Spectral Element Simulations on the NEC SX-Aurora TSUBASA

Niclas Jansson
KTH Royal Institute of Technology
PDC Center for High Performance Computing
Stockholm, Sweden
njansson@kth.se

ABSTRACT

Following the recent transition in the high performance computing landscape to more heterogeneous architectures, application developers are faced with the challenge of ensuring good performance across a diverse set of platforms. In this paper, we present our work on porting the spectral element code Nek5000 to the recent vector architecture SX-Aurora TSUBASA. Using Nek5000's mini-app Nekbone, we formulate suitable loop transformations in key kernels, allowing for better vectorization, increasing the baseline performance by a factor of six. Using the new transformations, we demonstrate that the main compute intensive matrix-vector and matrix-matrix multiplication kernels achieves close to half the peak performance of a SX-Aurora core. Our work also addresses the gather-scatter operations, a key kernel for efficient matrix-free spectral element formulation. We introduce a new implementation of Nek5000's gather-scatter library with mesh topology awareness for improved vectorization via exploitation of the SX-Aurora's hardware gather-scatter instructions, improving performance with up to 116%. A detailed description of the implementation is given together with a performance study, comparing both single node performance and strong scalability characteristics, running across multiple SX-Aurora cards.

CCS CONCEPTS

• **Mathematics of computing** → **Mathematical software performance**; • **Computing methodologies** → **Vector / streaming algorithms**.

KEYWORDS

Nek5000,spectral element, gather-scatter

1 INTRODUCTION

In recent years, we have seen a transition in the high performance computing landscape to more heterogenous architectures. The traditional homogenous scalar processing machines are replaced with heterogenous machines combining scalar processors with various kinds of accelerators. While offering high theoretical peak performance and high memory bandwidth, to efficiently exploit these systems complex programming models and large programming investments are necessary. This places a heavy burden on the application developers to ensure good performance across a diverse set of platforms, often only achieving a very small fraction of the theoretical peak performance.

The SX-Aurora TSUBASA is the recent vector computer from NEC. Unlike previous vector machines from NEC, TSUBASA comes in a PCI express card called a Vector Engine (VE), hosted in a standard x86 server Vector Host (VH). This heterogenous system offers similar performance benefits as other systems with accelerators such as high memory bandwidth and floating-point peak performance. However, from an application developer's perspective it offers two attractive advantages compared to a usual accelerator e.g. a GPU. A first benefit is due to the architecture. Being a vector processor with high memory bandwidth, applications often can achieve a high sustained performance [3, 11]. The second advantage are the different programming models supported. The VE can either be seen as an accelerator, where parts of an application can be offload from the VH, or run the entire application natively on the VE. Executing an application natively on the VE is similar to running on a traditional vector computer, removing all the complexities of heterogenous systems. Thus, if an application is vectorizable (data parallel), a developer can focus on tuning the numerics instead of optimizing memory transfers between host and accelerator.

SX-Aurora has proven to perform well in various benchmark suites, achieving comparable results to recent GPUs [12, 13]. It has also successfully been utilized in numerical solvers, e.g. in sparse matrix solvers [18], and also in entire applications achieving high sustained performance [10, 17]. These motivating results together with a productive programming model (native execution) makes SX-Aurora an interesting alternative to GPUs for accelerating real applications on contemporary heterogenous systems.

In this paper, we present our fist experience of porting and tuning the spectral element based computational fluid dynamics code Nek5000 to the SX-Aurora TSUBASA. To our knowledge, we here report the first attempt to port the code to the Vector Engine. We follow a similar approach as when the code was ported to GPUs [16], and use the mini-app Nekbone for our initial porting and tuning of key kernels. In experiments, we find that certain loop transformations result in significant improvements, while other kernels needed to be completely rewritten to perform well on the SX-Aurora. The tuned versions are compared against both CPU and GPU implementations, and strong scalability characteristics, when running across multiple Vector Engines, are investigated.

The outline of the paper is the following; In section 2 a background is given on the spectral element code and the mini-app Nekbone. Our experimental setup is presented and a performance baseline is established in Section 3. The porting and tuning of Nekbone is described in detail in Section 4 and 5 respectively. A performance study is given in Section 6, comparing our tuned version against Nekbone running on various architectures, and in Section 7 we summarize our results and outline future work.

2 NEK5000

Nek5000 is an open-source computational fluid dynamics code based on the spectral element method (SEM) [20]. Nek5000 solves

the incompressible Navier-Stokes equations, together with a number of additional physics (heat transfer, magneto-hydrodynamics, low Mach number, electrostatics) on general hexahedral spectral elements. Special focus is laid on single-core efficiency and large-scale parallel scalability. Nek5000 has a long history as a scalable solver, it won the Gordon Bell Prize in 1999 [21] and has demonstrated scalability on more than one million MPI ranks [7]. Nekbone is Nek5000's mini-app, it closely resembles the basic structure of the full code and is often used to evaluate the performance on new architectures.

2.1 Nekbone

Nekbone solves Poisson's equation with homogenous Dirichlet boundary conditions,

$$-\nabla^2 u = f \quad \text{in } \Omega, \tag{1}$$

$$u = 0 \quad \text{on } \partial\Omega. \tag{2}$$

The spectral element solution is based on the weak formulation, namely find $u \in V \subset H_0^1$ such that,

$$\int_\Omega \nabla u \nabla v d\Omega = \int_\Omega f v d\Omega \quad \forall v \in V. \tag{3}$$

We discreteize Ω into a set of E non-overlapping hexahedral elements $\Omega = \cup_{e=1}^{E} \Omega^e$ and define a piecewise polynomial approximation space V^N, with a tensor-product polynomial basis using one dimensional N-th order Legendre-Lagrange polynomials $l_i(\xi)$,

$$l_i(\xi) = \frac{N(N+1)^{-1}(1-\xi^2)L'_N(\xi_i)}{(\xi-\xi_i)L_N(\xi_i)} \quad \text{for } \xi \in [-1,1] \tag{4}$$

with Gauss-Lobatto-Legendre (GLL) quadrature points ξ_i, and N-th order Legendre polynomials L_N. The discrete solution u can then be expressed as a tensor product of the polynomials on the reference element,

$$u^e(\xi,\eta,\gamma) = \sum_{i,j,k}^{N} u^e_{ijk} l_i(\xi) l_j(\eta) l_k(\gamma), \tag{5}$$

where $\xi, \eta, \gamma \in [-1,1]$ are the coordinates of the reference element. Applying this formulation, the discrete, bilinear form $a(u,v)$ of (3) can be expressed as,

$$a(u,v) = \sum_{e=1}^{E} (v^e)^T \mathbf{D}^T \mathbf{G}^e \mathbf{D} u^e = \sum_{e=1}^{E} (v^e)^T A^e u^e, \tag{6}$$

where $\mathbf{G}^e$ is the tensor comprising of the geometric factors for mapping to and from the reference element and $\mathbf{D}$ the local derivatives of the operand at the GLL points.

For high-order methods, assembling either the local element matrix A^e or the full stiffness matrix A is prohibitively expensive. Therefore, a key to achieve good performance in SEM methods is to consider a matrix free formulation, where one always works with the unassembled matrix $A_L = \text{diag}\{A^1, A^2, \ldots, A^E\}$. Each degree of freedom in the discrete solution u is assigned a unique global number. To ensure continuity of functions on the element level u^e_{ijk}, we define a boolean gather matrix Q^T, mapping from local to global unique numbers in u. A corresponding scatter matrix is given by Q such that $u_L = \{u^e\}_{e=1}^{E}$. With the gather-scatter operations the discrete bilinear form (6) becomes

$$a(u,v) = \sum_{e=1}^{E} (v^e)^T A^e u^e = (\mathbf{Q}v)^T A_L \mathbf{Q} u = v^T A u. \tag{7}$$

As described in [1], Q or Q^T is never formed explicitly, only the action QQ^T is used as a single gather-scatter operation. For example, the matrix-vector product $w = Au$, becomes $w_L = QQ^T A_L u_L$ in the matrix-free formulation. In practice, the gather-scatter operation QQ^T is implemented using a local to global mapping (*loctglb*), translating local indices u^e_{ijk} to global indices u_i as illustrated in Algorithm 1.

The linear form $L(v)$ is formulated in a similar way and the resulting linear system is solved for by a preconditioned conjugate gradient solver, which in the matrix free form takes the form as described in Algorithm 2, where z_L, r_L, p_L, w_L, x_L are vectors with element-wise data e.g. $x_L = \{x^e\}_{e=1}^{E}$, $\mathcal{M}_L$ is a diagonal mask matrix to satisfy the boundary conditions and C_L a weight to correctly compute the inner products of shared entities in the local formulation. For a more detailed description of the spectral element formulation used in Nekbone please refer to e.g [8].

Algorithm 1 The gather-scatter kernels QQ^T.

Function $u = Q^T u_L$	**Function** $u_L = Qu$
for $e = 1, 2 \ldots, E$ **do**	**for** $e = 1, 2 \ldots, E$ **do**
for $i, j, k = 1, 2 \ldots, n$ **do**	**for** $i, j, k = 1, 2 \ldots, n$ **do**
$\hat{\imath} \leftarrow loctglb(i, j, k, e)$	$\hat{\imath} \leftarrow loctglb(i, j, k, e)$
$u_{\hat{\imath}} \leftarrow u_{\hat{\imath}} + u^e_{ijk}$	$u^e_{ijk} \leftarrow u_{\hat{\imath}}$
end for	**end for**
end for	**end for**

Algorithm 2 Matrix-free preconditioned conjugate gradient solver.

for $k = 1, 2, \ldots$ **do**
  $z_L \leftarrow M^{-1} r_L$
  $\rho_2 \leftarrow \gamma_1$
  $\rho_1 \leftarrow r_L^T C_L z_L$
  $\beta \leftarrow \frac{\rho_1}{\rho_2}$
  if $k = 1$ **then**
    $\beta \leftarrow 0$
  end if
  $p_L \leftarrow \beta p_L + z_L$
  $w_L \leftarrow \mathcal{M}_L Q Q^T A_L p_L$
  $\gamma \leftarrow w_L^T C_L p_L$
  $\alpha = \frac{\rho_1}{\gamma}$
  $x_L \leftarrow x_L + \alpha p_L$
  $r_L \leftarrow r_L - \alpha w_L$
  $\epsilon \leftarrow \sqrt{r_L^T C_L r_L}$
end for

2.2 Related work

Over the years, Nekbone has been used extensively as a testbed for code porting and tuning, in particular during the initial GPU port

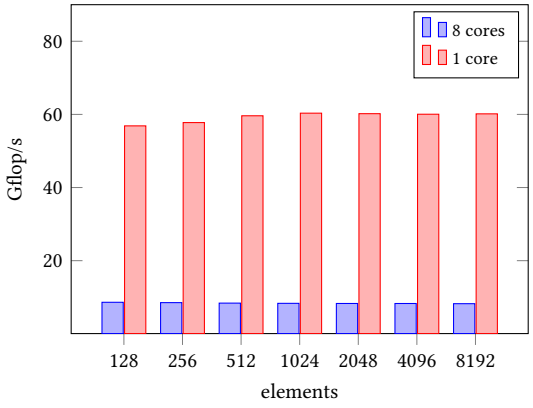

Figure 1: Baseline performance of Nekbone.

of Nek5000 [8, 16]. From the GPU work we know that the matrix-free spectral element formulation can perform well on accelerators, and the results serve as a performance baseline for this work. Furthermore, the high-order discontinuous-Galerkin solver Ateles was recently successfully ported to the SX-Aurora [10]. Since Ateles numerical methods share several characteristics with Nekbone's, it demonstrate that spectral elements should be suitable to run well on the Vector Engine.

3 EXPERIMENTAL SETUP AND PLATFORM

For the experiments in this paper we use one of the vector nodes in the heterogenous compute cluster Vulcan at HLRS. Each of the vector nodes is equipped with eight 1.4 GHz Vector Engines, with 48GB of HBM2 memory and a double precision peak performance of 2.15 Tflop/s per card. Each VE is equipped with eight cores, each with a double precision peak performance of 268.8 Gflop/s. The NEC Fortran compiler version 3.0.1 is used with optimization flags `-O4 -finline-functions`, and all tests are run in the native execution mode on the VE.

In all experiments Nekbone is compiled to use ninth order polynomials with ten quadrature points in each dimensions ($n_x = n_y = n_z = 10$). The conjugate gradient solver is fixed to 100 iterations, with the simplest preconditioner settings, corresponding to Nekbone's `example2`.

3.1 Baseline performance

To establish a performance baseline we ran Nekbone, unmodified from the repository [5], on one and eight VE cores respectively for a set of elements $E = 128, \ldots, 8192$ and measured the total flop/s.

In Figure 1 we present the baseline performance for both a single and eight VE cores. A first observation is that the baseline performance is very low. Both the single and multicore results are far away from the peak performance of a single core.

The poor performance is not due to inadequate vectorization. On the contrary, by studying the compiler listing it is clear that a majority of the code vectorizes well. However, profiling data from the experiments reveals that the vector length in each vectorized section of the code is too short. For each conjugate gradient iteration the work is split between the inner products and the matrix-vector multiplication (line 10 in Algorithm 2), which from here on we refer to as Ax. Since we can not improve the vectoriable length of the inner products, the first kernel to investigate is Ax.

Listing 1: The local matrix-vector product kernel $A^e u^e$.

```
call local_grad3(ur,us,ut,u,n,dxm1,dxtm1)

do i=1,nxyz
   wr = g(1,i)*ur(i) + g(2,i)*us(i) + g(3,i)*ut(i)
   ws = g(2,i)*ur(i) + g(4,i)*us(i) + g(5,i)*ut(i)
   wt = g(3,i)*ur(i) + g(5,i)*us(i) + g(6,i)*ut(i)
   ur(i) = wr
   us(i) = ws
   ut(i) = wt
enddo

call local_grad3_t(w,ur,us,ut,n,dxm1,dxtm1,wk)
```

In the standard Nekbone implementation, Ax is computed via a loop over each element, computing the local matrix-vector product kernel outlined in Listing 1. Here `ur,us,ut` are work arrays of size $n_x \times n_y \times n_z$, the number of GLL points in each direction, `g` the geometric factors $\mathbf{G}$ and `dxm1, dxtm1` are square matrices of size $n_x \times n_x$ (Nekbone assumes cubic shaped elements) corresponding to $\mathbf{D}$ and $\mathbf{D}^T$. The routines `local_grad3` and `local_grad3_t` compute the small matrix-matrix multiplication to form $\mathbf{D}u^e$ and $\mathbf{D}^T v^e$, each of a small size $\approx n_x \times n_x^2$. Even for a very high polynomial order of nine, these operations have an average vector length of less than 25, which is too short to make proper use of the SX-Aurora's vector units.

4 IMPROVING THE Ax KERNEL

To improve the performance of the matrix-vector product, we implemented a small benchmark that only contains the Ax kernel on line 10 in Algorithm 2. Since our main target is to establish a feasible performance for Ax, we also removed the communication parts QQ^T as well as the masking operator $\mathcal{M}_L$, leaving us with a benchmark problem similar to the CEED Bake-off Kernel BK5 [14]. Since the OpenACC GPU port of Nekbone faced similar problems, with too little data per element, our starting point was the slightly modified Ax formulation in [8], shown in Listing 2. In the modified formulation the small, per element matrix-matrix products performed in `local_grad3` (Listing 1) were merged into one loop for all elements. To further increase the work per iteration, it also contains the loop multiplying with the geometric factors $\mathbf{G}$. The second loop in the modified formulation is similar, with the loops for $\mathbf{D}^T v^e$ merged into a larger loop over all the elements (without geometric factors).

Despite the increased work per iteration, the OpenACC modified Ax formulation did not perform well on the Vector Engine due to the different access patterns for `ur`, `us` and `ut`. In order to increase locality, we divide up the three different computations in separate loops. Furthermore, since the geometric factor depend on the result from all three variables, this operation was moved to a separate loop. The arrays `ur`, `us` and `ut` are extended to hold the result of each $\mathbf{D}u^e$ evaluation, which are multiplied with the geometric factor in a separate loop afterwards.

The loops for computing $\mathbf{D}u^e$ can further be classified into a standard matrix-matrix product (`ur`) and strided matrix-matrix products

Listing 2: The *Ax* formulation from the OpenACC port.

```
do e = 1, nelt
   do k,j,i = 1, n ! tripel loop nest
      ur = 0
      us = 0
      ut = 0
      do l = 1, n
         ur = ur + dxm1(i,l)*u(l,j,k,e)
         us = us + dxm1(j,l)*u(i,l,k,e)
         ut = ut + dxm1(k,l)*u(i,j,l,e)
      end do
      wr(i,j,k,e) = g(i,j,k,1,e)*ur
                  + g(i,j,k,2,e)*us
                  + g(i,j,k,3,e)*ut
      ws(i,j,k,e) = g(i,j,k,2,e)*ur
                  + g(i,j,k,4,e)*us
                  + g(i,j,k,5,e)*ut
      wt(i,j,k,e) = g(i,j,k,3,e)*ur
                  + g(i,j,k,5,e)*us
                  + g(i,j,k,6,e)*ut
   end do ! end tripel loop nest k,j,i
end do

do e = 1, nelt
   do k,j,i = 1, n ! tripel loop nest k,j,i
      do l = 1, n
         w(i,j,k,e) = w(i,j,k,e)
                    + dxtm1(i,l)*wr(l,j,k,e)
                    + dxtm1(j,l)*ws(i,l,k,e)
                    + dxtm1(k,l)*wt(i,j,l,e)
      end do
   end do ! end tripel loop nest k,j,i
end do
```

Listing 3: Loop transformations of *Ax* for SX-Aurora.

```
do i = 1, n
   do j = 1, n*n*nelt
      wr = 0d0
      do l = 1, n
         wr = wr + dxm1(i,l)*u(l,j,1,1)
      end do
      ur(i,j,1,1) = wr
   end do
end do

do k = 1, n
   do i = 1, n
      do j = 1, n
         do e = 1, nelt
            ws = 0d0
            !NEC$ unroll_completely
            do l = 1, n
               ws = ws + dxm1(j,l)*u(i,l,k,e)
            end do
            us(i,j,k,e) = ws
         end do
      end do
   end do
end do

do j = 1, n
   do i = 1, n
      do k = 1, n
         do e = 1, nelt
            wt = 0d0
            !NEC$ unroll_completely
            do l = 1, n
               wt = wt + dxm1(k,l)*u(i,j,l,e)
            end do
            ut(i,j,k,e) = wt
         end do
      end do
   end do
end do
```

(us and ut). For the first type, we can collapse the three last dimensions of u into one loop resulting in a matrix-matrix product kernel that will be identified by the NEC compiler and replaced with a library call. For the strided products, we want to maximize the vector length per iteration. Thus, we move the element loop inwards, before the term by term product summation loop. To further increase performance, the i,j,k loops are permuted for each case to exploit locality, and the inner most loop is also unrolled completely, if the iteration count is know at compile time. In Listing 3 we present the loops for computing $\mathbf{D}u^e$, where computing $\mathbf{D}^T v^e$ is identical. The geometrical terms are added via a straightforward multiply and add loop with the ur,us and ut arrays, similar to what was performed on a per element basis in Listing 1.

To assess the performance of the tuned *Ax* kernel, we ran our small benchmark on a single Vector Engine for a set of elements $E = 128, \ldots, 8192$. In Figure 2, we present the total Gflop/s when running the benchmark on $1, \ldots, 8$ cores. Compared to the performance baseline, the *Ax* kernel with rearranged loops improved the baseline performance by a factor of six, achieving a maximum of 113.2 Gflop/s, more than 40% of the theoretical peak performance of a single VE core. However, once the problem size increases, the single core performance also starts to decrease. Using more cores increases performance, reaching a peak of 408.9 GFlop/s when using all eight cores, corresponding to more than 19% of the theoretical peak performance of the Vector Engine.

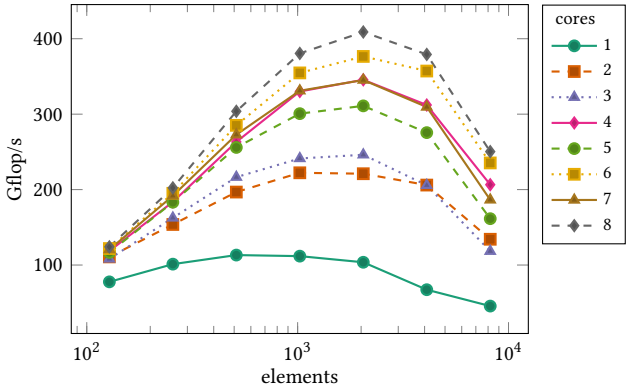

Figure 2: Performance of the *Ax* benchmark problem on a single VE for various numbers of elements and cores.

4.1 Using the tuned *Ax* kernel in Nekbone

The refactored *Ax* loops were incorporated into Nekbone and the performance was evaluated on a single VE for a set of elements $E = 128, \ldots, 8192$, measuring the total flop/s. In Figure 3, we present the performance of the tuned Nekbone using one and eight cores respectively. Both single and multicore performance has increased significantly compared to the baseline. However, a large fraction of the gained performance of the tuned *Ax* kernels is lost. For single core runs, Nekbone reached $\approx$ 25 Gflop/s compare to 113.2 Gflop/s for *Ax*, and when using all cores, performance drops from 408.9

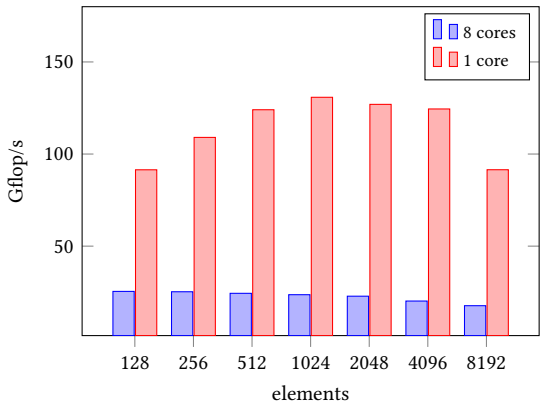

Figure 3: Performance of Nekbone using the tuned Ax kernel on a single VE using one and eight cores respectively.

Gflop/s in Ax to $\approx$ 127 Gflop/s when running the full Nekbone. Inspecting Algorithm 2, we see that besides the Ax operation in line 10, the rest of Nekbone roughly consists of vector operations, reductions and the gather-scatter operations QQ^T. When profiling Nekbone, neither the vector operations (inner products) nor reductions show as costly operations. The gather-scatter operation however accounts for a large fraction of the entire runtime, comparable to Ax itself. Furthermore, the profiling data reveals that the gather-scatter does not vectorize at all, thus causing the large execution time.

5 OPTIMIZED GATHER-SCATTER

In Nekbone, the gather-scatter operation is handled by a C library, gslib [6, 15]. The library is written as a generic sparse communication kernel, capable of performing various gather-scatter operations e.g. addition or multiplication as a black-box, only requiring a list of shared (global) ids and a local to global mapping (similar to *loctglb* in Algorithm 1) as input.

To efficiently carry out the gather-scatter operation, gslib groups local indices in u^e_{ijk} sharing the global id $\hat{\imath}$ in u_i, stored in a consecutive list terminated with -1 after each global id. The original CPU implementation then processes all local indices belonging to a global id until the terminator is found, resulting in short vector lengths. Furthermore, in gslib the implementation uses complex pointer arithmetic, which prevents vectorization by the NEC compiler. This is one of the major factor for the reduced performance comparing the tuned Ax kernel in Nekbone with the stand-alone Ax benchmark (Figure 3). The vectorization issues in gslib were also observed when porting Nek5000 to GPUs. When porting Nek5000's electromagnetics solver, the loops were refactored to removed the complex pointer arithmetic, and additional arrays were created to remove the necessary search for the terminator. Thus, loops were vectorizable and this refactored gather-scatter formulation gave significant performance improvements [19]. However, the additional arrays double the amount of indirect memory addressing per memory access making the refactorization less suitable for the SX-Aurora architecture.

Algorithm 3 New overlapped gather-scatter kernel.

1: $S \leftarrow \emptyset$
2: **for** $i = 1, 2, \ldots, m$ **do**
3: Post non-blocking receive on $buf(i)$
4: $S \leftarrow S \cup i$
5: **end for**
6: $v \leftarrow gather(shared_{id}, u_L)$
7: Post non-blocking sends of v
8: $v \leftarrow gather(local_{id}, u_L)$
9: $w_L \leftarrow scatter(local_{id}, v)$
10: **while** $S \neq \emptyset$ **do**
11: **for all** $j \in S$ **do**
12: **if** non-blocking receive j has completed **then**
13: $v \leftarrow gather(shared_{id}, buf(j))$
14: $S \leftarrow S \setminus j$
15: **end if**
16: **end for**
17: **end while**
18: $w_L \leftarrow scatter(shared_{id}, v)$

One of the objectives in the Horizon 2020 project EPiGRAM-HS [4] is to carry out a refactoring of Nek5000 to modern Fortran, and to enable the code for use on large-scale heterogenous systems. Of particular interest for the current work is the refactored version's newly developed gather-scatter kernel. Implemented directly in Fortran, the new kernel offers more vectorizable loop constructs. Therefore, to improve our gather-scatter performance further, we modified Nekbone to be based on the refactored version of Nek5000 and focused our tuning effort on the new Fortran based gather-scatter kernel.

Since the new kernel is implemented directly in the spectral element code, the gather-scatter kernel can utilize information about the underlying computational mesh. Using the mesh topology, gather-scatter operations are scheduled to allow for overlapping communication with computation. Elements are classified as *local*, if all shared global ids u_i are owned by the same VE core. If one or more ids are shared with another core, the element is classified as *shared*. Based on the element classification, each global id is then stored in a corresponding list of local and shared ids. The new gather-scatter formulation can then be expressed as in Algorithm 3 performing $w_L = QQ^T u_L$. First, all non-blocking receives (`MPI_Irecv`) for shared ids are posted. The shared ids are gathered into a buffer and transmitted to neighbors sharing the same ids using non-blocking send (`MPI_Isend`) operations. During the non-blocking communication, the local ids are gathered from u_L into a buffer v and scattered into the corresponding places in the output vector w_L. Once the local operation has completed, a loop polls each of the posted non-blocking receives until all has completed. During the loop, once data has been received, it is directly gathered into the buffer v. Finally, with all receives completed, the shared ids are scattered back into the output vector w_L.

In Algorithm 3, m is the number of neighbors to both receive and send data to/from and v is a local buffer for each shared global id in u_i. Since u_i is too large to be represented on each core, a mapping function dg, maps the shared id $\hat{\imath}$ to index j in v. Similarly, a function

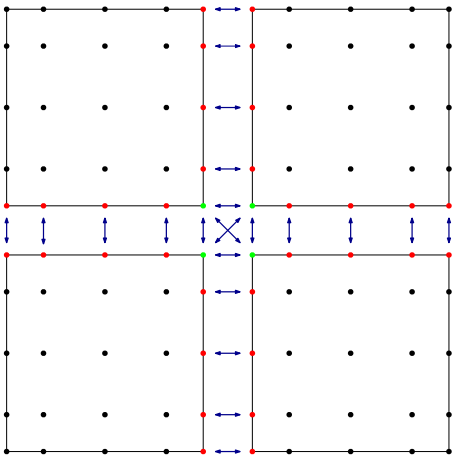

Figure 4: An illustration of the different types of shared ids, edge and corner, on a two dimensional mesh with four elements, each with five GLL points in each direction.

gd maps a shared $u_L(i)$ to $v(j)$. Let k be the number of (locally) shared ids, and dg and gd be the mapping functions, the gather calls in Algorithm 3 can then be expressed as,

for $i = 1, 2, \ldots, k$ **do**
  $v(dg(i)) \leftarrow v(dg(i)) + u_L(gd(i))$
end for

However, this formulation of the gather procedure causes certain issues for the compiler, preventing proper vectorization on the Vector Engine. The main problem comes from the mapping functions and the indirect addressing they are causing in u_L and v. Also, since the entries in the maps are related to local element indices to shared global ids, most of the entries are not injective (except for ids on the external boundary or between mesh partitions), using iteration independent pragmas e.g. `!NEC$ ivdep` will not work.

Starting with the map $gd(i)$, since this is only used to load entries from u_L, we can introduce a temporary vector w, and hoist the loading of u_L to a preceding loop executing

for $i = 1, 2, \ldots, k$ **do**
  $w(i) \leftarrow u_L(gd(i))$
end for

This loop can now be vectorized, and it will also be identified as a gather operation by the NEC compiler and use the optimized machine instructions for the memory operations.

For the second map $dg(i)$, we need to find injective structures to break the loop carried dependencies. Given the mesh topology, we can classify the type of a shared global id u_i. In Figure 4, we illustrate the two types of ids in a two dimensional mesh, edge and corner. An id on the edge can only have one neighbor while the corner can have arbitrary number of neighbors sharing the same id. In three dimensions, we have three types: corner, edge and faces. both the edge and corner can now have arbitrary number of neighbors, while faces only share the id with one neighbor. Assuming we are running on a single core, we now sort the maps such that all ids with an arbitrary number of neighbors are listed first, followed by the single neighbor ids. The first part of the reordered map contains

Algorithm 4 Optimized gather routine.

1: **for** $i = 1, 2, \ldots, k'$ **do**
2: $w(i) \leftarrow u_L(gd(i))$
3: **end for**
4: **for** $i = 1, 2, \ldots, k'$ **do**
5: $v(dg(i)) \leftarrow v(dg(i)) + w(i)$
6: **end for**
7: **for** $i = 1, 3, \ldots, k''$ **do**
8: $tmp \leftarrow u_L(gd(i)) + u_L(gd(i+1))$
9: $v(dg(i)) \leftarrow tmp$
10: $v(dg(i+1)) \leftarrow tmp$
11: **end for**

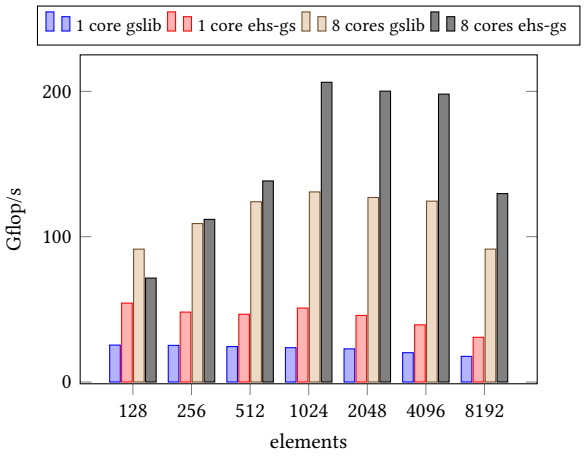

Figure 5: Performance of the tuned Nekbone using two different gather-scatter kernels, standard (gslib) and tuned (ehs-gs), evaluated on a single VE using one and eight cores respectively.

the non injective ids that we can not vectorize. In the second part we have a list of tuples (i, j) corresponding to indices in u_L sharing the same shared id $\hat{\imath}$. Thus, each tuple is independent and a loop over the tuples would be vectorizable. Let k be the total number of shared ids, k' the number of non-injective ids and k'' the number of injective ids. The local gather operations in Algorithm 3 can then be expressed as in Algorithm 4.

In the new gather formulation, the first loop will be vectorized with identified gather memory operations as discussed above. Unfortunately, the second loop can still not be vectorized, but by using the pre gathered w we have at least reduced the number of slow memory loads. Finally, if we use the `!NEC$ ivdep` pragma for the last loop it will be vectorized and the compiler will identify a gather memory operation on *tmp* and scatter memory operation on *v()*. The shared gather operations in Algorithm 3 follows the same pattern, with the exception that the last loop has unit stride, only performing an injective gather-scatter of $u_L(gd(i))$ into $v(dg(i))$. Since there is still a non vectorizable loop in the new formulation, it will perform better than Algorithm 3 only if $k'' \gg k'$.

5.1 Using the new gather-scatter in Nekbone

To evaluate the new gather-scatter kernel from EPiGRAM-HS, we ran Nekbone on a single VE for a set of elements $E = 128 \ldots 8192$, measuring total flop/s. In Figure 5, we present the performance of the new kernel against the standard gslib using one and eight cores respectively. On a single VE core, the new gather-scatter implementation is always faster than gslib, achieving up to more than twice the performance. This performance gain is purely due to the more vectorizable loops, see for example Algorithm 4. Multicore performance is in overall up to 1.5 faster than standard gslib. For the smaller test cases gslib is on par or even faster than the new implementation. In these cases, we have few elements per core, so less data to vectorize on, which causes overhead from the additional loops introduced in Algorithm 4 compared to standard gslib.

6 PERFORMANCE EVALUATION

We finally evaluated the performance of the SX-Aurora tuned Nekbone by comparing against the performance on different architectures. First we compared against CPUs on one full node on Beskow at PDC, a Cray XC40 with two 16 core Intel E5-2698v3 running at 2.3 GHz per node, equipped with 64GB of RAM. Secondly we ran the same case using GPUs on Piz Daint at CSCS, a Cray XC50 with one 12 core Intel E5-2690v3 running at 2.6GHz, equipped with 64GB of RAM and a NVIDIA Tesla P100 per node. On Beskow we used the standard CPU version of Nekbone [5] and the Intel compiler version 19.0.1, while on Piz Daint we use the OpenACC/CUDA version of Nekbone [8, 9] compiled using the PGI compiler version 19.7. Finally, a vector node in Vulcan at HLRS was used to run the SX-Aurora tuned Nekbone. The NEC Fortran compiler version 3.0.1 was used, and all tests ran on one of the 1.4GHz Vector Engines installed in each node.

In Figure 6, we present the performance (Gflop/s) for running Nekbone with $n_x = n_y = n_z = 10$ for a range of elements $E = 128, \ldots, 4096$ on the three different architectures. Similar to our previous experiments (Figure 2 and 5), SX-Aurora performance is strongly related to the amount of elements, with a clear peak before the performance stagnates and starts to decrease. Similar to what is observed in the CPU results when it starts to run out of cache space after 1024 elements. For the P100 the behavior is opposite, performance always increases with more elements.

In these experiments, Nekbone achieves a maximum of 9.6 % of the SX-Aurora's theoretical peak performance, 5.5% of the NVIDIA P100's peak performance and 17% of the two Intel E5-2698v3's performance (using the peak performance estimates for Haswell CPUs derived in [2]). However, the best CPU result is only achieved for one element configuration ($E = 1024$), while the GPU and Vector Engine achieved sustained performance for the three largest problem sizes.

6.1 Scalability

Since each vector node in Vulcan is equipped with eight SX-Aurora cards, we also ran a set of experiments to evaluate the strong scalability characteristics. Using the tuned Nekbone, we picked three different problem sizes 1024, 4096, 8192 and measured the performance, when running each problem on $1, \ldots, 8$ VEs respectively. Similarly to the single VE performance results, we see in Figure 7

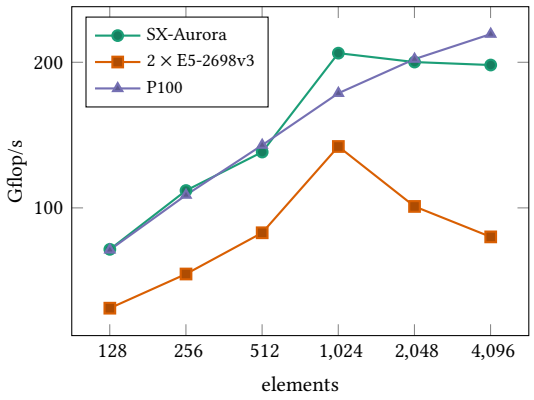

Figure 6: Performance comparison of Nekbone running on three different hardware architectures.

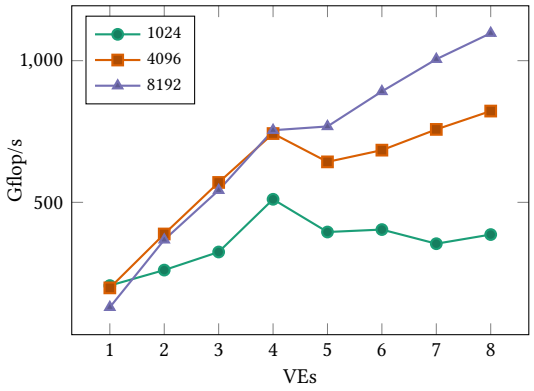

Figure 7: Performance results for Nekbone running across multiple Vector Engines.

that good scalability is only obtained, if there are enough elements assigned to each VE. For the two larger problem sizes the scalability is better, using 4096 elements near optimal scalability up to four VEs, and for 8192 we obtain a slightly super linear speedup up to four VEs due to our non-optimal base case.

Furthermore, for all three problem sizes there is a clearly visible performance degradation, when moving from using four to five vector engines. For 8192 elements, the sustained performance decreases from 8.8% (755 Gflop/s) using four VEs down to 7.1% (768 Gflop/s) using five VEs, and is reduced down to 6.9% (1097 Gflop/s), when all eight VEs are used. The vector nodes in Vulcan have two PCI express switches connecting the vector engines with the CPUs, allowing for direct MPI communication between the VEs. However, each PCI express switch can hold up to four VEs each. Thus, the additional jump between different PCI express switches is the most likely explanation for the performance drop when moving from four to five VEs.

7 SUMMARY AND FUTURE WORK

In this paper, we have presented our work on porting and optimizing spectral element simulations on the recent vector architecture SX-Aurora TSUBASA. Compared to contemporary heterogenous computing platforms, the SX-Aurora offers a more developer friendly programming model. The native execution mode allows an application developer to use the full potential of the SX-Aurora without having to deal with the complexities of heterogenous systems.

Using the mini-app Nekbone, we demonstrated that suitable loop transformations greatly improved vectorization and increased the performance by a factor of six. In key compute intensive kernels the transformations achieved 40% of the theoretical peak performance of a single SX-Aurora core, and for the entire Nekbone close to 10% of peak performance when using all eight cores. Our work also addressed the gather-scatter operations, a key kernel for efficient matrix-free spectral element formulations, albeit difficult to vectorize. Using a new gather-scatter formulation with mesh topology awareness, the operations could be partially vectorized, resulting in up to 1.5 faster performance. The performance of the tuned Nekbone was evaluated against both CPU and GPU implementation with convincing results.

The results for Nekbone presented in this paper have encouraged us to further develop and optimize the SX-Aurora port of the full spectral element code Nek5000. Our future work will focus on the gather-scatter kernel, which still is a major bottleneck despite the optimizations.

To conclude, our results demonstrate that spectral element simulations can be performed on recent vector architectures with favorable performance together with a developer friendly programming model. This makes the SX-Aurora TSUBASA a suitable platform for numerical simulations in today's heterogeneous computing landscape.

ACKNOWLEDGMENTS

This work was supported by the European Commission Horizon 2020 project grants "EXCELLERAT: The European Centre of Excellence for Engineering Applications" (grant reference 823691) and "EPiGRAM-HS: Exascale Programming Models for Heterogeneous Systems" (grant reference 801039). The experiments were performed on resources provided by Höchstleistungsrechenzentrum Stuttgart (HLRS) and PRACE Research Infrastructure resource Piz Daint hosted by CSCS, Switzerland and the Swedish National Infrastructure for Computing (SNIC) at PDC Center for High Performance Computing.

REFERENCES

[1] M. O. Deville, P. F. Fischer, and E. H. Mund. 2002. *High-Order Methods for Incompressible Fluid Flow.* Cambridge University Press, Cambridge.
[2] Romain Dolbeau. 2018. Theoretical peak FLOPS per instruction set: a tutorial. *The Journal of Supercomputing* 74, 3 (2018), 1341–1377.
[3] Ryusuke Egawa, Kazuhiko Komatsu, Shintaro Momose, Yoko Isobe, Akihiro Musa, Hiroyuki Takizawa, and Hiroaki Kobayashi. 2017. Potential of a modern vector supercomputer for practical applications: performance evaluation of SX-ACE. *The Journal of Supercomputing* 73, 9 (2017), 3948–3976.
[4] EPiGRAM-HS: Exascale Programming Models for Heterogeneous Systems. 2018. https://epigram-hs.eu/.
[5] Paul Fischer and Katherine Heisey. 2013. NEKBONE: The Thermal Hydraulics mini-application. https://github.com/Nek5000/Nekbone.
[6] P Fischer, J Lottes, D Pointer, and A Siegel. 2008. Petascale algorithms for reactor hydrodynamics. *Journal of Physics: Conference Series* 125 (jul 2008), 012076.
[7] Paul F. Fischer. 2015. Scaling Limits for PDE-Based Simulation (Invited). In *22nd AIAA Computational Fluid Dynamics Conference.*
[8] Jing Gong, Stefano Markidis, Erwin Laure, Matthew Otten, Paul Fischer, and Misun Min. 2016. Nekbone performance on GPUs with OpenACC and CUDA Fortran implementations. *The Journal of Supercomputing* 72, 11 (2016), 4160–4180.
[9] Andreas Jocksch, Jing Gong, Niclas Jansson, Adam Peplinski, Alan Gray, and Philipp Schlatter. 2021. Porting Nek5000 on GPUs Using OpenACC and CUDA. Poster at PASC'20. In press.
[10] Harald Klimach and Sabine Roller. 2020. Using the NEC Aurora TSUBASA for High-Order Discontinuous-Galerkin in Ateles. In *Sustained Simulation Performance 2018 and 2019.* Springer International Publishing, Cham, 57–68.
[11] Kazuhiko Komatsu, Ryusuke Egawa, Yoko Isobe, Ryusei Ogata, Hiroyuki Takizawa, and Hiroaki Kobayashi. 2015. An approach to the highest efficiency of the HPCG benchmark on the SX-ACE supercomputer. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC15), Poster.*
[12] Kazuhiko Komatsu and Hiroaki Kobayashi. 2020. Performance Evaluation of SX-Aurora TSUBASA by Using Benchmark Programs. In *Sustained Simulation Performance 2018 and 2019.* Springer International Publishing, Cham, 69–77.
[13] K. Komatsu, S. Momose, Y. Isobe, O. Watanabe, A. Musa, M. Yokokawa, T. Aoyama, M. Sato, and H. Kobayashi. 2018. Performance Evaluation of a Vector Supercomputer SX-Aurora TSUBASA. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis.* 685–696.
[14] libCEED. 2020. development site. https://github.com/ceed/libceed.
[15] James W. Lottes et al. 2008. GSLIB. https://github.com/Nek5000/gslib.
[16] Stefano Markidis, Jing Gong, Michael Schliephake, Erwin Laure, Alistair Hart, David Henty, Katherine Heisey, and Paul Fischer. 2015. OpenACC acceleration of the Nek5000 spectral element code. *The International Journal of High Performance Computing Applications* 29, 3 (2015), 311–319.
[17] Akihiro Musa, Takashi Abe, Takumi Kishitani, Takuya Inoue, Masayuki Sato, Kazuhiko Komatsu, Yoichi Murashima, Shunichi Koshimura, and Hiroaki Kobayashi. 2019. Performance Evaluation of Tsunami Inundation Simulation on SX-Aurora TSUBASA. In *Computational Science – ICCS 2019.* Springer International Publishing, Cham, 363–376.
[18] Kenji Ono, Toshihiro Kato, Satoshi Ohshima, and Takeshi Nanri. 2020. Scalable Direct-Iterative Hybrid Solver for Sparse Matrices on Multi-Core and Vector Architectures. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region* (Fukuoka, Japan) *(HPCAsia2020).* Association for Computing Machinery, New York, NY, USA, 11–21.
[19] Matthew Otten, Jing Gong, Azamat Mametjanov, Aaron Vose, John Levesque, Paul Fischer, and Misun Min. 2016. An MPI/OpenACC implementation of a high-order electromagnetics solver with GPUDirect communication. *The International Journal of High Performance Computing Applications* 30, 3 (2016), 320–334.
[20] James W. Lottes Paul F. Fischer and Stefan G. Kerkemeier. 2008. nek5000 Web page. http://nek5000.mcs.anl.gov.
[21] H. M. Tufo and P. F. Fischer. 1999. Terascale Spectral Element Algorithms and Implementations. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing* (Portland, Oregon, USA) *(SC '99).* Association for Computing Machinery, New York, NY, USA, 68–es.