Acceleration of Derivative Calculations with Application to Radial Basis Function – Finite-Differences on the Intel MIC Architecture

Gordon Erlebacher
Department of Scientific Computing
Florida State University
gordon.erlebach@gmail.com

Erik Saule
Department of Computer Science
University of North Carolina at Charlotte
esaule@uncc.edu

Natasha Flyer
Computational and Information Systems Laboratory
UCAR
flyer@ucar.edu

Evan Bollig
Minnesota Supercomputer Institute
University of Minnesota
bollig@gmail.com

ABSTRACT

In this paper, we develop an efficient scheme for the calculation of derivatives within the context of Radial Basis Function Finite-Difference (RBF-FD). RBF methods express functions as a linear combination of spherically symmetric basis functions on an arbitrary set of nodes. The Finite-Difference component expresses this combination over a local set of nodes neighboring the point where the derivative is sought. The derivative at all points takes the form of a sparse matrix/vector multiplication (SpMV).

In this paper, we consider the case of local stencils with a fixed number of nodes at each point and encode the sparse matrix in ELLPACK format. We increase the number of operations relative to memory bandwidth by interleaving the calculation of four derivatives of four different functions, or 16 different derivatives. We demonstrate a novel implementation on the Intel MIC architecture, taking into account its advanced swizzling and channel interchange features. We present benchmarks on a real data set that show an almost sevenfold increase in speed compared to efficient implementations of a single derivative, reaching a performance of almost 140 Gflop/s in single precision. We explain the results through consideration of operation count versus memory bandwidth.

Categories and Subject Descriptors

F.2 [**Analysis of Algorithms and Problem Complexity**]: Numerical Algorithms and Problems

Keywords

MIC, SIMD, SpMV, Sparse Matrix, Radial Basis Function

1. INTRODUCTION

The multiplication of a sparse matrix by a dense vector (SpMV) is an important kernel in many applied fields such as fluid dynamics [3], recommendation systems [4] and graph drawing [12]). Naturally, improving the performance of SpMV has captured the interest of many researchers; including the development of various implementations for CPUs [5, 28] and GPUs [2, 13, 26, 15]. The main challenge to obtain good performance for matrix vector multiplication in general, and sparse matrix vector multiplication in particular, is the low ratio of floating point operations to memory bandwidth. When the matrix is not dense, the problem is exacerbated due to non-uniform access patterns.

Common improvement techniques such as bandwidth reduction (matrix reordering [9]), register blocking, partitioning to fit in cache or TLB [18, 22, 23], unrolling [17] have impacts that are very dependent on the matrix and overall do not lead to dramatic improvement. (The state-of-the-art techniques in OSKI [27] provide some useful yet limited improvements). Register blocking [23] does not work well to many of the matrices in general use (although it can be applied with virtually no overhead thanks to compressed representations [6].) Indeed, there are about 8 bytes of the matrix to transfer from memory per nonzero in single precision; each nonzero requires two floating point operations leading to a flop-to-byte ratio of at most $\frac{1}{4}$. This limits the obtained performance to at most a quarter of the bandwidth of the architecture, wasting a lot of potentially useful cycles. The commonly used techniques are mostly designed to reach that bound rather than overcome it.

Fortunately that fate is not inevitable. One solution would be to schedule a more instruction-intensive kernel simultaneously with the execution of SpMV, relying on some hardware threading capabilities, such as Hyper-

Threading, to reduce the cycle wastage. However, most of the applications that use SpMV do not typically have an instruction-intensive kernel to run simultaneously.

Another solution, pursued in this paper, is to compute multiple SpMVs using matrices with identical spasity patterns, but with different matrix elements. Obviously not all the applications have such a property. However, important classes of applications such as graph recommendation [14], and the computation of derivatives for solving systems of PDEs using Radial Basis Function-generated Finite Differences (RBF-FD) [10] fall into the category of applications that require multiple SpMVs simultaneously. RBF-FD is a meshless method, which easily handles irregular geometries and local refinement with algorithmic complexity independent of dimensionality, and which can produce high-order derivative approximations. These methods are rapidly gaining ground in science and engineering modeling communities [1, 7, 11, 10, 21]. As a result, it is of interest to develop an efficient implementation on novel computer platforms for the calculation of derivatives within the context of RBF-FD; derivative calculations that account for the bulk of computer resources when running a numerical simulation implemented with RBF-FD. Typical equations solved by RBF-FD depend on multiple derivatives of multiple functions. Rather than compute each derivative individually, this paper investigates how to compute all derivatives simultaneously in an interleaved fashion. Specifically, we calculate four different derivatives (corresponding to four different sparse matrices with identical sparsity pattern) of four different functions (a common scenario in 3D fluid dynamics modeling) for a total of 16 derivatives.

To perform our analysis, we focus our attention on the improvement that can be achieved on the Intel Xeon Phi processor. It follows the Many Integrated Core (MIC) architecture, which has a significant memory bandwidth and peak flop throughput thanks to its 512-bit large SIMD registers. The Xeon Phi processor has been shown to be promising for sparse linear algebra compared to more classical CPU or GPU architectures [19, 16, 8].

In Section 2, we introduce and further motivate the RBF-FD method, giving an example of how calculating the derivatives for a common system of PDEs in fluid dynamics can be expressed as sixteen multiplications of four vectors by four sparse matrices with identical sparsity patterns.

Section 3 presents an estimation of the instruction intensity of various forms of the computations. We show that a sevenfold improvement can be expected when computing the sixteen multiplications simultaneously to reach a total of about 210 Gflop/s. This performance represents approximately 10% of the available flop/s of a Xeon Phi coprocessor. Provided the computation is mostly irregular, it is necessary to have an implementation that organizes the data in such a way it can perform a fully efficient Fused Multiply-Add every 10 cycles. We describe in Section 4 the details of the MIC architecture and how to use specialized load, store, swizzle and permutation instructions to efficiently bring the data from memory into the vector registers to be processed. Section 5 presents results on memory access and computational speed and their relation to choices made in the SpMV kernels. It also provides the actual performance of the various kernels on multiple classes of matrices, some generated for purpose of analysis, and some extracted from an application of RBF-FD. A performance of 135 Gflop/s in single precision is achieved using RBF-FD differentiation matrices, which is 3.75 times better than the theoretical peak performance of an SpMV operation. Concluding remarks and perspectives are provided in Section 6.

2. INTRODUCTION AND MOTIVATION FOR RBF-FD

RBFs approximate a function $f(\mathbf{x}) \in \mathbb{R}^d$ sampled at N distinct node locations by linearly combining translates of a single radially symmetric function, e.g. $\phi(r) = e^{-\varepsilon^2 r^2}$, where $r = \|\mathbf{x} - \mathbf{x}_i\|$ denotes the Euclidean distance between where the function $f(\mathbf{x})$ is evaluated, $\mathbf{x}$, and where the RBF, ϕ, is centered $\mathbf{x}_i$. The parameter ε controls the shape of the RBF. Note that the argument of an RBF is simply a scalar distance, r, independent of any coordinate system or dimension. As a result, nodes can be scattered as desired across complex physical domains with the implementation of the method being independent of dimensionality. Thus, no mesh generation is needed and algorithmic complexity does not increase with dimension.

To obtain a RBF derivative approximation at a node location that will result in a sparse differentiation matrix (DM), the RBF-FD method has been developed over the last decade [24, 25, 20, 29]. RBF-FD is conceptually similar to standard finite differences (FD). However, the differentiation weights for approximating a derivative at a given node (forming one row of the RBF-FD derivative matrix (DM)) enforce that the linear combination of function values at the $n << N$ nearest node locations (a stencil) be exact for RBFs centered at each of the n nodes rather than polynomials. The result is a sparse matrix with only n entries in each of its N rows, resulting in a total of nN nonzero entries. An example stencil is shown in Figure 1.

Due to the above attributes and its simplicity of implementation, the RBF-FD is gaining popularity in modeling systems of PDEs in a variety of fields in fluid dynamics (e.g., geosciences, combustion, aerodynamics [1, 7, 11, 10, 21]). For example, the full shallow water equations in a 3D Cartesian coordinate system for a rotating fluid are as follows:

$$
\begin{aligned}
\frac{\partial u}{\partial t} &= -\left(u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z} + f(yw - zv) + g\frac{\partial h}{\partial x}\right) \\
\frac{\partial v}{\partial t} &= -\left(u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + w\frac{\partial v}{\partial z} + f(zu - xw) + g\frac{\partial h}{\partial x}\right) \\
\frac{\partial w}{\partial t} &= -\left(u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} + w\frac{\partial w}{\partial z} + f(xv - yu) + g\frac{\partial h}{\partial x}\right) \\
\frac{\partial h}{\partial t} &= -\left(\frac{\partial (uh)}{\partial x} + \frac{\partial (vh)}{\partial y} + h\frac{\partial (wh)}{\partial z}\right),
\end{aligned}
$$

where f is the Coriolis force, $\{u, v, w\}$ are the components of the velocity vector in the respective $\{x, y, z\}$ directions, and h is the geopotential height (analogous to

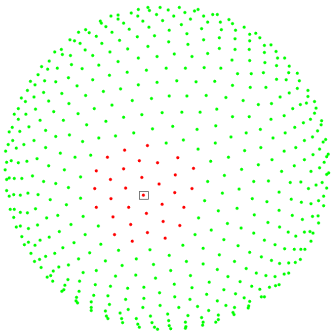


Figure 1: An example of an RBF-FD stencil of size $n = 32$ on a sphere of $N = 1024$ nodes to approximate any derivative operator at the point in the square. In nonsparse format, the DM is 1024×1024 with 32 nonzero entries per row.

pressure). Thus, we have four variables $\{u, v, w, h\}$ and three DMs, D_x, D_y, D_z representing ∂_x, ∂_y, ∂_z, respectively. Moreover, in order to stably time step the equations with an explicit time-stepping scheme, most numerical methods (including RBF-FD) require a fourth operator, hyperviscosity, to be added to each equation. Hyperviscosity takes the form of $\triangle^p$ ($\triangle$ denotes the Laplacian operator), where $p \geq 3$, which is approximated by the matrix D_{hyp}. Therefore, in order to solve this system of PDEs, the application of 4 DM to 4 unknowns, i.e. 16 SpMV is required at every time step. Due to the fact that RBFs are independent of a coordinate system, taking into account only distances to nearest neighbors, the DMs have the following properties:

- For a given n, regardless of the operator approximated, e.g., ∂_x, or the Laplacian, RBF-FD DMs will have the same number of nonzeros per row.
- For a given n and N, regardless of the operator approximated, the DMs will have the same sparsity pattern. Examples of an RBF-FD DM can be seen in Figures 8d-e.

Multiple variables transform a SpMV into a SpMM (Sparse Matrix/dense Matrix multiplication), which improves register utilization and decreases cache misses by vectorizing over the multiple source vectors. Furthermore, due to the properties just mentioned, multiple derivatives of a single function can be calculated rather than computing a single derivative of multiple functions. Using the PDE system above as an example, the block structure of the SpMM for computing all derivatives needed is

$$\begin{pmatrix} \underline{u}_x & \underline{v}_x & \underline{w}_x & \underline{h}_x \\ \underline{u}_y & \underline{v}_y & \underline{w}_y & \underline{h}_y \\ \underline{u}_z & \underline{v}_z & \underline{w}_z & \underline{h}_z \\ \underline{u}_{hyp} & \underline{v}_{hyp} & \underline{w}_{hyp} & \underline{h}_{hyp} \end{pmatrix} = \begin{pmatrix} D_x \\ D_y \\ D_z \\ D_{hyp} \end{pmatrix} \times (\underline{u}\ \underline{v}\ \underline{w}\ \underline{h}\ ) \quad (1)$$

where $\{\underline{u}, \underline{v}, \underline{w}, \underline{h}\}$ are the functions values at all nodes of the corresponding variables with the left hand side being the resulting derivative approximations. The increased memory bandwidth due to an increase in the number of derivative matrices is offset by improved cache utilization, leading to an overall performance benefit. In practice however, the number of vectors whose derivatives are required is limited, capping the performance of the matrix/vector multiplication. Similarly, the number of different derivatives in a particular computation is finite. It is for this reason that we seek to combine the benefits of multiple vectors and multiple matrices simultaneously. We limit this initial study to four vectors and four matrices, which is a common number found in fluid dynamic calculations. Equation 1 is of the form $Y = AX$, where Y encodes the x, y, z discrete derivatives (and hyperviscosity operator) of u, v, w, h, X encodes the variables (u, v, w, h) and A represents the 16 discrete differentiation operators.

We also note that the DMs are time-independent. They are computed once at the beginning of the simulation and used repeatedly to update the solution in time (for each iteration of the simulation). Thus, the cost to calculate the derivatives, although high (on the order of 1000 SpMVs), is amortized by the very long numerical simulations. The startup cost is eliminated when performing parameter studies on a given node distribution. Furthermore, the computation of the derivative matrices are done serially, but are trivially parallelizable.

3. MODELING THE POTENTIAL IMPROVEMENTS

We saw in the previous section that one can express the RBF problem as a multiplication of four matrices by four vectors. We present here an estimation of the variation on the flop intensity of the computation and its impact on the expected performance of the application. Relevant notation are given in Figure 2.

b_i	number of bytes per index
b_x	number of bytes per value
n_z	number of nonzeros per row of A
n_r	number of column/rows of A
n_c	total number of nonzeros
n_v	number of `x` vectors
n_m	number of matrices
s_M	size of the n_m matrices in bytes
s_x	size of the n_v `x` vectors in bytes
s_y	size of the $n_v n_m$ `y` vectors in bytes
cl	size of a cache line in bytes
b_{wT}	number of bytes written to memory
b_{rT}	minimum number of bytes read from memory
b_T	minimum number of bytes transferred
B_{rT}	maximum number of bytes read from memory
B_T	maximum number of bytes transferred
O	number of floating point operations
I_b	maximum computational intensity
I_w	minimum computational intensity

Figure 2: Notation relative to the application problem (upper section) and to the benchmarks (lower section.)

Each vector in the problem is of dimension n_r and each entry takes b_x bytes. There are n_v `x` vectors and $n_v n_m$ `y` vectors, which lead to the size of the `x` and `y` vectors:

$$s_x = n_v b_x n_r, \quad s_y = n_v n_m b_x n_r$$

The matrix is composed of n_r rows and columns with

n_z nonzeros per row leading to a total of

$$n_c = n_r n_z$$

nonzero entries in the matrix. Each of these nonzero entries has one index of size b_i and n_m values of size b_x. The matrices have a total size of

$$s_M = n_c(b_i + b_x n_m) = n_r n_z (b_i + b_x n_m)$$

If we assume an algorithm where the rows are processed one after the other, the amount of memory written is precisely the size of the `y` vector. (This assumption removes the possibility of blocking or cache partitioning techniques.)

$$b_{wT} = s_y = n_v n_m b_x n_r$$

The amount of data read from memory depends highly on both the algorithm's execution path, and on how the matrix is structured. But in the best case both the matrix A and the source vector `x` are read once from the main memory. (We assume that all the elements of `x` are involved in the SpMV.) Thus,

$$b_{rT} = s_M + s_x = n_r n_z (b_i + b_x n_m) + n_v b_x n_r$$

$$b_T = b_{rT} + b_{wT} = n_r n_z (b_i + b_x n_m) + n_v b_x n_r (1 + n_m)$$

Notice that there is no reason for a piece of the matrix to be read multiple times. But assuming that each element of the `x` vector is read a single time is a strong assumption. If using a single core, it assumes that either the cache of the architecture can store the full `x` vector or that the matrix is sufficiently well structured to cause no cache trashing. If using multiple cores, this assumes that no element of the `x` vectors will be used by multiple cores. [19] showed that there is very little cache trashing in practice; however having elements of the vectors used by multiple cores can have a significant impact on the performance (growing with n_v).

On the other hand, in the worst case, every time the `x` vector is accessed, the value needs to be transferred from memory again. So in total, there are as many transfers as the number of nonzeros in the matrix. Note however that most architectures cannot read memory a single byte at a time. Instead, a minimum number of bytes, equal to the size of a cacheline cl, are transferred at once. When there are multiple vectors, each nonzero element uses n_v consecutive entries. In the worst case, the number of bytes read and transferred is

$$B_{rT} = s_M + n_c cl \left\lceil \frac{n_v b_x}{cl} \right\rceil$$

$$B_T = n_v n_m b_x n_r + n_r n_z \left(b_i + b_x n_m + cl \left\lceil \frac{n_v b_x}{cl} \right\rceil \right)$$

In SpMV, each nonzero of the matrix requires two floating point operations: one for performing the multiplication and one for accumulating the result row-wise. Here we are dealing with $n_v n_m$ simultaneous SpMVs and the number of floating point operations is

$$O = 2 n_v n_m n_c = 2 n_v n_m n_z n_r$$

The computation intensity is the amount of floating point operations performed per byte transferred. In the worst case and in the best case, we have

$$I_b = \frac{O}{b_T} = \frac{2 n_v n_m}{(b_i + b_x n_m) + n_v n_m b_x n_z^{-1} + n_v b_x n_z^{-1}}$$

$$I_w = \frac{O}{B_T} = \frac{2 n_v n_m}{(b_i + b_x n_m) + n_v n_m b_x n_z^{-1} + cl \left\lceil \frac{n_v b_x}{cl} \right\rceil}$$

Figure 3 presents the flop to byte ratios for single precision computation in the best and the worst case on a classical cache-based architecture ($cl = 64$) and assuming $cl = 1$. We can easily see the potential improvement in the computational intensity when the number of matrices or vectors increases. There is a significant difference between the best and the worst case: there is a eightfold difference in the one matrix – one vector ($n_m = n_v = 1$) case but that gap closes with the increase in the number of vectors and matrices to a fourfold difference in the four matrices and four vectors ($n_m = n_v = 4$) case and 1.6 fold in the $n_m = n_v = 16$ case. The ratios computed with $cl = 1$ are mostly similar to the one with $cl = 64$ when the number of vectors is large. But the differences are important when the number of vectors is low: this highlights that accessing the memory with a granularity of one cache line is a main problem faced when performing a standard SpMV computation.

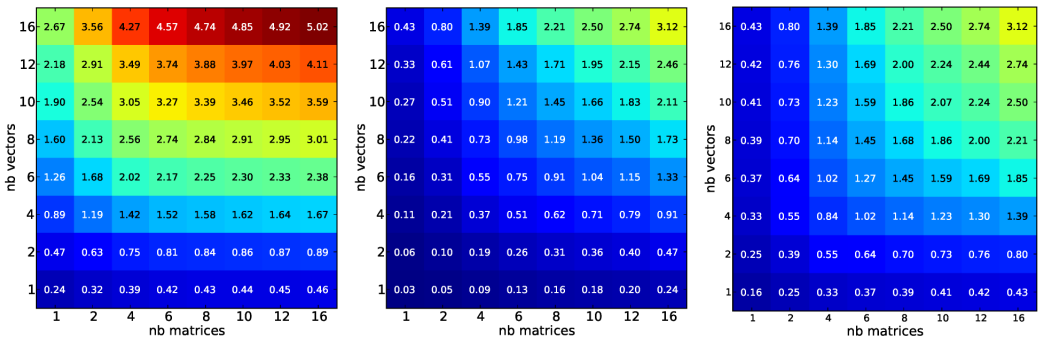
Figure 4 shows how these ratios translate into actual performance. This figure provides projected best and worst Gflop/s achievable assuming the computation is memory-bound and the architecture reaches 150 GB/s. Notice that [19] showed a higher peak bandwidth, but we will show in Section 5 why 150 GB/s is a better estimate of what one might achieve in these kernels. In single precision, the worse that one could achieve by using four vectors and four matrices is much higher than the best achievable using a classical SpMV computation. The best performance reachable is 210 Gflop/s: almost six times higher than the peak of the classical SpMV case.

4. EFFICIENT IMPLEMENTATION ON THE INTEL XEON PHI PROCESSOR

4.1 Intel Xeon Phi

In this work, we use an Intel Xeon Phi 5110P coprocessor. This card has 8 memory controllers that can each execute 5 billion transactions per second. Each has two 32-bit channels, achieving a total bandwidth of 320 GB/s aggregated across all the memory controllers. There are 60 cores clocked at 1.05 GHz. Their memory interfaces are 32-bit wide with two channels and the total bandwidth is 8.4 GB/s per core. Thus, the cores should be able to consume 504 GB/s at most. However, the bandwidth between the cores and the memory controllers is limited by the ring network that connects the cores and the memory controller. Its precise bandwidth is believed to be between 200 GB/s and 250 GB/s.

Each core in the architecture has a 32 kB L1 data cache, a 32 kB L1 instruction cache, and a 512 kB L2



(a) Best case. (b) Worst case. (c) Worst case (No cacheline effects)

Figure 3: Ratio of flops to bytes in single precision: 3(a) best case; 3(b) worst case when $cl = 64$; and 3(c) worst case neglecting that the memory transfers are with a granularity of one cache line (equivalent to $cl = 1$)

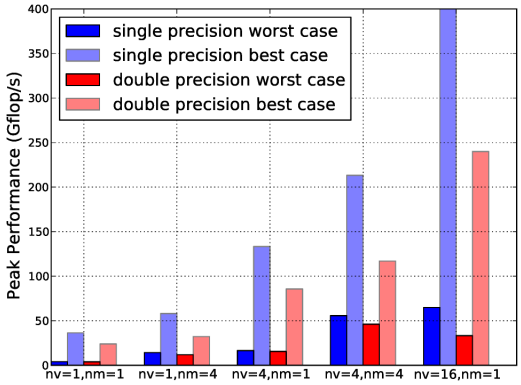


Figure 4: Estimation of the maximum achievable performance using a device with a bandwidth from memory to computational units of 150 GB/s when varying the number of vectors and matrices.

cache. The architecture of a core is based on the traditional Pentium architecture, which has been extended to 64-bit with hyperthreading. A core can hold four hardware contexts at any given time (240 threads in total). At each clock cycle, instructions from a single thread are executed. Due to some hardware constraints, two hardware contexts must be used to reach the peak instruction throughput in that architecture. Similar to the Pentium architecture, a core has two different concurrent instruction pipelines that allow the execution of two instructions per cycle. However, only one vector or floating point instruction can be executed per cycle.

Most of the performance of the architecture comes from the vector processing unit. Each core has 32 512-bit SIMD registers that can be used for double or single precision, that is, either as a vector of eight 64-bit values or as a vector of 16 32-bit values, respectively. The vector processing unit can perform arithmetic (addition and multiplication) instructions making it possible to reach 16 single precision (or 8 double precision) operations per cycle. The unit also supports Fused Multiply-Add (FMA) operations, which are typically counted as two operations when benchmarking. Therefore, the peak performance of the 5110P card is one Tflop/s in double precision and two Tflop/s in single precision. If FMA cannot be used, only half of these rates can be achieved.

4.2 Bringing the data into the vector register

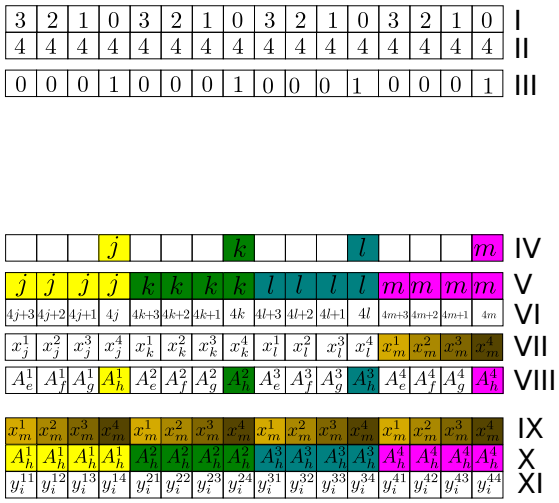
As explained in Section 3, we expect to achieve a performance of about 210 Gflop/s in single precision in the best case. We will focus on the single precision case with four vectors and four matrices. Similar techniques apply for other combinations. This target performance represents 10% of the peak performance of the architecture, which cannot be reached without an efficient vectorization. Indeed, to reach such a performance, a Fused Multiply-Add instruction on fully loaded registers must be executed at most every 10 cycles.

In term of memory layout, we adopt a format similar to the ELLPACK format to store the matrix since the number of nonzeros per row is constant throughout the matrix. The matrix is given in two arrays. The first array is `col_id` that describes the sparsity structure of the matrix. Each row of `col_id` consists of n_z entries that label the column index of each nonzero in A. The other array is `data` that gives the values of the nonzeros in the four matrices. The values of the matrices are interleaved by groups of four so that the value $A^l_{col\_id[k]}$ of the kth nonzero of the lth matrix is at index $\lfloor \frac{k-1}{4} \rfloor * 4 * n_m + (l-1) * 4 + (k-1)\%4$. (In other words, the first four entries are from A^1 and the next four are from A^2.) The vectors `x` are interleaved so that $\mathtt{x}^l_i$ and $\mathtt{x}^{l+1}_i$ are consecutive in memory. The vector `y` is similarly grouped in sets of 16, which correspond to four derivatives applied to four functions, at a single point.

We discuss how the implementation of the v4m4 kernel works using vector instructions. The code is given for reference in Figure 5 along with a graphical depiction

```
const __m512i offsets = _mm512_set4_epi32(3,2,1,0);
const __m512i four = _mm512_set4_epi32(4,4,4,4);
int int_mask = 0x1111;
__mmask16 mask = _mm512_int2mask(int_mask);

#pragma omp parallel for
for (int r=0; r < nb_rows; r++) {
 __m512 accu = _mm512_setzero_ps();

 for (int n=0; n < nz; n+=4) {
   float* a = &dom.col_id[0]+n+nz*r;
   __m512i vect_i = _mm512_setzero_ps();
   vect_i = _mm512_mask_loadunpacklo_ps(vect_i, mask, a);
   vect_i = _mm512_mask_loadunpackhi_ps(vect_i, mask, a);
   vect_i = _mm512_swizzle_ps(vect_i, _MM_SWIZ_REG_AAAA);
   vect_i = _mm512_fmadd_epi32(vect_i, four, offsets);

   all_vect = _mm512_i32gather_ps(vect_i, dom.vec_vt, scale);

   mat_ent = _mm512_load_ps(dom.data + nb_mat*(n + r*nz));

   //perform first tensor multiplication with AAAA pattern
   vect = permute(all_vect, _MM_PERM_AAAA);
   mat = _mm512_swizzle_ps(mat_ent, _MM_SWIZ_REG_AAAA);
   accu = _mm512_fmadd_ps(vect, mat, accu);

   //three more times with BBBB, CCCC, and DDDD patterns
 }
 _mm512_storenrngo_ps(dom.result_vt+nb_mat*nb_vec*r, accu);
```

(a) Source code



(b) Content of the vector registers.

Figure 5: Code snippet of the multiplication of 4 vectors with 4 matrices. When a line changes the content of a vector register its new content is shown on the right. The effects of swizzling and permutations are highlighted using a color code.

of the content of the registers. (We will show in Section 5 that an implementation that relies on compiler vectorization does not lead to desirable performance, which confirms the results of [19].) The registers in the MIC architecture are 512 bits wide and can store 16 floats or integers. (In comparison to OpenCL or CUDA, one can think of these registers as a warp.) The end goal is to load and format the nonzero values $A_h^1, A_h^2, A_h^3, A_h^4$ and the vector entries $x_m^1, x_m^2, x_m^3, x_m^4$ into two vector registers and apply Fused Multiply-Add on them, which is depicted in the `accu` line of Figure 5. Bringing the data into the vector register as shown in the figure is the difficult component of our proposed algorithm, which we proceed to explain further. A thread will perform the multiplications one row at a time, and parallelism is achieved by giving blocks of rows to each thread using an OpenMP construct.

The core of the technique is to load the maximum amount of data into large the registers and format the data for processing within these registers, minimizing the amount of interaction with memory. The Intel MIC is more for these operations than classical CPUs because it features larger vector registers and more elaborate vector reorganization instructions, such as swizzling, permutations and broadcasts. (The AVX instruction sets on classical CPUs are more primitive in that aspect.) A vector of 512 bits is composed of four lanes (sometimes called channels) of 128 bits, which are made of four segments of 32 bits. A swizzling operation allows the reorder of the elements within each lane (see how `mat` is extracted from `mat_ent` in Figure 5 line VIII to X). On the other hand, a permutation reorders entire lanes, without changing their contents (see for instance how `vect` is generated from `all_vect` in Figure 5 line VII to IX). Both permutation and swizzling support common reordering and broadcast. In the code, the "permute" function needs to use the `_mm512_permute4f128_epi32` instruction which is only defined on integers and therefore in place typecasts are necessary to convert a vector to and from integer type. (Note that we prefer swizzling to permutation if possible because we believe it is cheaper. Many instructions have swizzling capability, while permutation is its own instruction.)

Every operation that loads data from memory into a register transfers 512 bits, or 16 floats at a time. A single nonzero in the matrix will cause 16 operations to be performed, but it only uses four floats from the matrix and four floats from the `x` vector. So we would like to execute a single instruction to load the data for four nonzeros in the matrix, and another instruction to load the data for four nonzeros from the vectors. (The number of nonzeros on a row in our application is always a multiple of four; if it weren't, we would add some explicit zeros in the data structure for padding.)

The values from the nonzero elements in the matrix are the simplest to load. One can load the 16 floating point values for four nonzeros of the matrix at once. The four floats coming from each matrix are naturally grouped within each lane of the SIMD register. One can use a swizzle operation to broadcast the four elements of a single matrix to fill the whole register. (The broadcast of $A_h^1, A_h^2, A_h^3, A_h^4$ is shown in Figure 5.)

Loading the entries from the `x` vector is more involved. First, the column pointers of four different nonzeros are loaded into a vector register and the values are distributed one per lane of the vector register using an

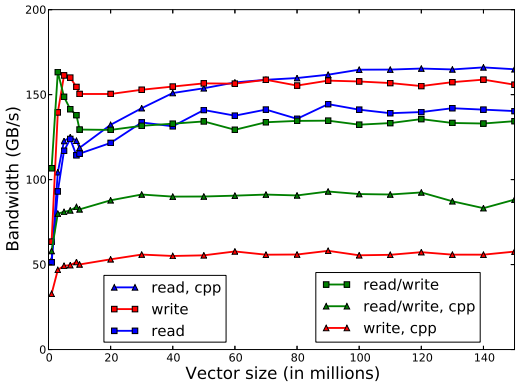


Figure 6: Bandwidth performance under idealized conditions. Entries with "cpp" (triangles) denote cases where vectorization is compiler generated.

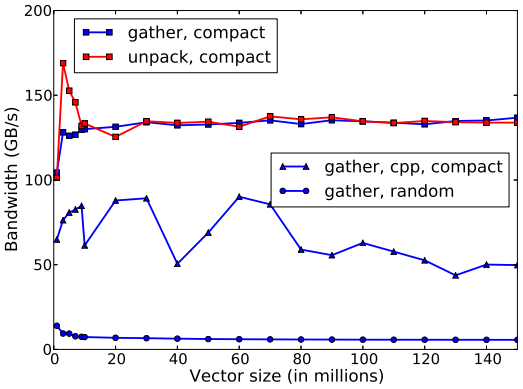


Figure 7: Performance of `gather` and `unpack` operations.

`unpack` operation in line IV[1] (The column pointers are named j, k, l, m in Figure 5.) They are replicated so that each value fills a single vector lane using a swizzle operation (in line V). Then each value is multiplied by four (because $n_v = 4$) and the offset vector $(3, 2, 1, 0)$ is added to each lane to obtain the correct index of the elements of the `x` vector within each lane using a Fused Multiply-Add operation in line VI. A `gather` operation is then performed in line VII to bring the 16 entries of the `x` vector into the SIMD register. At this point we have a register where each lane is the four vector entries for a nonzero element. Using a broadcast permutation, one can replicate an entry of the four vectors across each lane. (The broadcast of $x_m^1, x_m^2, x_m^3, x_m^4$ is shown in Figure 5 in line IX.)

Now that $A_h^1, A_h^2, A_h^3, A_h^4$ and $x_m^1, x_m^2, x_m^3, x_m^4$ are properly laid out in the vectors, a partial value for y_i can be computed in line XI. Once all the nonzeros of the row have been processed, the exact y_i value can be sent to memory using a the "No Read No Global Order" hint to optimize write traffic on the memory bus.

The model of Section 3 establishes that the memory transfer caps the computation at 210 GFlop/s, which is 10% of the peak single precision of the architecture. To reach this value, the cores need to execute one "useful" Fused Multiply-Add operation every ten cycles. The inner loop of the kernel spends 7 vector instructions to layout the data and then 3 per tensor product for a total of 12 vector instructions. Most of these instruction have a one cycle throughput cost except the gather operations which might need to perform up to 16 memory accesses. There are also some scalar operations but there are few of them and are executed on a different pipeline than the vector operations. Overall, the inner loop can emit the 4 Fused Multiply-Add in less than 40 cycles.

5. EXPERIMENTAL VALIDATION

In this section, we describe a series of numerical experiments to confirm the performance predictions of Section 3. To better understand our results, we first perform some bandwidth measurements, followed by the actual SpMV computations. All the codes are compiled with the Intel C Compiler version 13 and -O3 optimization. The codes use the IMCI instructions exposed by the compiler through intrinsics.

5.1 Bandwidth

Figure 6 shows a benchmark that computes the bandwidth achieved when reading and/or writing large arrays on the MIC card. The read benchmark performs a simple sum of the array. The write benchmark sets all the elemnts of the array to zero, while the read/write benchmarks copies the array into another one. We also investigate the difference between a straight C++ implementation (compiled with -O3 with proper tagging to inform the compiler of memory alignments) and an implementation with explicit use of the vector registers.

The `read_cpp` implementation reaches a performance of 168 GB/s which is better than the `read` implementation which only achieves 145 GB/s. This is explained by the compiler adding loop unrolling and software prefetching in the assembly code in the `read_cpp` case, which is not added in the `read` case. On the other hand, the `write` case reaches a performance of 158 GB/s while the `write_cpp` implementation never exceeds 58 GB/s. This difference is the result of `_mm512_store_nrngo_ps`, an instruction, which bypasses the "Read For Ownership" protocol and allows the writes to be performed in any order. The compiler can not use this instruction on its own because this would break the memory convention useful for some parallel algorithms. The `read/write` benchmark performs at 140 GB/s while the `read/write_cpp` benchmark performs at about 90 GB/s.

Note that SpMV kernels are not likely to use only simple read and simple writes. We also benchmark the performance we can achieve using `gather` and `unpack` operations and we present the results in Figure 7. The `unpack` instruction can reach a performance of 140 GB/s. (There is no easy way to implement the equivalent of the `unpack` operation without using explicit registers.) The performance of the `gather` operations depends in

[1]Notice that there are two calls to `unpack` for the LSB and MSB. Both are mandatory even if one is known to be nilpotent according the documentation of the hardware.

which order the array is read. If the indices are ordered (`compact`) then a performance of 140 GB/s is reached. If the indices are completely randomized, the performance drops below 5 GB/s. We can see that implementing the indirection mechanism without vector instructions lowers the performance significantly. (In the `compact` case, it never surpasses 100 GB/s and is close to 50 GB/s most of the time.)

To summarize, using vector instructions appears necessary in order to reach the highest bandwidth on the MIC architecture. For the instructions used in a typical SpMV computation, one can expect a bandwidth between 140 GB/s to 160 GB/s. That is why we use in our estimations a best achievable bandwidth of 150 GB/s despite the fact that a code carefully crafted to maximize bandwidth can reach higher values [19].

5.2 Instances

The different matrices used to perform our experiments and analysis are presented in Figure 8. The `Supercompact` matrices (Figure 8(a)) have been generated to only have nonzero elements in the first 32 columns of the matrix. As a result, only 32 values of `x` are read from memory, and the expense associated with cache misses is removed from consideration. The cost associated with `A` remains. The `Compact` matrices (Figure 8(b)) are generated to have 32 nonzeros per row centered around the diagonal and it represents the ideal case for many applications that rely on sparse matrix vector multiplication. The nonzero elements of the `Random` matrices (Figure 8(c)) see their nonzero elements randomly (uniformly) distributed in the matrix; they represent the worst case scenario for a cache-based architecture where the cache reutilisation is the lowest from one row to the next. The other type of matrix is used in derivative stencils in 3D RBF-FD calculations (Figure 8(d) shows a 32-point stencil of a 3D 8^3 grid).

We also apply a Reverse Cuthill-Mckee reordering to all the matrices. This ordering technique aims to reduce the distance between the nonzeros and the diagonal, hopefully increasing the cache hit ratio. The reordered version of the matrix can be seen in Figure 8(e).

5.3 Computations

We now investigate the actual performance that we can obtain when multiplying four vectors by four matrices in single precision. Figure 9 presents results for different types of matrices and also shows the minimum and maximum performance predicted in Section 3. Figure 9(a) gives the results for the `supercompact` and `random` matrices that represent both the best and the worst case for such a practical computation. The `supercompact` case peaks at 208 Gflop/s which is very close to the predicted peak performance of 213 Gflop/s. Conversely the performance of the `random` case decreases to 56 Gflop/s which is close to the lowest predicted performance of 55 Gflop/s. One can see that RCM ordering helps the `random` case but the impact decreases when the size of the matrix increases.

Figure 9(b) gives the results on the `compact` case, which represents the best realistic matrix one could find with very structured grids and a `RBF` derivative matrix extracted from a real 3D application. In the `compact` case, the performance can be as high as 195 Gflop/s, which is within 15% of the predicted peak performance. The performance of the `RBF` case varies between 100 and 140 Gflop/s. Most of the time, the RCM ordering provides an improvement which can be as high as 30 Gflop/s.

We finally investigate the central questions of this paper. Do we gain actual performance by transforming classical SpMV computation into the multiplication of four vectors by four matrices for the computation of the derivative of RBFs? Does using manual vectorization improve actual application performance? Figure 10 compares the performance achieved by a classical SpMV and by the multiplication of four vectors by four matrices using either standard C++ code and our optimized implementation. The classical SpMV computation reaches 14 Gflop/s. The standard C++ implementation reaches a performance of 38 Gflop/s while our optimized implementation almost reaches 140 Gflop/s. Notice that RCM ordering has no impact in the standard C++ implementation while it provides a significant improvement in our manually vectorized version. Thus, the C++ implementation is likely instruction-bound while our manually vectorized implementation is most likely memory-bound.

The main idea of this paper, computing 16 derivatives at a time, can be applied on different architectures such as multiprocessor multicore or GPU. To illustrate this fact, we conducted similar experiments on Cascade at the Minnesota Supercomputing Institute using two Intel Xeon E5 2670 nodes (8 cores, 2 threads and 20 Mbytes of L2 cache per core, with hyperthreading disabled.) The cores support the AVX instruction set. Both the compiler-generated code and the code written with the help of the AVX instruction set reach a read bandwidth of 85 GB/s. Using cache lines of 512 bits, we find a best/worst speeds of 2.4/20 Gflops/s and 32/120 Gflops/s, respectively for the 1vec/1mat and 4vec/4mat cases. Our implementation on Cascade reaches 45 Gflops for the 4/4 case, or 45/120=37% of peak theoretical performance on this particular processor setup. For reference, we achieve 140/210=66% of peak theoretical performance on the MIC. Note that peak performance is not that of the processor, but of the best possible algorithm on the processor. The gain in the 4/4 case with respect to 4x 1/1 case is 4x on the MIC and 2.5x on the front end, respectively.

6. CONCLUSION

In the previous sections, we have explored the practical implementation on an Intel Xeon Phi card of multiple derivative operators acting on multiple vectors within the context of RBF-FD. Each derivative has an associated sparse matrix with a fixed number of nonzeros on each row. While computing a single derivative of multiple functions is rather common, we accelerate the algorithm further by considering multiple matrices, with identical sparsity patterns, acting on multiple vectors. We specialize our study to four matrices and four derivatives, with 16 outputs, computed as a sum of outer products.

Our implementation makes use of the IMCI MIC instruction set and includes a number of swizzling and channel swapping operations, for an extremely efficient tensor product implementation. On a 96^3 3D grid, it reaches a performance of 140 Gflop/s which is 2.7 times faster than the best possible performance achievable for a single vector and a single derivative and more than 7 times faster than a best case practical 1/1 implementation.

In future work, we will examine the effect of larger stencil sizes and double precision. The presented performance does not reach the predicted peak and we believe that common techniques from SpMV such as cache partitioning can be successfully applied. We will also integrate the techniques presented in this paper within a fluid simulation using RBF-FD. We will also look into developing a similar technique on the Kepler family of GPUs, which offers swizzling and channeling operations.

Acknowledgment

Erlebacher acknowledges funding from NSF grant DMS-#0934331 (FSU), and the use of the MIC at FSU and at LCSE/U. Minnesota (provided by Prof. D. Yuen). NCAR is sponsored by NSF. Flyer acknowledges support of NSF grant DMS-0934317.

7. REFERENCES

[1] V. Bayona and M. Kindelan. Propagation of premixed laminar flames in 3D narrow open ducts using Rbf-generated finite differences. *Combustion Theory and Modelling*, 17:789–803, 2013.

[2] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, December 2008.

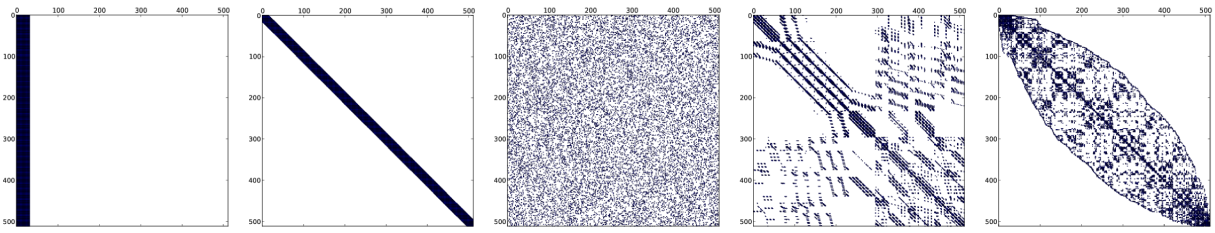
[3] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, 2003.

[4] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proc. of WWW*, 1998.

[5] A. Buluç, J. Fineman, M. Frigo, J. Gilbert, and C. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proc. SPAA '09*, pages 233–244, 2009.

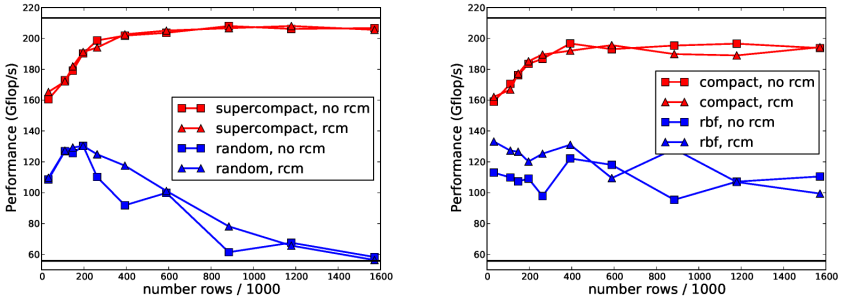
[6] A. Buluç, S. Williams, L. Oliker, and J. Demmel. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In *Proc. IPDPS*, 2011.

[7] P. P. Chinchapatnam, K. Djidjeli, P. B. Nair, and M. Tan. A compact RBF-FD based meshless



(a) Supercompact. (b) Compact. (c) Random. (d) 3D, no RCM. (e) 3D, RCM.

Figure 8: Different sparsity distributions. In all cases, there are 512 rows and 32 nonzeros per row. The last two matrices corresponds to a derivative stencil in 3D RBF-FD calculations (with and without RCM ordering).



(a) Supercompact and Random. (b) Compact and RBF-FD.

Figure 9: Performance of the manually vectorized code. The two horizontal lines depict the worst and the best predicted performances.

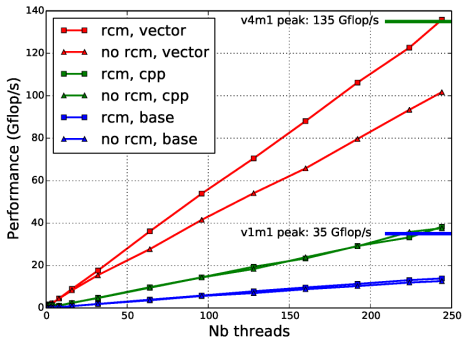


Figure 10: Performance of $y = Ax$ for a RBF derivative stencil of a 96^3 3D grid. Classical "one-matrix one-vector" SpMV ($n_m = n_v = 1$ given in blue) has low performance. The manually vectorized code (red) is 3.5 times faster than the compiler generated one (green). RCM ordering (squares) only improves the performance for our manually vectorized code. We also show peak theoretical performance for v1m1 and v4m1 on the MIC. Not shown is the peak theoretical performance of 210 Gflops for v4m4.

method for the incompressible Navier-Stokes equations. *J. Eng. Maritime Env.*, 223:275–290, 2009.

[8] T. Cramer, D. Schmidl, M. Klemm, and D. an Mey. Openmp programming on intel xeon phi coprocessors: An early performance comparison. In *Proc. of MARC*, November 2012.

[9] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proc. ACM national conference*, pages 157–172, 1969.

[10] N. Flyer, E. Lehto, S. Blaise, G. B. Wright, and A. St-Cyr. A guide to RBF-generated finite differences for nonlinear transport: Shallow water simulations on a sphere. *J. Comput. Phys*, 231:4078–4095, 2012.

[11] B. Fornberg and E. Lehto. Stabilization of RBF-generated finite difference methods for convective PDEs. *J. Comput. Phys.*, 230:2270–2285, 2011.

[12] Y. Koren. Drawing graphs by eigenvectors: Theory and practice. *Computers and Mathematics with Applications*, 49:1867–1888, 2005.

[13] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, A. Basermann, and A. Bishop. Sparse matrix-vector multiplication on gpgpu clusters: A new storage format and a scalable implementation. In *IPDPS Workshops*, pages 1696–1702, 2012.

[14] O. Küçüktunç, K. Kaya, E. Saule, and Ü. Çatalyürek. Fast recommendation on bibliographic networks with sparse-matrix ordering and partitioning. *Social Network Analysis and Mining*, 2013.

[15] M.K. Kumar. Accelerating sparse matrix kernels on graphics processing units. 2012.

[16] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey. Efficient sparse matrix-vector multiplication on x86-based many-core processors. In *Proc. of ICS*, 2013.

[17] J. Mellor-Crummey and J. Garvin. Optimizing sparse matrix-vector product computations using unroll and jam. *Int. J. High Perform. Comput. Appl.*, 18(2), May 2004.

[18] R. Nishtala, R. Vuduc, J. Demmel, and K. Yelick. When cache blocking of sparse matrix vector multiply works and why. *Appl. Algebra Eng., Commun. Comput.*, 18(3):297–311, May 2007.

[19] E. Saule, K. Kaya, and Ü. Çatalyürek. Performance evaluation of sparse matrix multiplication kernels on intel xeon phi. Technical Report arXiv1302.1078, 2013.

[20] C. Shu, H. Ding, and K. S. Yeo. Local radial basis function-based differential quadrature method and its application to solve two-dimensional incompressible Navier-Stokes equations. *Comput. Meth. Appl. Mech. Engrg.*, 192:941–954, 2003.

[21] D. Stevens, H. Power, M. Lees, and H. Morvan. The use of PDE centers in the local RBF hermitean method for 3D convective-diffusion problems. *J. Comput. Phys.*, 228:4606–4624, 2009.

[22] O. Temam and W. Jalby. Characterizing the behavior of sparse algorithms on caches. In *Proc. of SuperComputing*, pages 578–587, 1992.

[23] S. Toledo. Improving memory-system performance of sparse matrix-vector multiplication. In *PPSC*. SIAM, 1997.

[24] A. I. Tolstykh. On using RBF-based differencing formulas for unstructured and mixed structured-unstructured grid calculations. *Proc. of IMACS World Congress*, 228:4606–4624, 2000.

[25] A. I. Tolstykh and D. A. Shirobokov. On using radial basis functions in a "finite difference mode" with applications to elasticity problems. *Comput. Mech.*, 33:68–79, 2003.

[26] F. Vazquez, José-Jesús Fernández, and Ester M. Garzón. A new approach for sparse matrix vector product on nvidia gpus. *Concurrency and Computation: Practice and Experience*, 23(8):815–826, 2011.

[27] R. Vuduc, J. Demmel, and K. Yelic. OSKI: A library of automatically tuned sparse matrix kernels. In *Proc. SciDAC 2005, J. of Physics: Conference Series*, 2005.

[28] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proc. SC '07*, pages 38:1–38:12, 2007.

[29] G. B. Wright and B. Fornberg. Scattered node compact finite difference-type formulas generated from radial basis functions. *J. Comput. Phys.*, 212:99–123, 2006.