System Performance Comparison of Stencil Operations with the Convey HC-1

Revised version 2011/02/17, fixed peak performance of the HC-1

Julian M. Kunkel, Petra Nerge

Research Group: Scientifc Computing
University of Hamburg

2010-11-16

In this technical report our first experiences with a Convey HC-1 are documented. Several stencil application kernels are evaluated and related work in the area of CPUs, GPUs and FPGAs is discussed. Performance of the *C* and *Fortran* stencil benchmarks in single and double precision are reported. Benchmarks were run on Blizzard – the IBM supercomputer at DKRZ –, the working group's Intel Westmere cluster and the Convey HC-1 provided at KIT.

With the Vector personality, performance of the Convey system is not convincing. However, there lies potential in programming custom personalities. The major issue is to approximate performance of an implementation on a FPGA before the time consuming implementation is performed.

Contents

1 Introduction

The main motivation of this work is to do first steps with the Convey HC-1 and to assess the solution's potential. This work does not intent to answer all questions completely. Instead, we plan to further investigate the issue together with HMK Supercomputing and Convey.

The report is structured as follows: First, a brief introduction to hardware architectures, limitations and performance implications is given. Then, some related work and potential performance of stencil operations are discussed. In section 3, excerpts of all evaluated application kernels are provided and discussed. Measured performance on the three systems is assessed in section 4. Then, modifications needed to run applications on the Convey HC-1 are shown. Further experiences are discussed in section 4.4.

1.1 Hardware Architectures

In this section a brief overview of the hardware architectures used in the evaluation are given.

1.1.1 Intel Westmere

Westmere is the 32 nm die shrink of the Nehalem multi-core microarchitecture. An overview is given in figure 1.1[1]. A clock frequency of up to 3.46 GHz is possible.

Machine instructions are decoded and transformed into a sequence of micro-operations. Execution of the instructions and micro-operations is performed out-of-order. One 128 bit operand can be loaded and another one stored per cycle by the load and store units. The correct order is ensured by the Memory Order Buffer, which also optimizes memory throughput.
In theory all ports (execution units) can operate concurrently, that is, one floating point add can be performed on port 1 while one floating point multiplication is executed on port 0. In fact, both ports support SSE instructions. Therefore, SIMD operations could be performed on two double precision or four single precision at once, leading to an aggregated rate of eight single precision floating point operations per cycle.
The Nehalem microarchitecture, retires at most 4 micro-operations per cycle.
Fused multiply add units will be added around 2011 in the Haswell processor architecture. In the future – introduced with Sandy Bridge – Intel Advanced Vector Extensions (AVX) will enhance the vector length to 256 bits, extensions to 512 or 1024 bits are expected. The alignment requirements of the data (currently 128 bit boundaries) are expected to be relaxed.

There are a 32 KiB of L1 instruction cache and a 32 KiB of L1 data cache per core. Additionally, a L2 cache with 256 KiB is available. All cores share a common L3 cache, which contains the contents of the L2 caches of all cores (inclusive cache).
Memory bandwidth is limited by three channels per processor. For example, $1.333\,\text{GHz} \cdot 8\,\text{Bytes} = 32\,\text{GByte/s}$. NUMA interconnect between processors achieve 12.8 GByte/s in full-duplex per QPI link (the number of links depends on the system).

More details of the architecture can be found in [12].
As a side node, data transfer from host to any accelerator connected via PCI Express[2] is limited to 6.4 GByte/s.

1.1.2 IBM Power6

Power6 is a 65 nm dual core processor, which supports clock frequencies of up to 4.7 GHz.

Available execution units include two Fixed Point Units. Fixed point operations can perform 1 cycle back-to-back execution on dependent instructions. Two binary floating point units each perform one fused multiply add operation per cycle. In the pipeline, dependent operations can be started six cycles later. A SIMD unit allows to process four single precision floating point or integer operations per cycle.

Instructions are stored in a 64 KByte L1 instruction cache and processed in-order. Fixed point operations could operate concurrently with floating point operations. There are a 64 KByte L1 data cache and a 4 MByte L2 cache per core. All cores share one 32 MB L3 cache. Memory bandwidth is about 4 GByte/s per core.

More details about the microarchitecture can be found in [8, 10].

[1]Image source is from Wikipedia and licensed under *Creative Commons Attribution-Share Alike 3.0 Unported*
[2]Currently, PCI Express 2 with 16 lanes provides the highest speed. This will double with PCI Express 3.

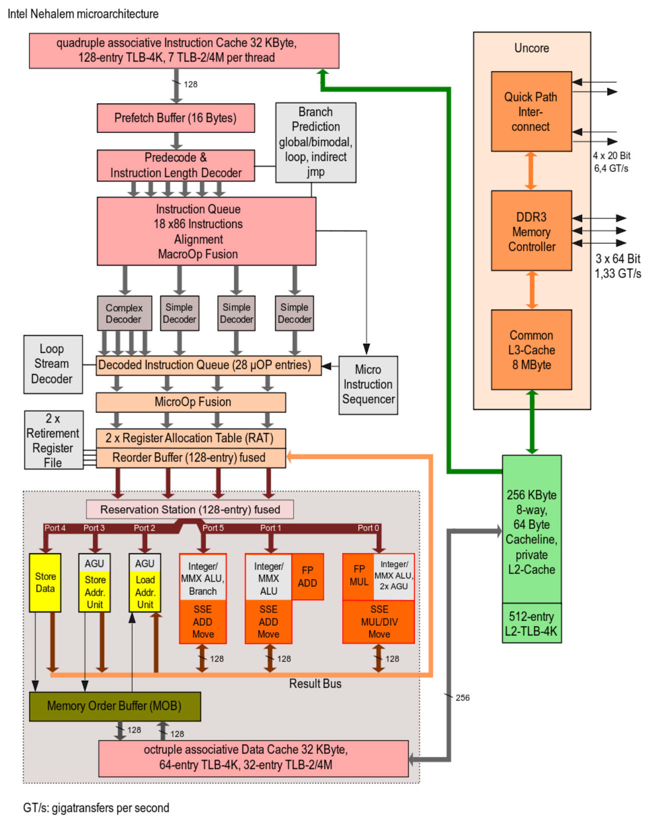
Intel Nehalem microarchitecture
quadruple associative Instruction Cache 32 KByte, 128-entry TLB-4K, 7 TLB-2/4M per thread
128
Prefetch Buffer (16 Bytes)
Branch Prediction global/bimodal, loop, indirect jmp
Predecode & Instruction Length Decoder
Instruction Queue 18 x86 Instructions Alignment MacroOp Fusion
Complex Decoder
Simple Decoder
Simple Decoder
Simple Decoder
Loop Stream Decoder
Decoded Instruction Queue (28 μOP entries)
Micro Instruction Sequencer
MicroOp Fusion
2 x Retirement Register File
2 x Register Allocation Table (RAT)
Reorder Buffer (128-entry) fused
Reservation Station (128-entry) fused
Port 4
Port 3
Port 2
Port 5
Port 1
Port 0
Store Data
AGU Store Addr. Unit
AGU Load Addr. Unit
Integer/ MMX ALU, Branch
SSE ADD Move
Integer/ MMX ALU
SSE ADD Move
FP ADD
FP MUL
Integer/ MMX ALU, 2x AGU
SSE MUL/DIV Move
128
Result Bus
Memory Order Buffer (MOB)
octruple associative Data Cache 32 KByte, 64-entry TLB-4K, 32-entry TLB-2/4M
256
Uncore
Quick Path Inter-connect
4 x 20 Bit 6,4 GT/s
DDR3 Memory Controller
3 x 64 Bit 1,33 GT/s
Common L3-Cache 8 MByte
256 KByte 8-way, 64 Byte Cacheline, private L2-Cache
512-entry L2-TLB-4K
GT/s: gigatransfers per second

Figure 1.1: Nehalem microarchitecture of the quad-core variant.

1.1.3 Convey HC-1

Convey HC-1 is a hybrid machine, which hosts one quad-core Xeon processor and one co-processor. In the dual socket machine one processor is replaced by an interconnect to the co-processor, which consists of four Xilinx Virtex-5 FPGAs and a memory subsystem. Memory configuration of the accelerator varies between 8 to 16 DDR2 channels (PC-5300/667 MHz), which achieve a maximum bandwidth of up to 80 GByte/s. A maximum of 128 GByte of memory can be integrated. Convey allows to use COTS memory or special scatter-gather memory, which allows data to be accessed in a fine granularity of 8 bytes instead of accessing whole cache lines of 64 bytes.

There are several important aspects of the technology compared to GPU accelerated systems. Memory of the host and co-processor can be accessed in a coherent way from both devices. However, device-local memory can be accessed much faster. The FPGAs can be reprogrammed with custom execution units to match a particular workload. Convey calls one configuration of the FPGAs a *personality*. Available personalities include a single precision vector personality with 32 function pipes. Each pipe provides one SIMD instruction 4 fused multiply add operations. Peak performance is at 76.8 Gflop/s (32 pipes · 4 FMA/pipe · 2 FLOP/FMA · 0.300 GHz).[3] The double precision personality has 16 function pipes.

Custom personalities can be created with the Personality Development Kit. The logic must be programmed in Verilog using the development suite from Xilinx.

1.2 Stencil Operations

Iterative algorithms often update the value of a grid point depending on the values of the neighbors. A stencil describes a fixed pattern of how to weight the neighbors and the old value of the grid point. An exemplary 2D stencil, which smoothens/interpolates values of the grid is given below in pseudo code:

```
a = b = c = d = 0.25
for i:    // for all rows
 for j:   // for all columns
   m[i][j] = a * o[i][j+1] + b * o[i][j-1]  + c * o[i-1][j] + d * o[i+1][j]
```

The performance of stencil operations depends on the memory bandwidth. In this case values for 4 neighbors are read, 3 additions and 4 multiplications are done, and one value is updated. Dependencies between loop

[3]This text is modified in the revised version, as the frequency of a personality varies up to 300 MHz. Both vector personalities run at 300 MHz and not at 150 MHz.

iterations do not exist. Therefore, the loop elements can be processed independently (concurrently if possible). An SIMD vectorization, however, is not directly applicable, because left and right neighbors are required for the computation.
Due to the common use of stencil operations plenty of research has been conducted. In the following an excerpt of performance numbers and achievements of recent research is provided.

In [6] multiple algorithms of 3D (double precision) stencil operations are evaluated on Itanium 2, Opteron, Power5 and Cell architecture. Depending on the architecture, the optimizations necessary to achieve best performance changes. The naive unaliased implementation already achieved 2 Gflop/s on the Power5 and 1.3 Gflop/s on the Itanium 2. However, the Itanium 2 can reach 1.7 Gflop/s with a complex cache usage optimization. The Cell architecture won by achieving 5.5 Gflop/s at 2.4 GHz clock frequency and 7.3 Gflop/s at 3.2 GHz respectivly.

Automatic tuning of stencil codes towards an architecture is evaluated in [4]. The dimension of the double precision 3D stencil grid is 256^3. Evaluated systems are NVIDIA GTX280, AMD Barcelona (Opteron 2356), Intel Clovertown (E5355) and IBM PowerXCell 8i. With one core about 1 Gflop/s was achieved on compute systems and 2.3 Gflop/s on the Cell by using SIMD. To enable scaling on common CPU architectures more sophisticated tuning is required. The influence of optimization methods with regards to different numbers of cores is shown nicely, several optimization methods are introduced and evaluated. Enabling all discussed optimizations leads to 2.5 Gflop/s on Clovertown (8 cores), 7 Gflop/s on Barcelona (8 cores), and 16 Gflop/s on Cell Blade (16 cores). A naive first implementation on a GeForce GTX280 achieved 1.4 Gflop/s. In this case, in each time step data is copied between host and GPU. If data could reside completely in device memory 10 Gflop/s can be achieved. 35 Gflop/s are possible with a modified scheme and without the overhead of copying data between the host and device memory.

Newer results for the auto-tuning library ([5]) use a Laplacian, Divergence and Gradient 3D stencil and show performance for several systems, among which are the Intel Nehalem and GTX280. The author assesses performance limitations of the kernels and the architecture based on flop/s and memory bandwidth. The auto-tuned Nehalem implementation is close to the theoretical peak performance of 10 Gflop/s for Laplacian, 6 Gflop/s for Divergence kernels and 3.5 Gflop/s for the Gradient kernel. The GTX280 performance is 13, 15 and 9 Gflop/s respectively. Further work of their group is to demonstrate the application of their auto-tuning to an icosahedral atmospheric climate simulation.

Peng et.al. [9] provide a scheme to parallelize stencil operations with SIMD instructions by using shuffle operations. In detail, they optimize stencil operations for the Lattice-Boltzmann method and finite-difference time-domain for seismic code by using SIMD instruction level parallelism and node parallelism with MPI. Evaluation is performed on the following platforms: Cell, Intel quad-core, IBM BlueGene/L and BlueGene/P. The spatial decomposition of the Lattice-Boltzman scaled up to 200,000 processors on the BlueGene/L. SIMD improved the speed of the computation part of the parallel Lattice-Boltzman method by a factor of 3.5 on the Cell with single precision. The SIMD technique was applied to the seismic code on a quad-core Xeon with 2.33 GHz, achieving a speedup of 2–3 with varying numbers of threads.

In [7] a new Fast Iterative Method is proposed and implemented to solve a class of Hamilton-Jacobi equations. On an NVIDIA GeForce 8800 GTX GPU a grid with the dimensions 256x256x100 this leads to a speedup of 100.

Jacobi's iterative method for the 2D Poisson equation is optimized for GPUs in [13]. Additionally, hybrid code utilizing GPUs and CPUs concurrently is discussed and evaluated. In the paper the authors discuss the Jacobi method and several optimization techniques for GPUs. On an NVIDIA C1060 a single precision performance of 36 Gflop/s and double precision performance of 16 Gflop/s is observed. This corresponds to memory bandwidths of 70 GByte/s (SP) and 60 GByte/s (DP). The overhead of data transfer between host and device is measured: with 100 iterations performed in the GPU the data transfer is negligible.

Several GPU implementations for a single precision Jacobi solver are discussed in [1]. Additionally, CPU versions are compared. The following GPUs are evaluated and embedded in different host systems: GeForce 8600GT, GeForce 8800GT and GTX280. With the GPU between about 8 and 68 Gflop/s have been achieved (with an observed memory bandwidth of up to 130 GByte/s). Results for the CPU are below 1 Gflop. However, a proper comparison is missing and the CPU peak seems too high for the quad-core, because only one core is utilized. Therefore, their comparison between GPU and CPUs must be critically assessed.

In [3] the 2D Jacobi method is evaluated on Tesla C870, C1060 and a Core 2 Quad Q9450 CPU. On the CPU the best performance is observed with 1 core and 1 thread in most cases. However, tests were conducted with up to 4 cores and 16 threads. On an input matrix with the dimension 8192^2 the C1060 outperformed the CPU by a factor of 5.5. With a small matrix the CPU behaved better than a C870, but the C1060 was twice as fast.

In [2] Reverse Time Migration – the most advanced seismic imaging technique – is evaluated on Tesla C1060, Cell and Xeon E5460. Additionally, a stencil implementation is discussed for the Convey HC-1. To estimate performance on the Convey HC-1 they consider memory bandwidth and available execution units on the Xilinix FPGAs. The execution time of the problem was lowest with GPUs followed by FPGAs. Previous work done in their group was an FPGA implementation on the Virtex-5 LX200 FPGA which showed a performance of 55.5 Gflop/s ([11]). Basically, they projected how a FPGA implementation on the Convey HC-1 would perform based on the resources available on the FPGA and the knowledge gained by their previous work. Development time for the GPU solution was lower due to the mature CUDA development tools.

Summing up, the related work confirms that – in tendency – optimization to an "acceptable" level of peak-performance is easier on CPU architectures and harder on exotic architectures like Cell, GPUs and FPGAs. However, to utilize available resources on modern CPUs vectorization of the stencil algorithm is required.

2 Evaluation

To assess the behavior of the HC-1 system, several existing codes are run on one core of a Westmere Linux Cluster, the Blizzard Power6 supercomputer at DKRZ and on the Convey HC-1. Application performance in terms of flop/s are measured and compared. Code is run at least three times. In all cases consistent results were observed.

2.1 Test Systems

Because only one logical CPU is used to assess performance, an introduction to aspects related to a single node is sufficient. The floating point capabilities of one processor of each system are shown in table 2.1.

Westmere Linux Cluster

Nodes on the Linux Cluster are equipped with two Intel Westmere hexa-core CPUs at 2.6 GHz (Intel Xeon 5650). 12 GByte of memory operate at 1,333 MHz. Ubuntu 10.04 is installed with a 2.6.32-24-server kernel. Code was compiled with GCC 4.4.3. For the measurement the TurboBoost feature was disabled, although it improves performance of our examples by 15%.

Blizzard Power6 Supercomputer

One node of the IBM supercomputer consists of 16 node cards each with a IBM Power 6 at 4.7 GHz. Blizzard runs with AIX 6.1. On the system GCC 4.4.2 and the IBM compilers XL C/C++ for AIX (V10.1) and XL Fortran for AIX (V12.1) are available.

Convey HC-1

Thanks to the Karlsruhe Institute of Technology and HMK Supercomputing access to a Convey HC-1 in Karlsruhe was possible.

We used the following compiler version:

```
Convey64 Compiler Suite: Version 2.0.0
Built on: 2010-07-14 07:30:59 -0500
Thread model: posix
GNU gcc version 3.3.1 (Convey64 2.0.0 driver)
Portions Copyright (c) 2007-2010 Convey Computer Corporation
Portions Copyright (c) 2002-2005 PathScale, Inc.
Portions Copyright (c) 2000-2001 Silicon Graphics, Inc.
All Rights Reserved.
Convey64 Compiler Suite: Version 2.0.0
Built on: 2010-07-14 07:30:59 -0500
Thread model: posix
GNU gcc version 4.2.0 (Convey64 2.0.0 driver)
```

	Intel Westmere	IBM Power6	Convey HC-1
processor	32 *nm* die shrink Nehalem multicore	65 *nm* dual core	Xilinx Virtex5 FGPA co-processor
clock frequency	2.66 GHz	4.7 GHz	0.300 GHz
floating point operations:			
single precision	8 per cycle	4 + 4 (VMX) per cycle	$4 \cdot 2$ per cycle
double precision	4 per cycle	4 per cycle	$4 \cdot 2$ per cycle
single precision Gflop/s	21.28 per core	36.4 per core	2.4 per pipe
double precision Gflop/s	10.64 per core	18.8 per core	2.4 per pipe

Table 2.1: Floating operation capabilities of available hardware architectures.

3 Compute Kernels

Compute kernels are split into three groups. The first one is an example provided by Convey which was already ported to the HC-1. The second one consists of stencils taken from a 2D Jacobi PDE solver. The last set of examples is a benchmark of index methods for an ocean model. Some applications are coded in Fortran and some in C.

3.1 Convey Example

Convey provides an example of a 3D stencil kernel in Fortran and in C. Both use single precision floating point operations.

Listing 3.1: Convey Fortran 3D kernel

```
integer(8), parameter:: NX=1000, NY=1000, NZ=100

do k=2,NZ-1
  do j=2,NY-1
    do i=2,NX-1
      p2(i,j,k) = sp*p1(i,j,k) + sx1 * p1(i-1,j  ,k  )      &
                               + sx2 * p1(i+1,j  ,k  )      &
                               + sy1 * p1(i  ,j-1,k  )      &
                               + sy2 * p1(i  ,j+1,k  )      &
                               + sz1 * p1(i  ,j  ,k-1)      &
                               + sz2 * p1(i  ,j  ,k+1)
    enddo
  enddo
enddo
```

3.1.1 Convey example only additions

The Convey example is rewritten to use only additions. Architectures which allow concurrent additions and multiplications could be briefly rated. For example, Power6 allows fused multiply add operations and Westmere has execution units to process both operations concurrently.

Listing 3.2: Convey Fortran 3D kernel modified to use only additions

```
      p2(i,j,k) = sp+p1(i,j,k) + sx1 + p1(i-1,j  ,k  )      &
                               + sx2 + p1(i+1,j  ,k  )      &
                               + sy1 + p1(i  ,j-1,k  )      &
                               + sy2 + p1(i  ,j+1,k  )      &
                               + sz1 + p1(i  ,j  ,k-1)      &
                               + sz2 + p1(i  ,j  ,k+1)
```

3.2 Jacobi PDE Solver

The Jacobi PDE application iteratively solves the Poisson equation in a 2D grid. The size of the grid and the iteration or accuracy can be specified on the command line. In the example a dimension of 800x800 with an iteration count of 10,000 is used.

Listing 3.3: Original C kernel of the Jacobi PDE

```
  for(i=1;i<N;i++)                  /* over all rows     */
  {                                 /*                   */
    for(j=1;j<N;j++)                /* over all columns  */
    {
      star =  Matrix[m2][i-1][j] + Matrix[m2][i][j-1]
            + Matrix[m2][i][j+1] + Matrix[m2][i+1][j]
            - 4.0*Matrix[m2][i][j];
            residuum=getResiduum(i, j, star);
            korrektur=residuum;
            // if (residuum<0) residuum=residuum*(-1);
            residuum = (residuum<0) ? -residuum: residuum;
            // track the maximum residuum to approximate the error.
            maxresiduum = (residuum < maxresiduum) ? maxresiduum : residuum;
            Matrix[m1][i][j]=Matrix[m2][i][j]+korrektur;
        }
    }
    ...

double getResiduum(int x, int y, double star)
{
  if (inf_func==FUNC_F0)
    return star*0.25;
  ...
}
```

3.2.1 Jacobi PDE solver simple stencil

The branching and function call inside the original C kernel prevents vectorization, therefore, a reduced kernel is measured, too.

Listing 3.4: Simplified loop for the PDE solver

```
    for(i=1;i<N;i++)                  /* over all rows     */
    {                                 /*                   */
      for(j=1;j<N;j++)                /* over all columns  */
      {
        Matrix[m1][i][j]= ( Matrix[m2][i-1][j] + Matrix[m2][i][j-1]
                          + Matrix[m2][i][j+1] + Matrix[m2][i+1][j]
                          ) * 0.25   ;
      }
    }
```

3.2.2 Manual loop body optimization

Naturally, equations are evaluated strictly from left to right, which prevents concurrent computation within one expression. Depending on the flags, a compiler could change the order. The presented manual loop body allows to perform two independent additions explicitly.

Listing 3.5: Manual loop body optimization

```
        register double a = Matrix[m2][i-1][j];
        register double b = Matrix[m2][i][j-1];
        register double c = Matrix[m2][i][j+1];
        register double d = Matrix[m2][i+1][j];

        register double e = a + b;
        register double f = c + d;
        Matrix[m1][i][j]=  (e + f) * 0.25  ;
```

3.2.3 Explicit pointer aliasing

Optimizing pointer access in C depends on the C aliasing rules. In *C99*, the “`restrict`” keyword specifies that a pointer argument does not alias any other pointer argument. The impact of the proper usage of the *C99* feature is tested on the code. Both GCC and XLC explicitly honor the `restrict` keyword.

3.2.4 Jacobi PDE solver single precision

In this test the Jacobi simple stencil is modified to use single precision floating point arithmetic.

3.3 ECOHAM

Stencil operations are common in climate simulations. Especially, physical processes are described by partial differential equations, which are often expressed by the finite-difference method. Modeling of ocean processes relies on the topography, which defines the regions of land and the depth of the ocean. For the simulations the underlying topography will be expressed by a discrete 3-dimensional grid or matrix, respectively. This matrix defines the topology of the simulated parameters. However, only wet points – that is, grid points with water – and their neighbors are of interest for the simulations. Grid points with land are useless. There are plenty possible ways to index the grid, while the best index method depends on the architecture and compiler. The implementation allows to use single and double precision.

In our example we chose the topography of the North Sea, with a spatial resolution of approximately 3 km, the latitude is decomposed by 380 grid points and the longitude is decomposed by 414 grid points. The depth is decomposed by 30 levels. The ratio of the wet grid points to all grid points is 20%.

Five different grid indexing methods have been chosen for evaluation.

3.3.1 3D indexing of the whole grid

The first indexing method is similar to the Convey example, only with different array dimensions. The whole 3-dimensional grid is indexed and the calculation includes all grid points. Values for land points are simply 0 and they do not matter for the simulation.

Listing 3.6: 3D indexing with calculation for all grid points (similar to the Convey example)

```
integer(kind=4), parameter:: iMax=380, jMax=414, kMax=30

do i = 2, iMax-1                 !over all lattitudes / rows
  do k = 2, kMax-1               !over all depth levels
    do j = 2, jMax-1             !over all longitudes / columns
      p2(j,k,i) = sp*p1(j,k,i) + sy1 * p1(j-1,k   ,i   )      &
                               + sy2 * p1(j+1,k   ,i   )      &
                               + sz1 * p1(j   ,k-1,i   )      &
                               + sz2 * p1(j   ,k+1,i   )      &
                               + sx1 * p1(j   ,k   ,i-1)      &
                               + sx2 * p1(j   ,k   ,i+1)
    enddo
  enddo
enddo
```

3.3.2 3D indexing of the wet grid points

The second indexing keeps the whole 3-dimensional grid in memory, but only the wet grid points are computed.

Listing 3.7: 3D indexing with calculation for the wet grid points

```
integer(kind=4), parameter:: iMax=380, jMax=414, kMax=30

!counts the wet grid points for each row and depth level:
!  jWet(k,i)
!calculates the index j for each jWet(k,i)              :
!  jWet3D(jWet,k,i)

do i = 2, iMax-1                      !over all lattitudes / rows
  do k = 2, kMax-1                    !over all depth levels
    do ij = 1, jWet(k,i)              !over wet grid points &
      j      = jWet3D(ij,k,i)         !of all longitudes / columns
      west   = p1(j-1,k   ,i   )      !because than it runs faster
      east   = p1(j+1,k   ,i   )
      above  = p1(j   ,k-1,i   )
      below  = p1(j   ,k-1,i   )
      north  = p1(j   ,k   ,i-1)
      south  = p1(j   ,k   ,i+1)
      center = p1(j   ,k   ,i   )
      p2(j,k,i) = sp*center + sy1 * west        &
                            + sy2 * east        &
```

```
                                      + sz1 * above        &
                                      + sz2 * below        &
                                      + sx1 * north        &
                                      + sx2 * south
      enddo
    enddo
  enddo
```

3.3.3 1D indexing of the wet grid points and their related dry neighbor grid points

In the following, three ways of how to pack the 3-dimensional grid points into a 1D vector are described.

The 3-dimensional wet grid points and their related dry neighbor points are packed into a vector. Therefore, a mask of the wet points and their related dry neighbor points is created and packed with the Fortran routine `pack()`. This routine preserves the memory packing order of the 3-dimensional order, therefore, the grid (j, k, i) is stored in the order j, k, i.

To keep the implementation simple – compared to the 3D indexing of just the wet points – points on the boundary are included in the computation.

Listing 3.8: 1D indexing of the wet grid points and their related points

```
integer(kind=4), parameter:: iMax=380, jMax=414, kMax=30

!counts the wet grid points:
!  CountWetPoints
!counts the wet grid points and theirs related dry grid points:
!  CountWetRelatedDryPoints

!initialising vector out of the pack of the wet grid points and
!their related neighbour points in the 3D grid:
!  p1(CountWetRelatedDryPoints)

!calculates the index of wet points in the vector p1:
!  CalcPoints(CountWetPoints)

!calculates the index of the related points in the vector p1:
!  CalcWestPoints (CountWetPoints) , CalcEastPoints (CountWetPoints)
!  CalcAbovePoints(CountWetPoints) , CalcBelowPoints(CountWetPoints)
!  CalcNorthPoints(CountWetPoints) , CalcSouthPoints(CountWetPoints)

center => p1
west   => p1 ; east  => p1
above  => p1 ; below => p1
norht  => p1 ; south => p1

do i = 1, CountWetPoints
  p2(CalcPoints(i)) = sp *center(CalcPoints(i))         &
                    + sy1*west  (CalcWestPoints(i))     &
                    + sy2*east  (CalcEastPoints(i))     &
                    + sz1*above (CalcAbovePoints(i))    &
                    + sz2*below (CalcBelowPoints(i))    &
                    + sx1*north (CalcNorthPoints(i))
                    + sx2*south (CalcSouthPoints(i))
enddo
```

3.3.4 1D indexing of the wet grid points

Now only the 3-dimensional wet grid points are packed into a vector. Like in the previous implementation the Fortran function `pack()` is used. To avoid conditions in the code one additional element is appended as a dummy to represent dry neighbor grid points. All dry points reference this dummy.

Listing 3.9: 1D indexing of the wet grid points and their related points

```
integer(kind=4), parameter:: iMax=380, jMax=414, kMax=30
```

	3D all	3D wet	1D wet + related dry	1D wet	1D wet fixed ordering
Lattitude: iMax	380	380	380	380	380
Longitude: jMax	414	414	414	414	414
Depth : kMax	30	30	30	30	30
Array Dimensions	jMax, kMax, iMax	jMax, kMax, iMax	wet points + related dry points	wet points	wet points
Array Elements	4719600	4719600	1060980	952450	952450
Array Size	18.9 MByte	18.9 MByte	4.2 MByte	3.8 MByte	3.8 MByte
Valid Points of Calculation	4360608	862170	952450	952450	952450

Table 3.1: Summary of ECOHAM indexing variants (single precission).

```
!counts the wet grid points:
!  CountWetPoints

!initialising vector out of the pack of the wet grid points
!and one additional element with value null, which are
!for all land grid points
!  p1(CountWetPoints)
!  p1(0) = 0.

!calculates the index of the related points in the vector p1:
!  RelatedWestPoints (CountWetPoints) , RelatedEastPoints (CountWetPoints)
!  RelatedAbovePoints(CountWetPoints) , RelatedBelowPoints(CountWetPoints)
!  RelatedNorthPoints(CountWetPoints) , RelatedSouthPoints(CountWetPoints)

do i = 1, CountWetPoints
   west (i) = p1(RelatedWestPoints (i))
   east (i) = p1(RelatedEastPoints (i))
   above(i) = p1(RelatedAbovePoints(i))
   below(i) = p1(RelatedBelowPoints(i))
   north(i) = p1(RelatedNorthPoints(i))
   south(i) = p1(RelatedSouthPoints(i))
enddo

do i = 1, CountWetPoints
   p2(CalcPoints(i)) = sp*center(i) + sy1*west (i)    &
                                    + sy2*east (i)    &
                                    + sz1*above(i)    &
                                    + sz2*below(i)    &
                                    + sx1*north(i)    &
                                    + sx2*south(i)
enddo
```

3.3.5 1D indexing of the wet grid points with fixed order (ECOHAM)

In this indexing method the 3 dimensional wet grid points are packed into a vector. The ordering of the packing is fixed to first store by depth levels (k), second by longitudes (j) and third by latitudes (i). This original indexing approach is used for biological processes which depend only on local processes. Dry neighbors are not important in this case. Related dry neighbor grid points are neglected and the index is set to the local grid point.

Listing 3.10: 1D indexing of the wet grid points and their related points

```
integer(kind=4), parameter:: iMax=380, jMax=414, kMax=30

!counts the wet grid points:
!  CountWetPoints
```

```
!initialising vector of the wet grid points with
!fixed ordering: 1. depth levels (k),
!2. longitudes (j), 3. lattitudes (i)
!  p1(CountWetPoints)

!calculates the index of the related points in the vector p1:
!  RelatedWestPoints (CountWetPoints) , RelatedEastPoints (CountWetPoints)
!  RelatedAbovePoints(CountWetPoints) , RelatedBelowPoints(CountWetPoints)
!  RelatedNorthPoints(CountWetPoints) , RelatedSouthPoints(CountWetPoints)

do i = 1, CountWetPoints
  p2(j,k,i) = sp*p1(i) + sy1*p1(RelatedWestPoints(i))   &
                       + sy2*p1(RelatedEastPoints(i))   &
                       + sz1*p1(RelatedAbovePoints(i))  &
                       + sz2*p1(RelatedBelowPoints(i))  &
                       + sx1*p1(RelatedNorthPoints(i))  &
                       + sz1*p1(RelatedSouthPoints(i))  &
enddo
```

4 Results

Compilation of the source code is done mainly with "`-O3`". GCC is used with the profile feature. That is, code is compiled with profiling, run and then recompiled by using knowledge about the runtime behavior to optimize it. This leads to additional performance gains of about 10 percent on the Intel system.

4.1 Convey Example

A comparison between the compute clusters and compiler flags used is shown in figure 4.1. The efficiency in terms of peak-performance is given in figure 4.3. Using GCC a single Westmere core delivers the highest performance for the C code. Fortran and C are both identically fast on the Westmere. The code is not vectorized with SIMD instructions due to the algorithmic specification. Therefore, two flop could be achieved per cycle (2.66 GHz). This corresponds to 73.7% of the potential performance the algorithm could achieve on the architecture. The variant in which only additions are performed achieves 1390 Mflop/s, which is about 50.5% of the achievable performance. This is due to the fact that only the floating point adder unit is used. Performance of a single precision variant achieved similar performance.

On Blizzard, the C compiler from IBM achieves a performance of about 1380 Mflop/s, which is similar to GCC. Output of `hpmccount` to profile hardware counters are shown in 4.1. This corresponds to 7.3 percent of the peak performance or 8.6 percent of the achievable performance on the architecture (6 out of 7 operations for the expression could be fused multiply add operations, both floating operation units could work concurrently). At this point a deeper investigation is necessary.

First, it should be checked whether there are any unnecessary operations added by the compiler. The total amount of floating point operations is approximately the ones needed to perform the computation. Every 2.5th instruction is one floating point instruction (`PM_RUN_INST_CMPL/PM_FPU_FLOP` = 2.51). Most operations are fused multiply operations, that is, one and a half operation is needed to prepare the floating point operation. 7 operands must be loaded and the result must be stored. 7 floating point instructions are needed to perform the calculation. Therefore, approximately 3 additional instructions per loop iteration are needed.

Second, the value of instructions per run cycle (IPC) is low, that is, smaller than 1 and therefore the operations stall on the CPU. Checking the assembler showed loop unrolling is done. However, the processing is different to the Fortran compiler. The problem is shown in listing 4.2. In C, the compiler enforced proper evaluation of the expressions, from left to right (no register renaming on Power6). On a Power6 the pipeline takes 6 cycles to feed results from the FPU back. Therefore, during the main computation of a loop iteration 25 cycles are wasted: 15 instructions are run in approximately 35 cycles, which results IPC of 2.33. The reason for the stall is found as the C compiler generates very inefficient assembler for the problem code.

Listing 4.1: Excerpt of hpmcount for the Convey example in C

```
 Total amount of time in user mode          : 37,135217 seconds

 Instructions per run cycle                  :          0,397
 Total floating point operations             :      50756,370 M
 Flop rate (flops / WCT)                     :       1361,546 Mflop/s
 Algebraic flop rate (flops / WCT)           :       1361,546 Mflop/s
FMA percentage                               :         92,308 %
% of peak performance                        :          7,251 %

PM_FPU_1FLOP (FPU executed one flop instruction )               :  3904339462
PM_FPU_FMA (FPU executed multiply-add instruction)              : 23426015170
PM_FPU_FSQRT_FDIV (FPU executed FSQRT or FDIV instruction)      :          21
PM_FPU_FLOP (FPU executed 1FLOP, FMA, FSQRT or FDIV instruction): 27330354653

PM_L2_LD_REQ_DATA (L2 data load requests)      :     1486489341
PM_L2_LD_MISS_DATA (L2 data load misses)       :      738380370
PM_L2_ST_REQ_DATA (L2 data store requests)     :      976719243
PM_L2_ST_MISS_DATA (L2 data store misses)      :      244632483
PM_RUN_INST_CMPL (Run instructions completed)  :    69465854554
PM_RUN_CYC (Run cycles)                        :   174948086689
```

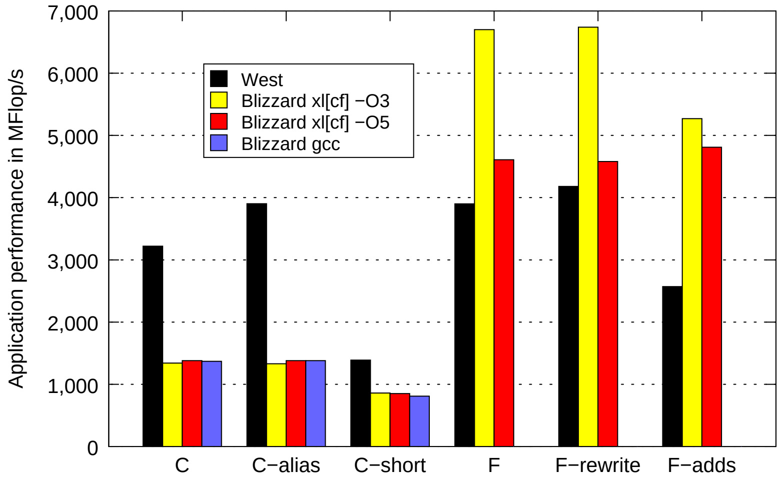

Figure 4.1: Stencil operation example from Convey run on compute architectures.

Listing 4.2: Assembler code excerpt for the XLC compiler output

```
stfs     fp0,4(r25)
lfs      fp10,8(r28)
fmuls    fp0,fp2,fp10
lfs      fp8,8(r26)
lfs      fp9,4(r24)
lfs      fp10,8(r27)
lfs      fp11,8(r23)
lfs      fp12,0(r24)
lfs      fp13,8(r24)
fmadds   fp0,fp1,fp9,fp0
fmadds   fp0,fp3,fp8,fp0
fmadds   fp0,fp4,fp10,fp0
fmadds   fp0,fp5,fp11,fp0
fmadds   fp0,fp6,fp12,fp0
fmadds   fp0,fp7,fp13,fp0
```

With Fortran the IBM compiler works better. Note that including these optimizations the Fortran compiler reorders the evaluation of the expressions, which means that the numeric differs from normal evaluation. If a user enforces proper evaluation, the created assembler code will probably look similar to the one created for the C Code. In theory, better optimization of the loop is possible to gain both performance and accuracy. This, however, requires manual tuning. Interestingly "`-O3`" achieved 6,700 Mflop/s while "`O5`" reported 4,610 Mflop/s. 6,700 Mflop/s are about 35 % of the hardware peak performance.

Looking at the hardware counter revealed that additional compiler optimization reduced the total number of floating point operations. However, the total number of floating point operations is increased. In detail, the fused multiply add operations are reduced but many more single floating point operations are created.

The tests in which a shorter loop is used to demonstrate the (well known) necessity of long inner loop bodies. In the case in which only additions are used, the Westmere architecture is limited to one flop/cycle. This cuts performance almost to a half. The Power6 does not suffer so much, yet the efficiency of the system lacks behind Westmere.

Now we compare the results with the results measured on the Convey HC-1. Performance results are visualized in figure 4.2. With proper compiler pragmas, the Convey compiler achieves 22 Gflop/s, which corresponds to 28% of the system peak performance with the vector personality. This high efficiency requires that the SIMD feature is used by the compiler.

4.2 Jacobi PDE Solver

Results for the 2D PDE solver are provided in figure 4.4. The modified code to run it on the system is shown in listing 4.3. About 2 Gflop/s are measured on the Westmere. Due to the problem statement and without the programmer's knowledge the compiler can achieve at most 2.66 Gflop/s. Performance on the Blizzard is low due to IBM's XLC compiler. For the original program the additional optimizations with "`-O5`" improved performance of the original problem at least to the level of the simple stencil kernel. Single precision improves performance only slightly. However, in the Convey case, more pipelines can be implemented in the FPGA than

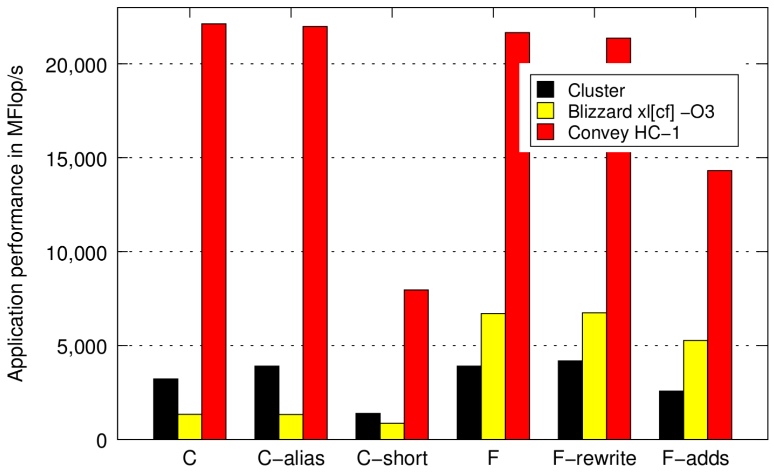

Figure 4.2: Stencil operations example from Convey – comparison between HC-1 performance and others.

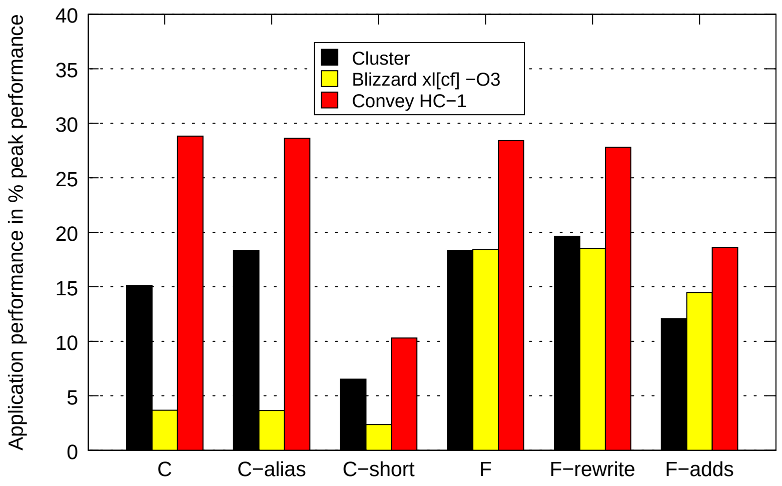

Figure 4.3: Efficiency of the stencil operation example from Convey.

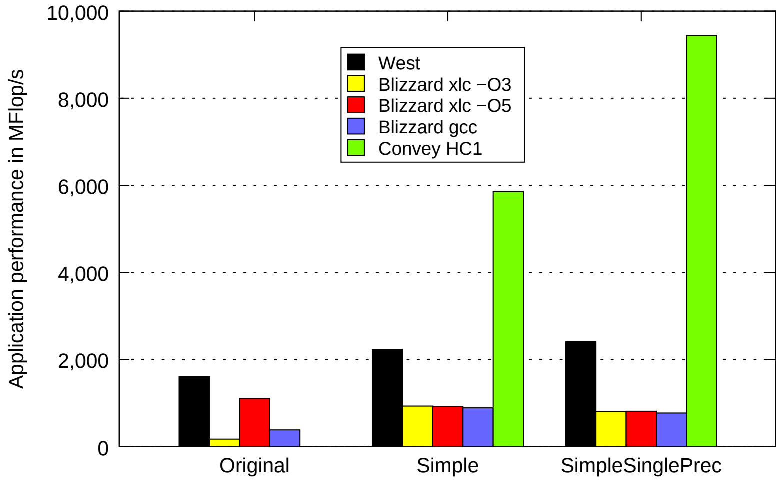

Figure 4.4: Variants of a Jacobi PDE solver.

for the double precision case. The performance on the Convey system sticks behind the results for the 3D stencil.

Listing 4.3: Jacobi PDE solver simple stencil with Convey pragmas

```
#pragma cny dual_target(1)
void calculate(double * restrict *restrict tm,
      double const * restrict *restrict sm, int n)
{
  long i,j;                                  /* local variables for loops */
  double val = 0.25;

#pragma cny array(tm[n][n])
#pragma cny array(sm[n][n])
  for(i=1;i<n-1;i++)                                /* over all rows     */
  {
#pragma cny no_loop_dep(tm, sm, Matrix)
    for(j=1;j<n-1;j++)                              /* over all columns  */
    {
      tm[i][j]= ( sm[i-1][j] + sm[i][j-1] + sm[i][j+1] + sm[i+1][j] ) * val;
    }
  }
}
#pragma cny dual_target(0)

int main(){
...
#pragma cny begin_coproc
  calculate(Matrix[m1], Matrix[m2], N+1);  /* solve the equation  */
#pragma cny end_coproc
...
}
```

The manual loop body optimization to enable concurrent processing of the equation improves the performance on Westmere from 3,900 Mflop/s to 4,180 Mflop/s. On Blizzard, the manual loop body does not improve performance, because the evaluation of equations is reordered by the XLF compiler.
A few additional tests have been performed to evaluate performance of higher (and lower) dimensions for the matrix. Due to cache misses, performance of all CPU systems is reduced, but efficient pre-fetching mostly mitigates the performance loss. The values provided in this section are created by using the best parameter set we found.

4.3 ECOHAM

Porting the Fortran code to the Convey required some minor modifications. Basically the loop had to be extracted into a separate function – see section 4.4 for more information. Results for the different indexing

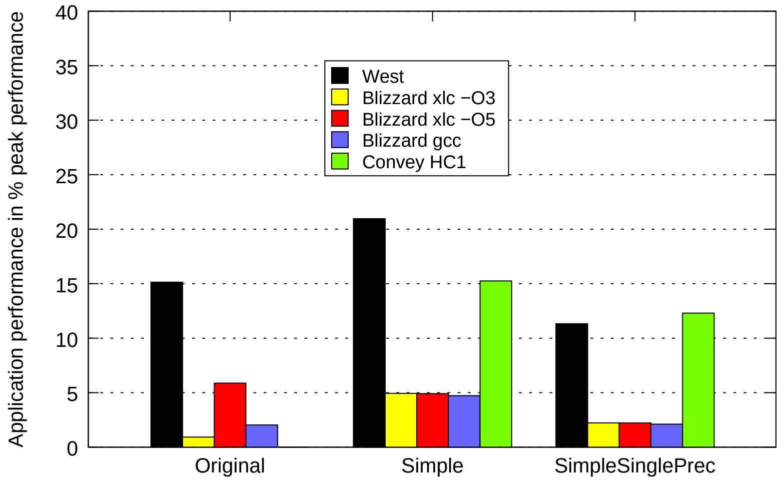

Figure 4.5: Efficiency of variants of a Jacobi PDE solver.

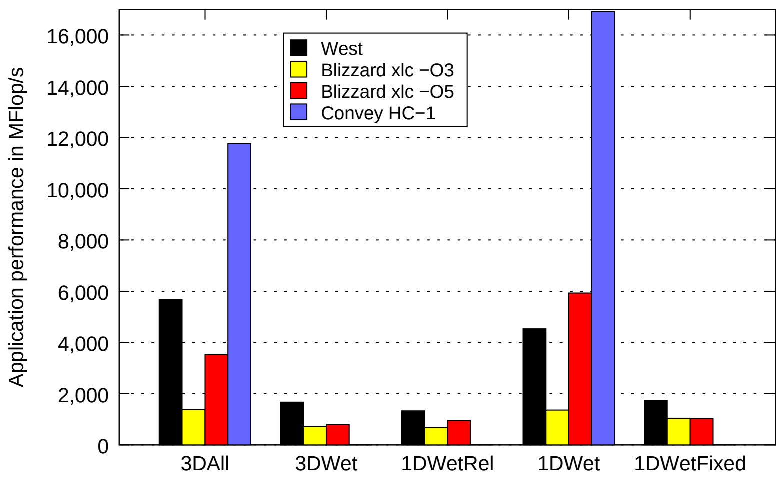

Figure 4.6: Index variant evaluation for a climate code.

methods are provided in figure 4.6. The application performance reported does not care about additional computation for dry points. Therefore, the really interesting flop/s for the “3DAll” case must be multiplied with 0.2, because roughly 20% points are wet points. The best indexing method is “1DWet”. Interestingly, on the Westmere architecture the achievable flop/s are higher than for “1DWet”, for Blizzard and Convey “1DWet” wins. With “1DWet” 17 Gflop/s can be observed on the Convey HC-1. However, on Blizzard 6 Gflop/s were measured on a single core already. With double precision the results on the compute architecture are similar, therefore only the single precision results are shown here.

4.4 Experience with the Convey HC-1

The Linux environment and documentation allows to do first steps rapidly. A modified Open64 compiler allows to specify personalities to use during the code compilation. During the compilation the proper personality must be selected. Plenty of documentation exists for the developer. It discusses modifications necessary to port code to the convey. A very useful example is the Convey Programmers Guide. Due to the coherent memory between FPGA and host an incremental parallelization is possible. Reserving memory on the proper device or migrating frequently accessed data is required to achieve good performance. Memory allocation on the co-processor and migration were both tested, and are easy to use.

Without proper data location the observed performance was about 4 Mflop/s, which is less than 1/1000 of observable performance. During the experiments sometimes the compiler did not vectorize loops. However, as the compiler reports statistics about vectorized loops, those could be localized. For a loop the compiler shows the number of all operation types (binary, logical, ...) which are in the loop. Running loops on the co-processor which are not vectorized causes a dramatic performance loss, which is similar to the case where data is allocated on the wrong device.

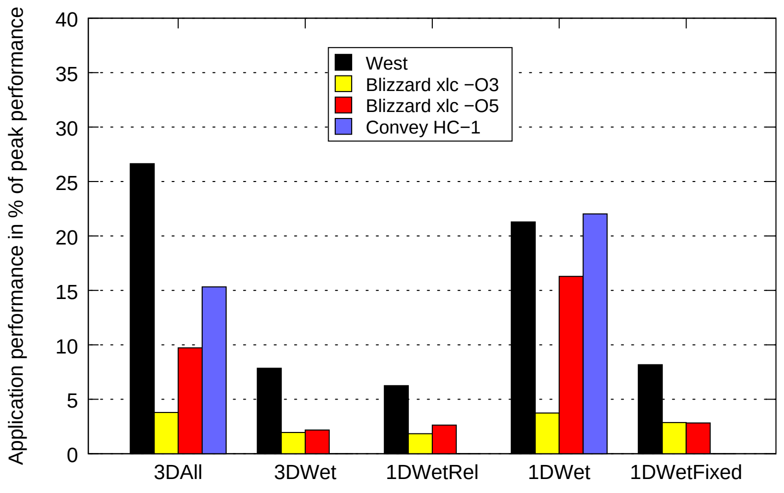

Figure 4.7: Efficiency of the index variants for a climate code.

Porting Fortran code to the Convey is easier than porting C, because by default Fortran has more strict aliasing rules and avoids pointers. Therefore, the compiler can make more assumptions about data usage.

During the evaluation we contacted Convey about some minor issues and a compiler bug which prohibited vectorization on one example code. The answer was prompt and accurate, a few days later the compiler bug was fixed by their compiler team.

There are several restrictions to co-processor code. For example, no I/O or system call can be made on the co-processor, and no function can be called which is not compiled for the co-processor. In our examples in Fortran we dynamically allocated the matrix. Unfortunately, the compiler could not vectorize the loops. Refactoring the code by extracting the loops into a separate function for the co-processor fixed this issue.

Several environment flags help to assess usage of the FPGA. We extensively used a flag which prevents code to run if the co-processor cannot be used. Another flag counts the number of calls to the co-processor.

The Convey Open64 compiler is a bit behind the current compilers. For example, the file extension `F03` is not detected.

5 Conclusions

In this first report we could get a first glance at the Convey system and we could evaluate some low-level performance aspects on our systems. Compilers and code often have plenty of room for optimization, which is especially true for the C compiler on Blizzard. The Westmere architecture is very efficient for the given problems. Due to the problem statement vector instructions could not be used. However, there are approaches to rewrite stencil operations accordingly to improve performance. There is the need to assess performance furhter, even by inspecting assembler code and assessing hardware capabilities for the given case.

The Convey HC-1 has the potential to be used for general purpose computation. Even though the system is still new, the documentation and provided software look rather mature. Even though the provided compiler is not the newest version, it works well. However, on Convey 22 Gflop/s were measured on the FPGA. If one compares this to the 4 Gflop/s on a single core of a commodity CPU the high initial costs of the new solution probably can not persuade common application developers.

Development on GPUs is now common practice and is applicable for many problems. For example the Testla C2070 achieves up to 630 Gflop/s in double precision. Memory is limited to 6 GByte. The price of the GPU accelerator is about 3,500$. In the related work we cited work which achieved 36 Gflop/s with the older generation of GPUs. Compared to the Convey HC-1 GPU programming does not allow the easy incremental parallelization of the Convey, data localization must be handled manually. A kernel on a GPU looks different from the sequential code, on the Convey system the code is identical and can still be compiled on other compilers. Portland Group's GPU compiler might help to utilize the GPU to some extent. However, deeper understanding of the hardware and software is necessary to exploit the hardware's full potential.

With the coming Knight family – which reaches approximately 512 Gflop/s – from Intel the easy CPU programming model can be applied directly to accelerators. Unfortunately, to utilize this approach completely converting code to vector operations is mandatory as the system will provide full performance only on 512 bit vectors. However, as it will require a shorter training period to gain some performance out of the box, this seems to be the way to go for large codes which must not run at full performance – for instance, because they are not run 24/7.

For long-running climate codes, that is, a few codes which burn half the CPU hours per year performance gains are vital. Unfortunately, the true strength of re-programming the FPGA – to gear the co-processor towards the application – could not be evaluated by us. Related work in this area shows that an stencil kernel could be implemented efficiently, although recent GPUs deliver a better performance. As all high performance implementations require a deep understanding of the software kernel and hardware characteristics, hand-optimized code is faster.

5.1 Future Steps

The main considerations for us are to approximate the potential performance of an equation – or algorithm – before they run on a given architecture. A theoretical performance assessment allows to estimate best performance a compiler could deliver on a given system. We will work in this direction to tighten performance bounds on our architectures.

To proof the usefulness of the Convey approach to existing codes it seems mandatory that performance can be estimated on the system before a new personality is written. In this context the estimation of how much space particular executions require on the FPGA and of how many cycles are needed to perform the operation can guide the direction. Let us assume, for instance, that in the best case 50 double precision adders could be implemented on each Xilinx FPGA. Each could perform one flop/s, which means that 200 Gflop/s could be achieved. To compute this bound one could consider memory limitations, and the size and number of lookup-tables, flip-flops and provided higher level execution units.

HMK Supercomputing promised to provide estimates for potential implementations to avoid doing time consuming research and implementations if the potential does not suffice for the code. By providing some rough space estimates for execution units this would enable us to check existing code for routines which could be implemented efficiently on the Convey HC-1. Then, a closer inspection of the code blocks by Convey should tighten these bounds. Finally, this will allow us to determine the benefit to port these applications to the Convey HC-1.

Acknowledgements

Thanks to Professor Heuvelin and Dr. Michael Neaga from the Karlsruhe Institute of Technology (KIT) who provided us CPU time on their Convey HC-1. Thanks to Michael Kuhn und Timo Minartz for reviewing this report.

Bibliography

[1] Roman Amorim, Gundolf Haase, Manfred Liebmann, and Rodrigo Weber dos Santos. Comparing CUDA and OpenGL implementations for a Jacobi iteration, 2008.

[2] Mauricio Araya-Polo, Javier Cabezas, Mauricio Hanzich, Miquel Pericas, Felix Rubio, Isaac Gelado, Muhammad Shafiq, Enric Morancho, Nacho Navarro, Eduard Ayguade, Jose Maria Cela, and Mateo Valero. Assessing Accelerator-based HPC Reverse Time Migration. *IEEE Transactions on Parallel and Distributed Systems*, 99(PrePrints), 2010.

[3] Jose Maria Cecilia, Jose Manuel Garcia, and Manuel Ujaldon. CUDA 2D Stencil Computations for the Jacobi Method. In *Para 2010: State of the Art in Scientific and Parallel Computing*, 2010.

[4] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil Computation Optimization and Auto-tuning on State-of-the-Art Multicore Architectures. In *In Proc. SC2008: High performance computing, networking, and storage conference*, 2008.

[5] S. Kamil, Cy Chan, L. Oliker, and J. Williams Shalf. An Auto-Tuning Framework for Parallel Multicore Stencil Computations. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2010.

[6] Shoaib Kamil, Kaushik Datta, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Implicit and explicit optimizations for stencil computations. In *In MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 51–60. ACM Press, 2006.

[7] Won ki Jeong and Ross T. Whitaker. A fast iterative method for a class of hamilton-jacobi equations on parallel systems. Technical report, University of Utah School of Computing Technical, 2007.

[8] H. Q. Le, Willikam J. Starke, J. Fields, F. O'Connell, D. Nguyen, B. Ronchetti, W. Sauer, E. Schwarz, and M. Vaden. IBM POWER6 microarchitecture, 2007.

[9] Liu Peng, Richard Seymour, Ken ichi Nomura, Rajiv K. Kalia, Aiichiro Nakano, Priya Vashishta, Alexander Loddoch, Michael Netzband, William R. Volz, and Chap C. Wong. High-order stencil computations on multicore clusters. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–11, Washington, DC, USA, 2009. IEEE Computer Society.

[10] Mathias Pütz. POWER6 Overview for Application Developers, 2008.

[11] M. Shafiq, M. Pericas, R. de la Cruz, M. Araya-Polo, N. Navarro, and E. Ayguade. Exploiting memory customization in FPGA for 3D stencil computations. In *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, pages 38–45. IEEE, 2009.

[12] Michael E. Thomadakis. The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms. `sc.tamu.edu/systems/eos/nehalem.pdf`, 2010.

[13] Sundaresan Venkatasubramanian and Richard W. Vuduc. Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 244–255, New York, NY, USA, 2009. ACM.