Progress with Nebo: A portable, performant EDSL for multiphysics applications

SIDDARTHA RAVICHANDRAN, MICHAEL BROWN, BABAK GOSHAYESHI, and JAMES SUTHER-LAND, University of Utah

Nebo is a declarative language specific to the domain of numerical solution of partial differential equations on structured grids. Nebo supports three modes of execution across CPU and GPU back-ends for sequential and parallel execution. The syntax is consistent across the various modes of execution making the task of writing applications on top of Nebo easy. Nebo is embedded in C++ and uses template meta-programming extensively to move much of the complexity to compile time, thus making it performant at run time. In this paper, we present extensions made to Nebo to support particle transport including smoothed particle hydrodynamics as well as integration with the Kokkos library. We also discuss performance of Nebo on CPU and GPU platforms.

1 INTRODUCTION

Techniques for writing high performance software that exploit increasingly diverse hardward - from multicore CPUs to GPUs - evolve more slowly than hardware. And more often than not such code is written at a very low level, which can be labor intensive and error-prone for domain scientists. With that in mind and specific to the domain of solution of PDEs on structured grids, Nebo aims at improving productivity for application programmers/domain experts by exposing interfaces that are scalable, consistent and portable across multiple architectures, without compromising on efficiency and performance. Nebo was designed specifically to isolate software developers from hardware details. Nebo is in essence a declarative domain-specific language (DSL) embedded in C++ [Earl et al. 2017].

Nebo was designed for use in high-performance simulation projects such as Wasatch [Saad and Sutherland 2016], which is a component within the Uintah [Berzins et al. 2012; Parker 2002] framework and has demonstrated scalability to 262,000 cores. Wasatch provides a flexible toolkit for solving convection-diffusion-reaction problems, and focuses on turbulent reacting flow simulations using large eddy simulation.

Nebo has a restrictive declarative syntax, represented as an abstract syntax tree (AST) by leveraging the C++ template system. There are currently three modes of execution: sequential (single-threaded), parallel-CPU (multi-threaded) and parallel-GPU. Nebo generates efficient code based on the intended backend/mode of execution for a given computation.

Memory management and data transfers between devices is left to either the framework or the end user, although Nebo has tools for allowing the user to migrate data between CPU and GPU. By providing a restricted syntax, Nebo is intentionally limited in its capability for its domain. It is intended to be used as a toolkit to enable high-performance simulation of PDEs (or parametric ODEs) on structured grids. The sequential mode of execution performs at least as well as the hand-written code it replaces. However, the parallel modes scale well on both multicore CPU and GPU platforms [Earl et al. 2017].

In the following sections, we cover a brief overview of what Nebo already provides followed by the extensions made to it since the description provided by [Earl et al. 2017]. Specifically, Nebo has introduced particle-cell interpolation to solve conservation equations thereby enabling hybrid computing in multiphysics simulations. We have also implemented kernels to facilitate smoothed particle hydrodynamics (SPH) simulations via a mapped reduction operation. Finally, we have explored means to improve performance by switching to a Kokkos [Edwards et al. 2014] backend to target multiple architectures.

2 CONSTRUCTS

Before discussing new features in Nebo, we briefly review the key constructs that Nebo provides, with further details available in [Earl et al. 2017].

2.1 Spatial Fields

Computations performed using Nebo essentially work on meshes, stored as an array of data. A `SpatialField` is a construct is used to represent a field on a mesh in up to three dimensions. There are numerous types of fields supported representing the physical location of the field on the mesh: cell volume centers as well as x, y and z faces. In addition, Nebo supports staggered meshes which are commonly used in finite-volume computational fluid dynamics simulations. Particle field types are also supported, for nearly 20 distinct field types in Nebo.

To create a field, we specify its memory window, boundary cell information, ghost cell information and optionally the device on which the field's memory would be allocated. There are provisions to expand a field to multiple devices but at a given time a field can be active on only a single device. A field on multiple devices, can only be written to on the active device and its memory on all other devices requires re-validation once the write is complete. Similarly a field values can be read from a particular device if it is valid on that particular device. This is controlled by the valid flag for each device the field exists on.

A memory window, as the name suggests, is a window into the memory of a particular field. It contains information (extents and offsets) needed to arrive at the flat index of the field's first value. And also information needed to determine the flat index of a particular field value while iterating over up to three dimensions of the field. This allows Nebo operations to be performed over subsets of the field if needed.

Nebo naturally supports the concept of ghost cells to enable distributed computing where the global field is partitioned onto processes and ghost regions are populated with neighbor process values to facilitate stencil operations. Nebo allows for fields to have an arbitrary number of ghost cells on each of the six faces of the field.

2.2 Spatial Masks

A `SpatialField`, as described above, represents a continuous space in up to three dimensions. Nebo also provides the notion of a `SpatialMask`, which defines a set of points and can be used in conjunction with Nebo operations to

perform calculations only on the mask (see §3.3). This is particularly useful in imposing boundary conditions, where a mask may be created for each boundary to facilitate operations germane to that boundary.

3 TRADITIONAL NEBO OPERATIONS

3.1 Mathematical Operators

Expressions written in Nebo include the following: algebraic operators (addition [·+·], subtraction [·−·], multiplication [·∗·], and division [·/·], and negation [−·]); trigonometric functions (sine [sin(·)], cosine [cos(·)], tangent [tan(·)], and hyperbolic tangent [tanh(·)]); extremum functions (minimum [min(·, ·)] and maximum [max(·, ·)]), and other mathematical functions (exponentiation with base e [exp(·)], exponentiation with given base [pow(·, ·)], absolute value [abs(·)], square root [sqrt(·)], and natural logarithm [log(·)]). Nebo provides support for these operators and functions through operator (and function) overloading and template meta-programming, and the set of supported functions is easily extensible. These operators can be arbitrarily nested with other Nebo constructs including assignments (§3.2), conditionals (§3.3), reductions (§3.4) and stencils (§3.5).

3.2 Assignment

Assignments allow for a field to be written into using the assignment operator[· <<= ·]. As expected the LHS would be a field and the RHS would be an expression which would produce a value for each point in the LHS. These expressions could include mathematical operators seen above, conditional expressions (§3.3), stencil operations (§3.5) and mapped reductions (§3.4).

```
Field a, b, c, f;
// ...
f <<= a + b * c;
```

The above snippet of code writes into f across its extents using the values of fields a, b and c. For every point p across the extents of f, $f_p = a_p + b_p * c_p$.

3.3 Conditional

As Nebo operates on a field pointwise over three dimensions, there needs to be someway to operate conditionally at a particular (x,y,z) based on the value of the field at that point. The *cond* operator allows one to write such code.

```
Field a, b, f;
// populate a , b ...
f <<= cond(a > 0, 1.0)
        (a);
```

The above snippet of code writes into f across its extents using the values of fields a and b. For every point p across the extents of f, if a_p (the value of a at p) is greater than 0 then $f_p = 1.0$ else $f_p = a_p$.

Nebo supports the following boolean operations on fields to form valid expressions to be used within cond: using any of the C++ numeric comparison operators (· == ·, ·! = ·, · < ·, · > ·, · <= ·, and · >= ·); or a logical connective of Nebo boolean expressions, using any of the C++ logical connective operators (·&&·, ·||·, and !·).

As mentioned in §2.2, we can use masks in conjunction with the cond operator:

```
Field a, b, c, f;
// populate a, b, c ...
Mask m;
```

```
157    // define mask points ...
158    f <<= cond( a < b,  −1.0)     // if a[i]<b[i], f[i] = −1
159                ( m,  c )          // elseif m[i] is a masked point, f[i] = c[i]
160                ( 2.0 );           // else f[i] = 2.0
```

The above snippet of code writes into f across its extents using the values of fields a. For every point i across the extents of f, if a_i (the value of a at p) is less than b_i then $f_i = -1.0$ else if point i is a mask point, then $f_i = c_i$, otherwise $f_i = 2.0$. Each of the fields in the cond statement above could themselves be Nebo expressions, inlined within the call to cond.

3.4 Reduction

These operations enable reducing the values of a field into a singular value. For instance, finding sum of all the values of a field:

```
Field a,b;
// populate a , b ...
const double s = nebo_sum( a + tanh(b) );
```

In the above snippet, for every point i across the extents of a and b, $s = \sum_p a_i + \tanh(b_i)$. Nebo currently supports reductions for min, max, sum and L_2 norm.

3.5 Stencils

While solving PDEs, stencil operations are used for interpolation and discrete calculus operations [Earl et al. 2017]. Nebo's support for stencils is extensible, and it presently supports nearly 200 distinct stencil operators to perform gradient, divergence and interpolation operations between different types of SpatialFields.

For instance, to solve the heat equation along a one-dimensional pipe, we would need to solve $\frac{\partial T}{\partial t} = \frac{1}{\rho c_p} \frac{\partial}{\partial x}\left(\lambda \frac{\partial T}{\partial x}\right)$. The term $\frac{\partial}{\partial x}\left(\lambda \frac{\partial T}{\partial x}\right)$ is computed using stencils as shown in the code snippet below and depicted in Figure 1.


```
typedef BasicOpTypes<SVolField>::DivX        DivX;
typedef BasicOpTypes<SVolField>::GradX       GradX;
typedef BasicOpTypes<SVolField>::InterpC2FX  InterpX;
//.. retrieve operators
SVolField divq, T, lambda;
SSurfXField q;
// set T, lambda ...
q <<= interpX(lambda) + gradX(T);
divq <<= divX(q);
```
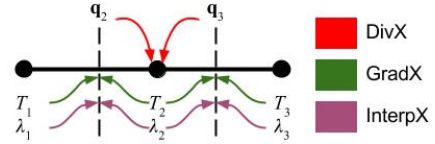
Fig. 1. Example of stencils involved in the finite-volume construction of $\frac{\partial}{\partial x}\left(\lambda \frac{\partial T}{\partial x}\right)$.

However, this could be combined as:

```
divq <<= divX( interpX(lambda) + gradX(T) );
```

which would eliminate a temporary field and result in one, rather than two kernel calls.

4 AST CREATION

The abstract syntax tree created by Nebo leverages on the meta-template system provided by C++. As seen above, using Nebo one can write a wide variety of expressions to run field assignments, reductions, etc.. These expressions form a

templated object in Nebo termed as a `NeboExpression`. For instance, the RHS of an assignment operation results in an `NeboExpression`, one which is evaluated for every point across the extents of the LHS.

```
Field out, a,b,c;
// populate a, b, c...
out <<= a + b * c;
```

This simple assignment operation results in the creation of the AST in terms of a `NeboExpression` as shown in Figure 2, and is encoded in the type type system as:

```
NeboExpression< SumOp<NeboConstField, ProdOp<NeboConstField, NeboConstField > >
```

A `NeboConstField` is a wrapper for a `SpatialField` coupled with the execution mode, constructed internally when the `NeboExpression` is created. Similar objects are created for scalars, single valued fields (fields that essentially store a single value across all its points) and mappers (see §6.2).

In the above example, the `NeboExpressions` are created by overloading operators $[\cdot + \cdot]$ and $[\cdot * \cdot]$. By using templates, Nebo can restrict the types of operands a particular operation can take. For example, if we were to run a mapped reduction we can restrict the second parameter to be a mapper and passing in an expression for the second argument will result in a compile error in that case.
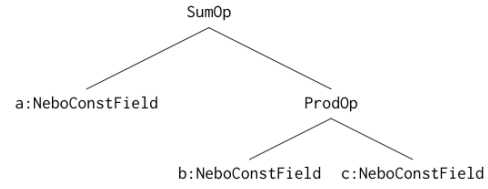

Fig. 2. Abstract syntax tree for out `<<=` a+b*c.

5 EXECUTION MODES/BACKENDS

Nebo supports three execution modes at runtime:

- Sequential (single threaded CPU) - SeqWalk
- Parallel CPU(multi threaded on the CPU) - Resize
- Parallel GPU(multi-threaded on the GPU) - GPUWalk

As discussed in §4, all computations are expressed as a `NeboExpression` consisting the various individual operations in the form of an AST. These operations include another template parameter that helps couple the backend/mode that will be used to execute the particular operation. The `Initial` mode is the default mode with which expressions are created.

```
Field a,b,c,out;
// populate a, b, c ...
out <<= a + b * c;
```

The above snippet produces the following `NeboExpression` when constructed initially at run time:

```
NeboExpression <SumOp<Initial, NeboConstField, ProdOp<Initial, NeboConstField, NeboConstField >>
```

Next, when the assignment is to be carried out or in other words when field out is to be written into, the following steps are carried out:

(1) For the assignment to be carried out, resource availability is satisfied, if for out active on the CPU, fields a, b and c are available and valid on the CPU device's memory. The same applies for GPU as well if it is the active device for out.

(2) Depending on the location of the field to be computed, the backend is selected for runtime deployment. The options presently supported in Nebo are the *sequential* (§5.1), *CPU-parallel* (§5.2) and *GPU-parallel* (§5.3) backends.

5.1 Sequential Backend

```
NeboExpression<SumOp<SeqWalk, NeboConstField, ProdOp<SeqWalk, NeboConstField, NeboConstField>>
```

The above expression is created and is evaluated for every point across the extents of the field out as follows:

```
for(int z = limits.get_minus(2); z < limits.get_plus(2); z++) {
  for(int y = limits.get_minus(1); y < limits.get_plus(1); y++) {
    for(int x = limits.get_minus(0); x < limits.get_plus(0); x++) {
      out(x, y, z) = rhs.eval(x, y, z); // rhs is the NeboExpression
    }
  }
}
```

5.2 CPU-Parallel

```
NeboExpression<SumOp<Resize, NeboConstField, ProdOp<Resize, NeboConstField, NeboConstField>>
```

The Resize mode is similar in execution to the SeqWalk mode in that each thread spawned executes the sequential code shown above but on a partition of the space/extents. Basically, based on the number of threads specified during deployment, partitions of the space are created and each partition is scheduled in a FIFO work queue, each with a instance of the NeboExpression created in Resize mode. Semaphores are used to synchronize onto the main thread once all the jobs are complete.

5.3 GPU-Parallel

```
NeboExpression<SumOp<GPUWalk, NeboConstField, ProdOp<GPUWalk, NeboConstField, NeboConstField>>
```

The CUDA kernel spawned by Nebo is one that spans 256 (16x16) blocks and a grid size based on the extents of the field as shown below.

```
int blockDim = 16;
int xGDim = xExtent / blockDim + ((xExtent % blockDim) > 0 ? 1 : 0);
int yGDim = yExtent / blockDim + ((yExtent % blockDim) > 0 ? 1 : 0);
dim3 dimBlock(blockDim, blockDim);
dim3 dimGrid(xGDim, yGDim);
```

Since the threads are created across two dimensions the GPUWalk loop is slightly different in comparison to the SeqWalk loop.

```
const int ii = blockIdx.x * blockDim.x + threadIdx.x;
const int jj = blockIdx.y * blockDim.y + threadIdx.y;
const int x = ii + xLow;
const int y = jj + yLow;
for(int z = zLow; z < zHigh; z++) {
  // rhs is the NeboExpression in GPUWalk mode
  if(valid()) { out(x, y, z) = rhs.eval(x, y, z); };
};
```

The valid() function guards against threads that are assigned outside the extents by CUDA as threads are assigned consistently across blocks. The end user is expected to handle some synchronization because Nebo uses asynchronous

kernel invocations. For instance, in Wasatch synchronization is handled by initializing CUDA streams and associating them with fields [Earl et al. 2017].

6 RECENT EXTENSIONS TO NEBO

6.1 Particle - Cell interpolation

For multiphase systems including the Lagrangian transport of particles within an Eulerian mesh, we frequently require interpolation between particle properties and mesh properties, *e.g.* between a `ParticleField` and a `CellField` (volume).

When performing these operations, particle size is needed to determine the overlap of particles with cells (*e.g.*, when a particle is crossing cell boundaries). This, together with information on the mesh itself, is used to interpolate information between particles and cells:

```
Mesh mesh;
CellField t_cell;
ParticleField t_particle;
ParticleField p_x, p_y, p_z; // particle positions
ParticleField d_p;           // particle diameter
// ...
CellToParticle c2p(mesh);
c2p.set_coordinate_information( &p_x, &p_y, &p_z, &d_p);
c2p.apply_to_field(t_cell, t_particle);
```

Pointers are provided to the `set_coordinate_information` method to allow for degenerate 2D and 1D cases.

In some cases, we may want to only consider the particle centroid rather than distributing information about the particle to all cells that it encounters. This is particularly attractive in regimes where the particle diameter is small relative to the cell size. Figure 3 illustrates the speedup observed when doing 'direct-injection' rather than interpolation. In obtaining these results, particles were randomly distributed through various domain sizes with an average particle number density of 0.1, 1 and 10 particles per cell.

Nebo also supports GPU execution for particle-cell interpolants. Speedups for GPU relative to single-core execution are shown in Figure 4. For larger mesh sizes we observe speedups of 40-50×.
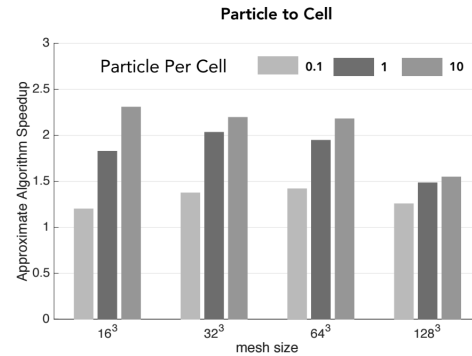

Fig. 3. Speedup when using direct-injection rather than trilinear interpolation in particle to cell interpolants.
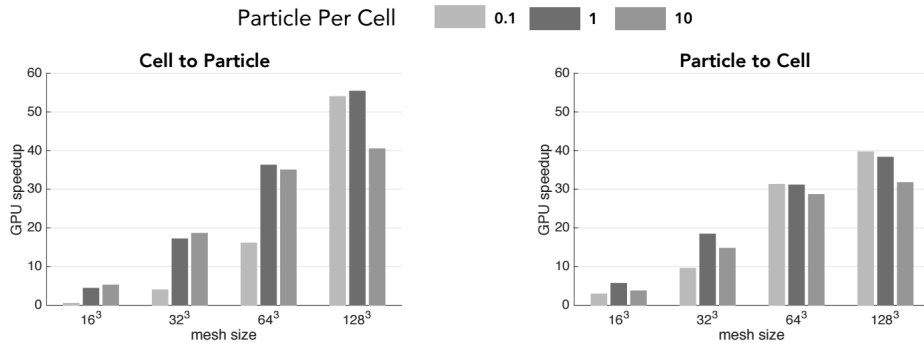

Fig. 4. Speedup achieved by running the interpolation on GPU.

6.2 Mapped Reduction

The mapped reduction operation was created with the intention of running computations to allow simulation of Smoothed-Particle Hydrodynamics (SPH). Let us illustrate this with an example.

```
ParticleField p_x, p_y, p_z; // particle positions
double h; // smoothing radius
double mass_p; // particle mass
size_t n; // number of particles

IntVec min_domain, max_domain; // represent simulation space bounds

RangeSearch rangeSearch (..., min_domain, max_domain, n, h, &p_x, &p_y, &p_z);

ParticleField v_x, v_y, v_z; // particle velocities

ParticleField density_t; // density at time step

density_t <<= nebo_mapped_reduction (mass_p*
        ((local(v_x) - v_x) * (local(p_x)-p_x)
      + (local(v_y) - v_y) * (local(p_y)-p_y)
      + (local(v_z) - v_z) * (local(p_z)-p_z))
      * W(mapped_value()), rangeSearch.getCurrentState())
```

The above snippet is essentially running $\rho_i = \sum_j m * v_{ij} * W(d_{ij})$. That is, for every particle i, we perform a summation over all particles j (otherwise referred to as a mapping) that fall within the smoothing radius of particle i. We perform a range search to find this mapping for each particle via the construction of a `RangeSearch` object, which is wrapped by Nebo as a mapper. W represents a kernel function for SPH computation and d_{ij} represents the distance between particle i and j. v_{ij} represents the relative velocity between the i and j particles.

The `nebo_mapped_reduction()` function takes in two parameters, the first being a nebo expression and the second being the current state of the mapper which provides the mapping (set of points) over which the reduction (summation in the example) is performed. The fields in the nebo expression are indexed based on the mapping and in order to access the value of particle i, we use the `local()` operator. The distance between a given pair of particles ij is given by `mapped_value()` in the case of range search. In general, `mapped_value()` is meant to access any value provided by the mapper on which basis the mapping was produced.

The `RangeSearch` object constructs a uniform grid structure based on the bounds of the particles and the smoothing radius. Based on the size of the each cell in the grid, we essentially reduce the computation needed to locate the particles that fall within the smoothing radius of a given particle, as we can restrict our search to only a certain set of adjacent cells. Figure 5 illustrates how constructing a uniform grid with cell size $h/\sqrt{2}$, reduces search to only the adjacent 20 cells.

Experiments from running mapped reductions on particle sets were also carried out using a `RangeSearch` mapper. Each time step involved three mapped reduction operations each for density, x-velocity and y-velocity. They also involved several
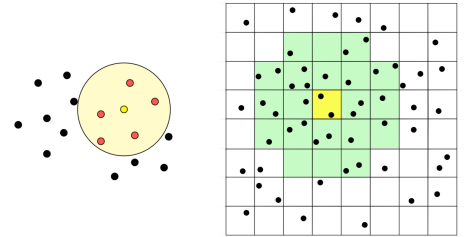

Fig. 5. In the image on the left, the red particles represent all particles that fall within the smoothing radius of the yellow particle. The image on the right represents how construction of a uniform grid reduces the search.

assignments to update the positions of the particles. Figure 6 shows speedups relative to single-core timings for various numbers of particles. The experiment was deployed to operate on 48 threads on 24-core system and a tesla K80 GPU.

The mapped_value() node which is used to access the distance (d_{ij}) is currently not functional for a GPU backend. This causes the GPU speedup to saturate after around 2500 particles. We expect the mapped_value() node to be extended in future and achieve better speedups on a GPU backend at larger particle counts.

Fig. 6. Multi-core and GPU speedup relative to sequential execution of mapped reductions.

6.3 Kokkos Integration

Kokkos implements a programming model in C++ for writing performance-portable applications targeting all major HPC platforms [Edwards et al. 2014]. It provides abstractions for both parallel execution and data management. We have implemented a Kokkos backend in Nebo to compare it to Nebo's native backend. Kokkos presently provides support for four platforms: Serial, OpenMP, PThreads and CUDA. Instead of initializing our execution space within Nebo and calling a mode specific assign (like we saw in §5), we call a kokkos_assign function as shown below.

```
// Execution space refers to the Kokkos platform: Serial, Cuda, etc.
// GPU flag is used internally to generate the appropriate functor wrapper
// Lhs and Rhs types are inferred
template<typename ExecutionSpace, bool GPU, typename LhsType, typename RhsType>
inline static void kokkos_assign(LhsType lhs,
                                 RhsType rhs,
                                 IntVec const & extents,
                                 GhostData const & ghosts,
                                 IntVec const & hasBC,
                                 GhostData const limits)
{
    const int xExtent = limits.get_plus(0) - limits.get_minus(0);
    const int yExtent = limits.get_plus(1) - limits.get_minus(1);
    const int zExtent = limits.get_plus(2) - limits.get_minus(2);
    const int length = static_cast<int>(xExtent * yExtent * zExtent);

    // Kokkos function to execute the functor wrapper based on the execution space
    Kokkos::parallel_for(Kokkos::RangePolicy<ExecutionSpace, int>(0, length),
                         KokkosFunctorWrapper<LhsType, RhsType, GPU>(lhs, rhs, limits));
}
```

We have achieved no speedup by integrating Kokkos with Nebo across varying domain sizes and type of operations for sequential execution. We speculate this to be occurring due to Kokkos's flat indexing scheme in comparison to Nebo's triple indexing. Having to switch from a flat index to a triple index for every functor call is an expensive operation for large domain sizes. We ran three operations to ascertain performance of the Kokkos backend:

- Simple Addition: f1 <<= f2 + f3
- Compound Statement: f1 <<= cos(pow(sin(pow(f2, f3)), f3))/f3;
- Stencil Statement: f1 <<= DivX(GradX(f2 + f3)) - DivY(GradY(f2 + f3)) - DivZ(GradZ(f2 + f3));

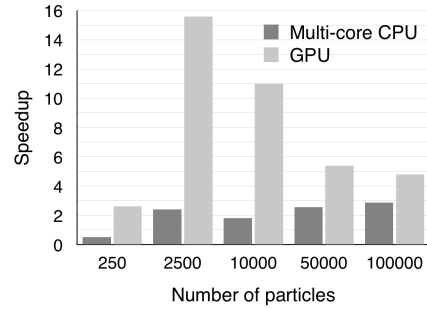Figure 7 shows speedups on multicore-CPU and GPU relative to the native Nebo implementation.
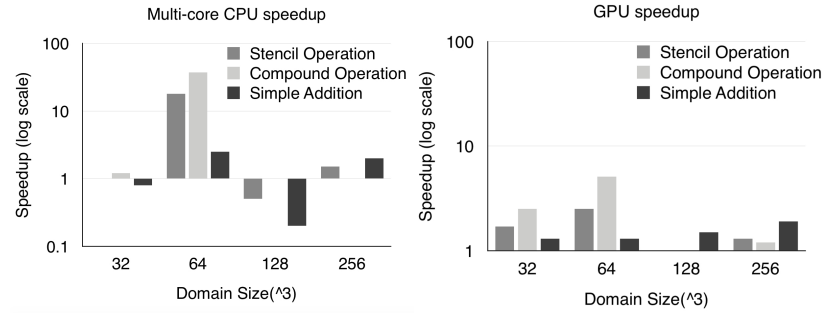
Fig. 7. Multi-core CPU and GPU speedup relative their corresponding native Nebo backends.

The speedups for multi-core CPU are erratic for a domain size of 32^3 owing to the large amount of time taken by native Nebo for performing thread synchronization. We could achieve a better speedup but for the conversion between flat and triple indexing that happens for every evaluation.

7 CONCLUSIONS

Nebo enables application programmers and domain experts to write code that is efficient, scalable, and portable across multiple architectures. A single code base can be written to scale under parallel architectures (multi-core CPU and GPU) which clearly outperforms the serial execution. By embedding Nebo in C++, we avoid the need to adopt an independent compiler (like in other DSLs), thereby also benefiting from a single-phase compilation. Also, by leveraging operator overloading and the template metaprogramming in C++, Nebo has been able to provide intuitive interfaces for application programmers to express a large number of computations in their domain. We have extended Nebo to support computations on particles, and results show reasonable performance on GPU. We have also implemented a Nebo backend using the Kokkos portable performance library and demonstrated that Kokkos provides speedup relative to native Nebo backends in some, but not all, cases.

REFERENCES

M. Berzins, Q. Meng, J. Schmidt, and J. C. Sutherland. 2012. *EuroPar 2011: Parallel Processing Workshops.* Vol. 7155. Springer, Chapter Dag-based software frameworks for PDEs, 324–333.

Christopher Earl, Matthew Might, Abhishek Bagusetty, and James C. Sutherland. 2017. Nebo: An efficient, parallel, and portable domain-specific language for numerically solving partial differential equations. *Journal of Systems and Software* 125 (March 2017), 389–400. https://doi.org/10.1016/j.jss.2016.01.023

H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202 – 3216. https://doi.org/10.1016/j.jpdc.2014.07.003 Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

Steven G. Parker. 2002. *Computational Science ICCS 2002.* Vol. 2331. Springer, Chapter A Component-based Architecture for Parallel Multi-Physics PDE Simulation, 719–734.

Tony Saad and James C Sutherland. 2016. Wasatch: An architecture-proof multiphysics development environment using a Domain Specific Language and graph theory. *Journal of Computational Science* 17, 3 (may 2016), 639–646. https://doi.org/10.1016/j.jocs.2016.04.010