher is the assigned matching cost. In the benchmark, for each image pair a maximal possible disparity d is given and limits the number of possible matching positions that are investigated. Mei et al. combine two metrics for the cost initialization, the absolute difference cost C_{AD} and a census cost C_{census}. The absolute difference cost is defined as the average intensity difference of all three color channels of the two compared pixels. For the census cost, first the census transform is computed for both images. This transform compares each pixel's grey value with the grey values of its neighbors in a 9 x 7 window and sets a result bit to 1 or 0 if the intensity is higher or not higher respectively. The results are encoded in a 64 bit string for each pixel. Then, the census cost C_{census} for a pair of pixels in the left and right image is given by the Hamming distance of their census bit strings, which can be computed by a bitwise xor operation and a popcount (counting the 1s) of the resulting bit string. Figure 3 illustrates the census cost for the central pixel of a 3 x 3 window.

C_{AD} and C_{census} are scaled by an exponential function that also enables the weighting of outliers and then added to form the final cost C. After each pixel obtained a cost value for each possible disparity value, a first disparity image can be computed, by simply selecting for each pixel the disparity value with the lowest cost.

B. Cost Aggregation

Using only the initial costs, the cost volume or a resulting disparity image is very noisy because it tries to match every

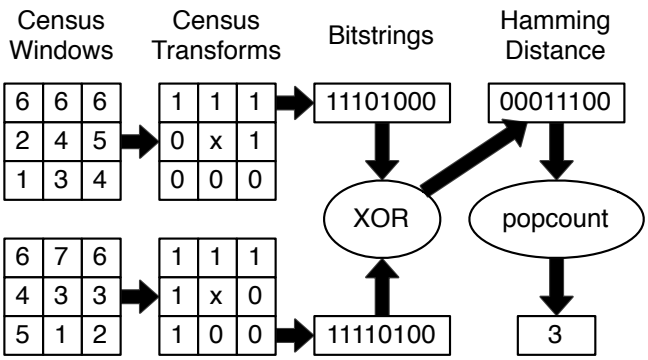


Figure 3. Census transform and Hamming distance for 3 x 3 windows

pixel individually. However, whenever neighboring pixels belong to the same object, they are likely to have the same or a very similar disparity. The cost aggregation tries to exploit this property by summing up costs of neighboring and similar pixels.

To define those neighboring and similar pixels, for each pixel p four so called arm lengths are computed, one for each of four directions. An arm specifies how far the region of similar pixels stretches in its direction. The arm lengths are limited by their absolute length, the color difference between two adjacent pixels on the arm and the color difference of the arm's end pixels. Now the so called support region for p is formed by using the vertical arms of p as base axis and including all horizontal arms of all pixels q on this base axis.

In the aggregation step, for each disparity value, the matching costs of all pixels in the support region of pixel p are summed up to form a new cost value of p. A naïve implementation would require many additions for each support region, but they can be computed much more efficiently with 1D cost volumes, which effectively limit the computational effort to two additions and two subtractions per support region and per disparity value, at the cost of having non-linear memory access patterns.

Mei et al. repeat this aggregation step four times, however for steps 2 and 4, they form the support region in different orientation, using the vertical arms of all pixels q on a horizontal base axis of pixel p, which changes the shape of the support region.

C. Scanline Optimization

The scanline optimization is a technique introduced by Hirschmüller [6] in order to smoothen the matching images. It's basic idea is, that any changes of the assigned disparity between two neighboring pixels should only happen, when they are backed by a difference in matching costs that is above some small penalty P_1. Arbitrary large disparity changes need to overcome a larger cost penalty P_2. The penalties also depend on intensity changes in the original image at that position, since disparity discontinuities are only likely to arise at object boundaries, which often also cause large changes in the original image's intensity values.

These ideas are implemented by a method that iteratively runs over scanlines in different directions. Mei et al. use two horizontal and two vertical scanlines that cover each pixel in all four directions. For every pixel p along a scanline direction r and for every possible disparity d, a scanline cost C_r is computed that depends on the scanline cost values of the previous pixel $p - r$ along the line.

$$\begin{aligned} C_r(p,d) =& C_1(p,d) + \min[C_r(p-r,d), \\ & C_r(p-r,d \pm 1) + P_1, \\ & \min_k C_r(p-r,k) + P_2] - \min_k C_r(p-r,k) \end{aligned} \quad (1)$$

These recursive values are computed with a dynamic programming pattern along each scanline.

V. Parallelization and Results

In this section, we analyze which aspects of the three matching phases can be parallelized. We focus on vectorization potential for the Convey Vector Personality, but also comment on parallelization for multicore CPUs with OpenMP and vectorization for CPU SIMD units. We document in how far we were able to achieve vectorization using the Convey vectorizing compiler and what is possible with the ISA of the Convey Vector Personality. We also report on the achieved runtimes (Figure 4) of each phase for the image pair *Cones*. The first three tests of each phase were conducted on a Convey HC-1 with standard RAM modules, as fourth test, we repeated the third tests of optimized assembler programs on a HC-1 equipped with scatter-gather RAM modules.

Considering other image pairs, the CPU performance is negatively impacted by larger image sizes, because they reduce the effectiveness of its caches, whereas the FPGAs offer the same memory bandwidth for any data size that fits its DRAM memory and can profit from larger images if they help to fill the vector units. The *Cones* image pair is one of the largest in the benchmark, however with only 450 x 375 pixels and 60 allowed disparity values, in real world applications much larger data sizes and thus better speedup results for some stereo matching use cases are to be expected.

For the scanline optimization phase, which turned out to be best suited for the Vector Personality, we also compare the achieved results to the measured peak performance of the Vector Personality. We again subdivide the remainder of this section into the three major matching phases.

A. Cost Initialization

The cost initialization is completely independent over both image dimensions and over all disparity values. Thus it can be easily parallelized with OpenMP over the outer dimensions of the loops and data structures. The Vector

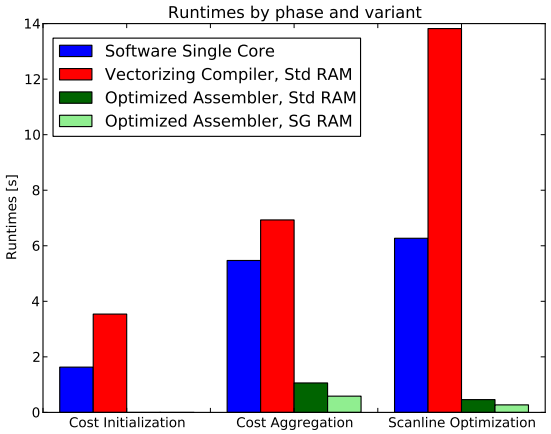


Figure 4. Runtimes of three matching phases for image pair Cones

Personality is not very suitable for the generation of the census transform and computation of Hamming distances C_{census}. The C++ template class *bitset*, which can be used to generate efficient CPU implementations, hinders any automatic vectorization.

When coding out the required steps on basic data types (uint64), the census transform requires a conditional statement to determine if a certain bit is to be set or not. The compiler is not able to vectorize this statement. Similarly, when no hardware instruction for popcount can be used, counting the set bits of a bit string requires a conditional statement, which prevents automatic vectorization.

On the other hand, the computation of C_{AD} as well as the exponential scaling of both cost metrics can automatically be offloaded to the Vector Personality after some code transformations. It required splitting of the three color channels of the image into separate arrays, pragmas specifying the array dimensions and a pragma to indicate no loop dependencies.

The performance results after partly vectorization of this phase are disappointing (left part of figure 4), we see a slowdown compared to the initial single core CPU implementation. Four aspects contribute to this final slowdown: the clock frequency difference between CPU and FPGAs, a less efficient software implementation in order to enable vectorization, the data transfers between host memory and FPGA memory, and additional memory accesses, because we store the computed hamming distances on the host in order to do the scaling on the FPGAs.

It would be possible to implement the entire phase on the Vector Personality by using the vector mask registers to hold the census bit strings. They even support a popcount instruction to count the bits in order to determine the hamming distance. Making use of those features of the vector instruction set, we could again reduce the memory accesses and data transfers. However, we still can't expect such an assembler implementation to be nearly as efficient as a custom hardware solution for those bit level algorithms as presented in [12]. Also, it will be nearly impossible to let the compiler automatically generate this kind of assembler implementation from source code automatically, so we don't expect a high development productivity for this phase with the approach of automatic vectorization.

B. Cost Aggregation

For the cost aggregation phase, the computation of arm lengths is hard to vectorize, because the number of steps for each arm depends on its final length. Since it doesn't contribute much to the total runtime of the phase, we leave it on the host CPU.

The cost aggregation itself is independent for all disparity levels, which again enables easy OpenMP parallelization. It runs in horizontal and vertical phases. In the horizontal phases, each line can be processed independently, in the vertical phases each column, which we can use for vectorization. However the irregular memory access patterns caused by different arm lengths would prevent any CPU SIMD acceleration.

In order to achieve automatic vectorization, in addition to the previously mentioned methods, we added an empty border region to the cost arrays in order to avoid the control statements that otherwise need to make sure that no memory access crosses the array borders. Also we needed to simplify the array indexing in order to help the vectorizer. However, after successful vectorization, we were disappointed to again see a slowdown (middle part of figure 4). The obvious sources of inefficiency were repeated loads of positions for outer array dimensions inside the innermost loops. Also, the loop order is not optimal, since the compiler can only vectorize innermost loops. However, letting the innermost loop iterate over the disparity values and vectorizing the second loop of the loop nest allows reuse of the arm length values.

Manually writing assembly code that optimizes those two aspects enabled us to achieve speedups over the original software version. We also made use of vector register rotation, which helped to save another vector load in the inner loop. However, the vector registers are still not fully utilized by the "Cones" image pair but rather operate on 375 and 450 values for the horizontal and vertical phases respectively, so higher speedups are possible for better suited image dimensions.

C. Scanline Optimization

The scanline optimization runs independently for each set of parallel scanlines. There are two sets of horizontal and two sets of vertical scanlines, which in the current implementation limits the speedup potential of OpenMP parallelization, because for the scanlines that run adjacent to the array dimensions, cache conflicts lead to slowdowns.

```
int difference, threshold; //given
int penalty; //to be computed
// variant with condition
if (difference >= threshold)
        penalty = c1;
else
        penalty = c2;
// equivalent condition free variant
int tmp = difference - threshold;
tmp = min(tmp, 0);
// now tmp==0 for all if cases
tmp = min(-tmp, 1);
// now tmp==1 for all else cases
penalty = (1-tmp) * c1 + tmp * c2;
```

Listing 2. Example for condition-free penalty

However this could be overcome by creating separate cost arrays with switched dimensions for two of the scanline runs. CPU SIMD acceleration for this phase is thinkable, but would require more drastic changes to the array layouts, putting elements of parallel scanline groups into the innermost dimension of all data structures. It would still require an additional step for merging of the resulting horizontal and vertical arrays, which can not be done by CPU SIMD units.

The main obstacles for automatic vectorization were again conditional statements depending on the scanline direction and too complex array access patterns, which had to be solved splitting up arrays and writing a separate function for each scanline direction. Conditional statements that can be translated to minimum operations were automatically resolved by the vectorizer, however the selection of penalty values required much more effort. Making use of the fact that the color differences, which determine the penalty values, can only take discrete values, it was possible to replace the two comparisons and three conditional statements by a series of 18 arithmetic and logic instructions that yield the desired penalty values without masked operations. The pseudocode in listing 2 illustrates the technique for a simpler example.

This sequence was implemented in assembler and included via intrinsic operation. Using the intrinsic and vectorizing the remaining C code after a number of additional transformations again yield slowdowns rather than speedups. Again, the key optimization technique that we were able to apply on assembly level was to interchange the loop order, and optimizing data reuse. Also by using a vector partitioning technique, it was possible to better utilize the vector registers, since vectorizing only image width or height didn't fill the 1024 elements of the vector registers. After these optimizations, we were able to measure a speedup factor of 23.2 compared the assembler version on a HC-1 with scatter-gather RAM to the single core CPU implementation, which is the highest speedup in this case study. This phase profits most from the scatter-gather RAM with a factor 2.02 over the standard RAM. Though all its memory accesses are deterministic, they are still very irregular.

	Data Transfer	Computation
Instructions	8	24
Image Dimensions	450 * 375	
Disparity Values	60	
Directions	4	
Images	2	
Total Amount	2.41 GB	1.94 GFLOP
Required Time SG RAM	0.27 s	
Throughput	8.93 GB/s	7.16 GFLOPS
% of Theoretical Peak	11.2%	18.7 %
% of Saxpy Kernel SG RAM	20.2%	97.3 %

Table II
PERFORMANCE ACHIEVED WITH THE ASSEMBLER VERSION OF SCANLINE OPTIMIZATION

In order to evaluate the utilization of the Vector Personality in this example, we summarize the total size of memory accesses and total number of floating point vector operations of the inner kernel loops of the scanline optimization in table II. We estimate that less than a total of 3% more memory accesses and floating point vector operations occur in the outer loops, most of them for computing the penalty values, and chose to neglect them in these calculations. After dividing the results by the measured runtimes, we can compare the resulting performance values to the peak and measured values from table I. We note that the compute throughput almost matches that of the simple saxpy kernel. Due to a higher ratio of computation to memory accesses, pure bandwidth is much less of an issue in this case, so the latencies between individual instructions seems to be the dominant factor for both examples.

When comparing the functionality of our assembler program on the Vector Personality to how a custom FPGA implementation could work, we note that a number of operations doesn't require the full floating point arithmetic units that we use and thus could be implemented more area efficient and in particular the penalty computation also faster. So we could instantiate a functional unit for each step in the assembler programs inner loop and let the parallel scanlines run through the sequence of functional units pipelined instead of parallel in lockstep. Overall we expect that such a design would both reduce the latency of the pipeline and the used area, which could be used to exploit more parallelism. Also, since the memory access pattern is completely deterministic for this phase, all memory operations could be streamed and connected to the functional units via buffers, which again reduces the latency of the pipeline.

VI. CONCLUSION

We investigated the Convey platform with the FPGAs configured with the Vector Personality for its use for a stereo matching application with abundant parallelism and

different phases with different computational patterns. Not surprisingly, the possibility to leave non runtime critical parts of the algorithm on the host CPU and focusing on the runtime intense part for acceleration significantly facilitated the development process.

However, the automatic vectorization process in its current state did not match our expectations, both in terms of effort and results. The limitations of the vectorizing compiler required much more effort than just adding pragmas to the code. In fact, we needed to apply many source code transformations in order to automatically vectorize at least some phases of the algorithm. Some of those transformations have a negative impact on the software performance of the implementation, so in order to retain the single source principle and still have good performance for all targets, we ended up having multiple code paths. Overall, we spend several weeks to achieve the presented automatic vectorization results, not mentioning the assembly optimizations, whereas we were able to parallelize the entire software version with OpenMP including refinement steps not covered here in a single day. To be fair, considerably more time could be spend on these OpenMP parallelizations to improve scaling by sophisticated cache tiling strategies.

Comparing the vectorization process with a typical FPGA tool flow, maybe even done by developers with as limited background in FPGA design, as we had for developing with the Convey tools, it still seems reasonable. Coming up with a good custom FPGA implementation for the presented algorithm will most likely require even much more time.

Now considering the results, the efforts to enable automatic vectorization were not rewarded by speedups from automatic vectorization. For some steps like the census transform, custom circuits on the FPGA are substantially superior to the configuration of FPGAs as vector processor anyway. However, manual optimizations to the assembler code showed that speedups of at least one order of magnitude are possible with the Vector Personality given the right computational pattern, as seen in the scanline optimization phase. The major limitations of the performance achieved with automatic vectorization are twofold. Firstly, the compiler can only vectorize inner loops of loop nests. However vectorizing outer loops and reusing vector data in the inner loops can be much more efficient. Secondly multidimensional array lookups in the inner loops lead to load operations in the inner loop for each dimension, where some could be moved to outer dimension. We do believe that those optimizations could also be applied automatically by the vectorizing compiler. Also, many time consuming manual code transformations could be avoided by improving the compiler and in particular adding support for more masked vector instructions. If this is done, we think that the presented workflow could become an alternative to the highly productive parallelization with OpenMP. Then the Convey platform would be able to offer the hardware resources to always use parallelization strategy that is best suited for the computational pattern.

ACKNOWLEDGMENT

This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901). The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreement No. 257906.

REFERENCES

[1] vision.middlebury.edu/stereo/.

[2] W. Augustin, V. Heuveline, and J.-P. Weiss. Convey HC-1 hybrid core computer – the potential of FPGAs in numerical simulation. In *Proc. Int. Workshop on New Frontiers in High-performance and Hardware-aware Computing (HipHaC)*. KIT Scientific Publishing, Mar. 2011.

[3] J. D. Bakos. High-performance heterogeneous computing with the Convey HC-1. *Computing in Science and Engineering*, 12(6):80–87, Nov. 2010.

[4] F. Bodin and S. Bihan. Heterogeneous multicore parallel programming for graphics processing units. *Scientific Programming*, 17(4):325–336, Dec. 2009.

[5] T. Brewer. Instruction set innovations for the Convey HC-1 computer. *IEEE Micro*, 30(2):70 –79, march-april 2010.

[6] H. Hirschmüller. Stereo processing by semiglobal matching and mutual information. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(2):328–341, Feb. 2008.

[7] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua. An evaluation of vectorizing compilers. In *Proc. Int. Conf. on Parallel Architecture and Compilation Techniques*, PACT '11, pages 372–382, Washington, DC, USA, 2011. IEEE Computer Society.

[8] X. Mei, X. Sun, M. Zhou, S. Jiao, H. Wang, and X. Zhang. On building an accurate stereo matching system on graphics hardware. In *Proc. ICCV Workshop on GPU in Computer Vision Applications (GPUCV)*, 2011.

[9] B. Meyer, J. Schumacher, C. Plessl, and J. Förstner. Convey vector personalities – FPGA acceleration with an OpenMP-like programming effort? In *Proc. Int. Conf. on Field Programmable Logic and Applications (FPL)*, Aug. 2012.

[10] D. Scharstein and R. Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *Int. Journal on Computer Vision*, 47(1-3):7–42, Apr. 2002.

[11] Xilinx. Virtex-5 FPGA XtremeDSP design considerations user guide, Jan. 2012.

[12] L. Zhang, K. Zhang, T. S. Chang, G. Lafruit, G. K. Kuzmanov, and D. Verkest. Real-time high-definition stereo matching on FPGA. In *Proc. Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, FPGA '11, pages 55–64, New York, NY, USA, 2011. ACM.