SAND2017-7039C

Revisiting Online Autotuning for Sparse-Matrix Vector Multiplication Kernels on Next-Generation Architectures

Simon Garcia De Gonzalo, Simon D. Hammond, Christian R. Trott, and Wen-Mei Hwu

***Abstract*—Sparse-Matrix Vector products (SpMV) are highly irregular computational kernels that can be found in a diverse collection of high-performance science applications. Performance for this important kernel is often highly correlated with the associated matrix sparsity, which, in turn, governs the computational granularity, and therefore, the efficiency of the memory system.**

In this paper, we propose to extend the current set of Kokkos profiling tools with an autotuner that can iterate over possible choices for thread-team size and vector width, taking advantage of runtime information, while, choosing the optimal parameters for a particular input. This approach allows an iterative application that calls the same kernel multiple times to continue to progress towards a solution while, at the same time, alleviating the burden from the application programmer of knowing details of the underlying hardware and accounting for variable inputs. We compare the autotuner approach against a fixed approach that attempts to use all the hardware resources all the time, and show that the optimal choice made by the autotuner is significantly different among the two latest classes of accelerator architectures. After 100 iterations we identify which subset of the matrices benefit from improved performance, while others are near the break-even point, where the overhead of the tool has been completely hidden. We highlight the properties of sparse matrices that can help determine when autotuning will be of benefit. Finally, we connect the overhead of the autotuner to specific sparsity patterns and hardware resources.

Keywords*—*Sparse-Matrix, Autotuning, Performance, Accelerators

I. Introduction

The current high performance computing (HPC) landscape has become increasingly diverse. High performance nodes now feature many distinct architectural characteristics that can significantly affect performance if they are not well utilized. These features range from the type and number of processing units to complex memory hierarchies and memory-media types. Within a processing unit there exists significant variation across vendor implementations with respect to the number of threads, vector units, vector widths, and cache hierarchies. The introduction of accelerator-type devices such as NVIDIA's GPUs and Intel's Xeon-Phi [10] has only exacerbated the differences. All of these architectural differences can pose challenges for code portability. In some cases, a significant tuning effort on the part of the application programmers is required to run on each different type of architecture, resulting in an unsustainable burden for maintaining multiple source codes, or at least multiple compilation configurations, for these hardware devices.

For these reasons, many mechanisms, such as OpenMP [4], OpenACC [17], and OpenCL [14], have been created to target a single source code that can execute on both multi-core processors and accelerator devices. Each has advantages and disadvantages over the others. The developer is forced to determine which mechanism will provide an application with the greatest performance across all possible hardware devices. In some cases, such a choice implies a vendor alignment. This has motivated programming models such as C++AMP [8], Kokkos [6], [7] and RAJA [9] that seek to provide parallel abstractions as part of the C++ language. These abstractions in turn allow the complexity and variety in hardware to be hidden from the application programmer enabling greater levels of performance portability and reduced vendor lock-in from a single application source.

In this paper, we extend the baseline Kokkos programming model with an online tuner that is able to gradually adjust the available performance knobs exposed in the Kokkos runtime. We evaluate the autotuner on the latest classes of HPC devices – NVIDIA's Pascal GP100 GPU, and Intel's Knights Landing (KNL) architectures. In so doing we make the following contributions:

- **Re-evaluation of Autotuning SpMV Methods on Contemporary Systems** - we provide an overview on the relative benefits of using autotuning for SpMV on bleeding-edge computing hardware found in several of the next-generation of high-performance systems;
- **Analysis of Autotuning Performance Improvement** - Applied performance analysis of on our dynamic autotuner to understand what effects the choices may have during computation. We have employed best-of-class vendor supplied performance tools to improve accuracy and broaden the range of information captured;
- **Identification of Important Hardware Restrictions** - We expose important hardware resource/design characteristics and resulting limits that can significantly affect SpMV performance that may be used to help optimize static kernel parameters in situations where iterative methods may not be present or overheads from autotuning may be too expensive to warrant their use.

This paper is laid out as follows: Section II provides the reader with background information on the Kokkos programming model; Section III provides an overview of our autotuning tool; in Section IV we provide performance results from using our tool for tuning SpMV calls and, finally, we provide conclusions in Section V.

II. Background

A. Kokkos

The Kokkos programming model provides the application programmer the means to abstract their code from the finer intricacies of hardware details, and at the same time maintain portability across hardware devices. Kokkos accomplishes these goals by providing abstractions for the memory space (*how* and *where* is data allocated), and execution space (*where* parallel patterns are executed). Parallel patterns in Kokkos form part of the API and are defined as a simple set of patterns such as parallel-for, -reduce, and -scan that can be nested within each other to form more complex schemes. An example of nested patterns is illustrated in lines 19 to 32 of listing 1.

Kokkos works on top of an abstract machine model, which assumes multiple types and numbers of computing units within a node. Each compute unit comes with one or more memory spaces optimized for that unit. The execution space represents a group of homogeneous units within the machine model. These units are used to execute a parallel pattern. The programmer can then create multiple instances of execution spaces that target different units. The process of compiling and running computational kernels on these different spaces is abstracted completely by Kokkos. This ability largely removes the need for hardware-specific optimizations to be part of the main code base although these can be added by the programmer and interoperated with where desired. Kokkos can provide further tuning through parameters passed to the Kokkos API. We use these parameters for the autotuner proposed in this paper. The memory spaces follow the same logic as the execution spaces. Different instances can represent different types of memory within a node, such as conventional DDR, on-package High-Bandwidth Memory (HBM), as well as future devices such as non-volatile memory. For accelerators, a memory instance can be used to describe various types of memory such as shared, texture, and global memory.

B. Sparse-Matrix Vector Multiplication (SpMV)

Sparse matrix-vector multiplication (SpMV), defined as $y = Ax$ where A is a sparse input matrix, x is a dense input vector, and y is the vector product, is a widely used computational kernel found in many scientific applications. The performance of this operation is of great importance for iterative linear solvers since, SpMV can potentially be called up to many thousands of times until solution convergence. Due to its importance there exists a large variety of implementations and extensive literature dedicated to it. Unlike dense matrix-vector operations, SpMV has to deal with a broad range of irregularity, in particular regarding the sparsity of the matrix. This asymmetry makes it difficult to develop general-purpose, high-performance solutions. Implementations often struggle to make successful use of caches, memory hierarchies, and available computational resources. An extensive summary of approaches by Vuduc [16] is a good overview of this area. Using Kokkos to write a portable high-performing SpMV kernel, we have to deal with the same obstacles. Tuning parameters have to take into account the hardware as well as the matrix characteristics. By making use of KokkosP ("Kokkos-Profiling") runtime performance hooks that are built directly into the Kokkos-runtime, we develop an autotuning framework that can use runtime timing information to search for the best performing set of parameters for a particular matrix on a particular hardware device. A skeleton implementation of SpMV using Kokkos can be seen in listing 1. In this implementation we utilize the standard Compressed-Row Storage (CRS) format since this is the most commonly occurring format in the Trilinos linear solver portfolio (where Kokkos is heavily used to implement next-generation solver capabilities). The algorithm is defined within a functor from line 19 to 32. It uses a series of nested parallel patterns in lines 19 and 22 to represent the behavior of the SpMV abstractly. This functor is instantiated in line 5 and used as the last parameter for the parallel pattern in line 11. This pattern breaks the problem size into groups of team threads and determines vector widths.

TABLE I: Search-Space bounds per architecture: The limits for each device were determined by running a large number of SPMV problems. The resulting bounds nicely match hardware characteristics of each device. It is important to note that the search space for the KNL is about 3 times larger than the Pascal search space

Architecture	Max Team Size	Min Team Size	Max Vector Width	Min Vector Width
Pascal	1024	32	32	1
Knights Landing	2048	1	16	1

Listing 1: Kokkos SPMV skeleton code

```
int team_size;
int vector_length;

//instantiate SPMV functor
SPMV_Functor<...> func (...);

//registration of the team_size and vector_length parameters
Kokkos::Profiling::autoTune(&team_size,&vector_length);

//These parameters are use by the Kokkos TeamPolicy to map to hardware
Kokkos::parallel_for("SPMV",
                     Kokkos::TeamPolicy <Kokkos::Schedule<Kokkos::Dynamic>>
                                (league_size,team_size,vector_length),func);
// ...
struct SPMV_Functor {
// ...
  operator() (const team_member& dev) const
  {
    Kokkos::parallel_for(Kokkos::TeamThreadRange(dev,0,rows_per_team),[=](...){
      // ...
      //perform vector reduction
      Kokkos::parallel_reduce(Kokkos::ThreadVectorRange(dev,row_length),[=](..){
        // ...
        lsum += ...;
      },sum);

      // Add the results once per thread
      Kokkos::single(Kokkos::PerThread(dev), [&] () {
          // ...
          m_y(iRow) = sum ;
      });
    });
  }
//...
}
```

III. Autotuner

The autotuner is implemented using the KokkosP performance tools interface. This is an experimental runtime hooks interface which is able to connect deep into the Kokkos runtime to receive rich notifications on the start of parallel execution, data-structure

allocation, migration and other important application events. An important characteristic of KokkosP is that it maintains an identical event model across the architectures and backends supported by Kokkos enabling portability even for performance analysis tools, and, in the case of this paper, dynamic autotuners. In recent internal research prototypes the hooks have also been demonstrated as a mechanism for provided enhanced information on application behavior for third party tools such as Intel's VTune Analyzer XE [11] and NVIDIAs's NSight [1] GPU profiler. We utilize the connectivity to Intel's VTune profiling tools later in this work.

Each of the tools written to utilize KokkosP are dynamically loaded and can be enabled by defining an environment variable. The autotuner was developed within the same profiling framework but with the difference that it can run as a dynamic tool that uses runtime information provided by the hooks as feedback to improve the parameters that are usually left for the application scientists to determine. We note the significant utility of being able to *dynamically* load autotuners into Kokkos applications, opening the door for application, or even, problem specific tuners to be utilized by our complex solver libraries. The autotuner consists of two main components, the parameter registration interface, and the search space iterator.

A. Tuning Parameter Registration Interface

For the autotuner to be able to modify the parameters that are fed into the existing Kokkos API, the user must first register the tunable parameters using the registration interface designed for the autotuner. An example of the registration interface can be seen in line 8 of code listing 1. Registration works by keeping track of the pointers to those variables and updating them with new values as improved choices become available. These variables are internally associated with a parallel pattern label (in this case, the parallel_for label, "SPMV" in line 11 of the same code listing). The association is formed by storing registered parameters and mapping these to a parallel pattern the moment it is first executed. This implies that variable registration should follow a particular order. Registration must be followed by a parallel pattern. If, for example, multiple pairs of parameters are registered in a row, only the last registration will be associated when the next parallel pattern is executed. Because the registration works by the aliasing of pointers and not any data copying, it has a negligible overhead.

B. Search Space Iterator

Once a set of parameters is registered and associated with a particular parallel pattern label, the search space iterator (SPI) is responsible for keeping track of the currently selected parameter choices. In this case study we modify the "team size" and "vector width" parameters which adjust the mapping of parallelism onto the hardware platforms being used (see [15] for discussion of how these are utilized by Kokkos). The SPI is also responsible for iterating through the possible combinations of parameters for any specific piece of hardware. Each hardware platform has a different search space, which relates directly to the architecture features of a device. The search space for the architectures used in this paper are illustrated in Table I. The limits for each device were determined by running a large selection of SpMV problems with different sparsity and structure characteristics. We then picked the highest and lowest optimal parameters. Results confirm the close relationship between performance and hardware features. For the Pascal architecture team sizes should not go below 32 due to the warp size execution being an integral part of GPU architecture. No optimal combination of team size and vector width will go over 1,024 due to hardware limits. For Knights Landing (KNL) [12], vector size should not exceed 16 since no more vector lanes are available to use. In this case each processor core has two 512-bit vectors that can handle up to eight double-precision elements each. The team size is limited by hardware resources. If we utilize all hyper-threads on the KNL by using all 72 cores and the max four hyper threads per core and fully use the vector lanes to execute 16 elements we get 2,304, which is slightly above 2,048.

The SPI is triggered whenever the same parallel pattern label is encountered. When executing a parallel pattern, the autotuner maintains a map of all pre-existing labels (and associated registered tuning parameters). Each parallel pattern label record stores the kernel's name, number of times it has been called, and total execution time. We further extend the information tracked to include data about the current and best team size and vector width. Thus, upon finding a matching label, the autotuner will trigger the SPI by updating the record.

We implemented the SPI using a naive approach of iterating over all possible options by increasing the vector size by multiplying by two until the maximum vector length is reached. At this point we switch to increase the team size by multiplying by two and reset the vector length to the minimum size. The algorithm increases the vector size and the team size by two due to hardware features such as vector widths and number of hyper-thread being designed to be powers of two. When the SPI is activated it will record the current set of parameters and the corresponding execution time. It will then check if the current execution time is faster than the current fastest set of parameters. If so, it will update the current best team size and best vector width accordingly. Finally, the SPI will update the current set of parameters to reflect the next set in the search space to be used when encountering this parallel pattern again. When all the possibilities in the search space have been attempted, the SPI will default to the best set of parameters and turn itself off to prevent continuous auto-tuning overhead interfering with the full application. This approach hinges on iterative applications that call the same computational kernels multiple times during execution of the program, where each time the SPI will attempt a new set of parameters. Kernels that are only executed a few times during the lifetime of an application will not be able to take advantage of this system.

C. Non-Zero (NNZ) Count Hint

In many cases, not all iterations of a search space are worth exploring due to the characteristics of a particular matrix. If a matrix has a low number of non-zeros per row, we may choose to skip some of the search space iterations that we

suspect won't result in improved performance by taking that information into account. Some of the vector width choices in the search space can be discarded if we know there are not enough non-zero elements in a row to take advantage of all the vector lanes. Thus we can shrink the search space, and therefore reach the optimal configuration faster, by adding a check that will limit the SPI to only iterate through parameters that can meet the above criteria. We accomplish this by adding an extra optional parameter to the autotuner that will register the number of non-zero elements per row that will later be used by the SPI for shrinking the iteration space. We call this extra optional parameter that represents the number of non-zero elements per row hint NNZ.

IV. Results

In this section, we present an evaluation of the effectiveness of the autotuner tool with and without the number of non-zeros per row hint. We compare it with two other schemes: the "Fixed" approach, and the "Oracle". The Fixed approach stands for the conservative choice made by an application programmer that is likely going to use most of the available resources of the hardware without taking into account any matrix characteristics. We define the fixed parameters as {team size, vector width} of {4, 8} for the KNL devices and {32, 32} for GPUs. The Oracle will always use the best configuration from the start. We evaluate these schemes using 22 sparse matrix-vector problems taken from the University of Florida Sparse Matrix Collection [5], which contains a set of real-world matrices of different sizes and sparsity patterns from a wide variety of domains. Table II shows a description of the matrices used. We show that after 100 iterations, the autotuner is useful in finding the most optimal setup, in some instances incurring some significant overhead when compared with the Fixed approach and in some cases substantially outperforming it. Furthermore, a clear difference between architecture can be observed, as some architectures are more sensitive to suboptimal choices than others.

A. Experimental Setup

All experiments were run on the two proposed hardware systems – a Tesla P100-SXM2 GPU part of the NVIDIA Pascal architecture, and Intel's Xeon-Phi Knights Landing self-hosted processor. Table III summarizes these architectures in more detail. For runs on the Pascal GPU we utilize the CUDA 8.0.44 toolkit and GCC 5.4.0 as the host compiler. On the KNL we configure the SpMV kernels to be compiled with Intel's 17 Update 1 compiler with flags enabled to generate optimized KNL-specific (AVX512) instructions. In both hardware environments RedHat Enterprise Linux 7.3 is used. For the Intel KNL we also explore different configurations that are possible with this architecture. We chose from a number of different clustering styles and memory type partitions since its High-Bandwidth Memory can be used in "flat" (as a memory for which direct allocation can occur) as well as "cache" where the hardware utilizes the memory to cache data from the DDR4 memory pool. We ran on seven different setups that are possible from the many combinations and show results for the best performing configuration: flat.

TABLE II: Attributes of each of the matrices that were used. The number in parenthesis after the name is the average number of non-zero elements per row. The rest of the table shows the number of rows, columns and the total number of non-zero elements in the matrix.

Matrix Name	Number of Rows	Number of Columns	Non-Zeros
olm2000(3)	2,000	2,000	7,996
relat9(3)	12,360,060	549,336	38,955,420
mc2depi(3)	525,825	525,825	2,100,225
lnsp131(4)	131	131	536
memship(5)	2,707,524	2,707,524	14,810,202
lnsp511(5)	511	511	2,796
atmosmodl(6)	1,489,752	1,489,752	10,319,760
dc1(6)	116,835	116,835	766,396
fullship(8)	2,987,012	2,987,012	26,621,990
ex18(12)	5,773	5,773	71,805
cage10(13)	11,397	11,397	150,645
coater1(14)	1,348	1,348	19,457
cage13(16)	445,315	445,315	7,479,343
cage14(18)	1,505,785	1,505,785	27,130,349
coater2(21)	9,540	9,540	207,308
ex25(29)	848	848	24,612
rma10(50)	46,835	46,835	2,374,001
invextr1(58)	30,412	30,412	1,793,881
raefsky3(70)	21,200	21,200	1,488,768
ns3Da(82)	20,414	20,414	1,679,599
RM07R(98)	381,689	381,689	37,464,962
fluorem(140)	2,017,169	2,017,169	283,073,458

For the KNL-Flat configuration, the HBM memory extends the physical address space with the DDR4 memory. We ran in this configuration with the HBM used as both the default and non-default memory (*i.e.* using DDR4) to study the effects of the HBM on the autotuner and overall performance. Finally, for all KNL-Flat configurations we ran with two environment variables. `OMP_NUM_THREADS=256`, and `KMP_AFFINITY=compact`, to control the number of threads and placement. The results are labeled to indicate the memory partition used – either DDR4 or HBM. All results are from executing the SpMV kernel for 100 iterations which, anecdotally, represent a reasonable number of calls when the kernel is used in a more complex solver.

B. Experimental Results

1) NVIDIA P100 Pascal GPU: The normalized performance of the autotuner on the NVIDIA P100-SXM2 can be observed in Figure 1. These results are sorted based on the sparsity pattern of each test matrix, from very sparse on the left to increasing density on the right. The number of non-zero elements per row is shown in parentheses next to the each matrix name.

TABLE III: Table of Architecture Details [2][13]

Architecture	Cores	Logical Cores	Frequency	GFLOPs (double)	Max. Memory	Max. Memory B/W
NVIDIA Tesla P100-SXM2 GPU (Pascal)	56	3584	1328 MHz	5,427	16 GB (HBM2)	732 GB/s
Intel Xeon Phi (Knight's Landing)	72	288	1.5 GHz	3,543	384 GB (DDR4) 16 GB (HBM)	90 GB/s (DDR4) 400 GB/s (HBM)

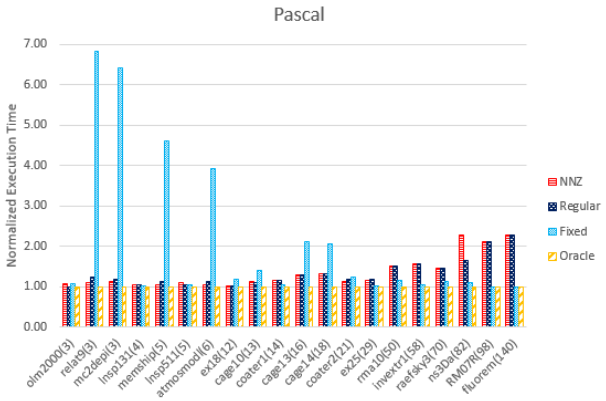

Fig. 1: Results for the NVIDIA P100-SXM2 Pascal architecture (lower is better). All results are normalized based on the Oracle. The autotuner strategies are explained in the first paragraph of section IV

We can immediately notice a clear distinction based on the above description. Matrices that have a low number of non-zero elements per row perform very poorly when using a fixed size for the thread teams and vector width. This performance can be explained due to there not being enough uniform work per row take full advantage of a large degree of threads per team. It is important to note, based on Table II, that matrices that have this behavior like `relat9`, `mc2depi`, `memship` and `atmosmodl` are vast matrices containing well over a million elements. It is well known that large irregular data sets perform poorly on the GPU due to divergence as shown in [3]. By forcing a large team of threads, it enforces a significant divergence penalty. For other highly sparse but far smaller matrices like `olm2000`, `lnsp131`, `lnsp511`, and `dc1` that only have a few thousand elements, by choosing a larger size team of threads and vector size, the SPI will not incur significant amounts of divergence penalty. Thus the Fixed choice performs almost optimally when compared to the Oracle. When we analyze the effectiveness of the autotuner for the large matrices with a low number of non-zero elements per row, 100 iterations are enough to make the online autotuner worth using. It performs much better than the Fixed choice, the benefit in performance ranging from 3.5x to 5X faster, and only 12% to 23% slower than the Oracle. The improvement for these type of matrices from the Regular approach to the NNZ approach varies between 11% in the best case to only about 5% in the worst case. For small matrices with a low number of non-zeros per row, there is essentially no benefit from the NNZ hint, and in many cases it will hinder performance. The reason for this drop in performance is due to the optimal choice for vector width being larger than the number of non-zeros per row. Thus the NNZ approach never reaches the most optimal setup. As an example, we look more deeply into the different configurations chosen for `olm2000` and `lnsp511`. For `olm2000` the number of non-zero elements per row is three. The NNZ approach limits the vector width to two due to a vector width of four being larger than the number of elements per row. When we ran the Regular approach, which searches the entire search space, the most optimal vector width was determined to be four. Thus, the Regular approach performs 2.4X faster than the NNZ approach.

When the number of elements per row increases to the beyond 10, we see a slightly different result. In all cases, the autotuner performs only marginally better than the Fixed choice. The performance benefits range from 17% to 64%. When compared to the Oracle, the autotuner performs only 1% to 30% slower. With small matrices suffering almost no overhead and large matrices with millions of elements reaching up to 30%. What this means is that 100 iterations are just enough to amortize the overhead of the tool to at least break even regarding performance compared to the fixed choice. For some of these matrices, such as `cage13` and `cage14` the differences between the Fixed and the Oracle are as high as 2X. In other words, once beyond the break-even point, because the difference between the Fixed and Oracle is significant, it would only take a few more iterations to make a noticeable improvement in performance using the autotuner.

Finally, when the matrix row density increases, we see that the difference in performance between the autotuner and the Fixed approach worsen, and the difference between the Fixed approach and the Oracle shrinks substantially. This behavior is also explained by warp divergence. Because the number of non-zeros per row reaches and surpasses the size of a CUDA warp, 32, the penalty of warp divergence is no longer an issue. What this means is that, by around 50 non-zero elements per row, it would require a substantial increase in iterations to break even against the fixed approach, and after breaking even, it would take a further increase in iterations for the autotuner to make a significant difference. We also notice that the autotuner starts to perform worse compared to the Oracle when the density of non-zero elements per row increases. Meaning that a suboptimal choice of thread size and vector with by the autotuner has a larger penalty. A more sophisticated autotuning tool would disable itself based on this information. We plan to use all of these observations to improve the performance of the autotuner tool further by taking into account more properties of the matrix such as total size and the total number of iterations

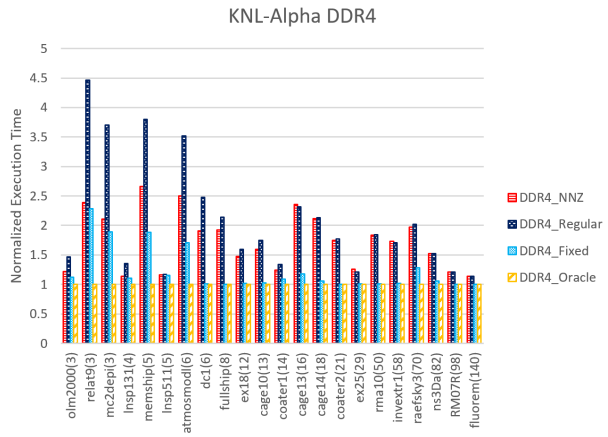

Fig. 2: Results for the Knights Landing architecture using the KNL-Flat DDR4-only Memory configuration (lower is better). All results are normalized based on the Oracle. The autotuner strategies are explained in the first paragraph of section IV

in the simulation when possible.

2) Knights Landing KNL-Flat: The results for the Knights Landing architecture using the KNL-Flat configuration are shown in Figure 2 and 3. These results are presented in the same fashion as previous results. Unlike the GPU results, this figure shows results for two different execution modes. The first mode shown in Figure 2, uses the standard DDR4-only memory in the KNL. This mode also uses two environment variables to control the number of threads and placement. The second mode, HBM, shown in Figure 3 is similar to the DDR4 mode with the only difference being the type of memory specified to the KNL changed to High Bandwidth Memory (*i.e., this runs entirely in the high-bandwidth resource*). The Fixed approached was defined to be four threads per team with a vector width of eight. This setup was chosen because it maximizes the available hyper threads per core available in the KNL, and when all threads are vectorizing, they will adequately fill the available vector resources.

The performance of the autotuner can be classified into three different groups, A, B, and C. When using the autotuner, the worst performing group of matrices is group A. This group is composed of `relat9`, `mc2depti`, `memchip`, `atmosmodl`, `dc1`, `fullchip`, `cage13`, and `cage14`. Looking at Table II, we can characterize this group as having large sparse matrices, with a large number of rows and columns but very few non-zero elements. For these type of matrices, the autotuner Regular scheme performs between 2.1X to 4.4X slower than the Oracle. The NNZ approach can trim the search space, and lower this difference to between 1.9X to 2.6X. We can explain the performance of the autotuner through cache behavior. For `relat9`, the optimal combination of threads per team and vector width is $\{1, 1\}$. This choice is excellent for large sparse matrices because the few non-zero elements are not contiguous and do not benefit out of a large number of threads in a team. A representative example of

TABLE IV: KNL-Flat performance and cache metrics for relat9 using different configurations: Ideal CPI of KNL is 2. SIMD compute metrics report the ratio of SIMD compute instructions to the total number of memory loads

Execution Type	CPI	L2 Hit rate	SIMD compute to L1 access	SIMD compute to L2 access
Oracle {1, 1}	3.9	0.687	0.451	37.838
Fixed {4, 8}	4.0	0.736	0.307	31.375
Fixed {256, 16}	8.9	0.565	0.065	11.847

group A is `atmosmodl`. Using a larger number of threads and a large vector width causes cache trashing and cache misses. In Table IV, we showed the results of running Intel's VTune profiling tool and compared the Cycles Per Instruction (CPI) Rate, L2 Hit Rate, SIMD compute-to-L1 access, and SIMD compute-to-L2 access for the Oracle {1, 1}, standard Fixed {4, 16}, and an alternative Fixed {256,16} for the relat9 matrix that confirms our reasoning. The ideal CPI rate for KNL is 2, The Oracle{1, 1} manage to achieves a CPI of 3.9 compared to a CPI of 8.9 when using a Fixed {265, 16}. A high CPI value usually indicates latency in the system that could be improved. The L2 Hit Rate, which measures the ratio of requests that hit the L2 to the total number of request serviced, is better in Oracle as well. Perhaps the most significant penalty suffered by a large number of threads per team is the poor SIMD compute-to-L1 access. This metric provides the ratio of SIMD compute instructions to the total number of memory loads to the L1 cache. It is important for the KNL that this ratio is large to ensure the efficient usage of computing resources. The difference between the Oracle and a Fixed {256,16} is of 7X. There is also a difference of 3X for the similar metric of SIMD compute-to-L2 access. The autotuner Regular approach by attempting larger threads and vectors sizes suffer from substantial overhead penalties when iterating through the entire search space. The NNZ approach performed better by avoiding many of the expensive wrong choices. Finally, the Fixed approach of {4, 8} that we use as the standard choice for most matrices performs 2.2X slower than the Oracle for `realt9`, very close in performance to the NNZ approach. We can conclude that for this group of matrices the autotuner is not beneficial, and a smart heuristic should disable the autotuner based on the matrix characteristics.

Group B is characterized as having far smaller and slightly denser matrices than group A, with a larger number of non-zero elements per row. Examples of these are `ex18`, `cage10`, `coater2`, `rma10`, `invextr1`, `raefsky3` and `ns3Da`. For these type of matrices, the autotuner performs about 1.5X to 2.0X slower for the Regular, and 1.4X to 1.9X for NNZ when compared to the Oracle. The Fixed approach {4, 8}, performs only about 5% slower than the Oracle in general. Because this group of matrices is made up of smaller matrices that are more densely packed, there is a higher chance that better performance could be achieved by the use of more threads per team or wider vector length. One such matrix is `cage10` [5]. We looked at the top three configurations chosen by the autotuner for this

TABLE V: Memory access metrics for large sparse matrices with and without HBM enabled. DRAM Bandwidth Bound is the percentage of time system spend on DRAM access.

Matrix	Memory Mode	DRAM Bandwidth Bound	Bandwidth Utilization
RM07R	DDR4	29.9%	83 GB/s
RM07R	HBM	0%	167 GB/s
fluorem	DDR4	45%	86 GB/s
fluorem	HBM	0%	207 GB/s

matrix. We see that the set of team size and vector width chosen where {4, 1}, {4, 2}, and {4, 8} respectively. The difference in performance between the top three choices is on the scale of hundredth of a second. In fact, this small difference continues to be true for the top 10 choices. What this means for the autotuner is that a suboptimal choice does not incur such large penalty as group A. The Fixed {4, 8} happens to be among the best options for these type of matrix. We can conclude that for this group, 100 iterations is not enough to offset the overhead of online tuning. A conservative Fixed choice that attempts to use the available resources in the hardware will perform almost optimally.

For group C, the autotuner performs with the lowest overhead and closer to the Oracle performance. The Regular approach has about 13% to 45% performance loss compared to the Oracle, and the NNZ approach has about 13% to 26% loss. To explain the performance of the autotuner, we have to subdivide this group into two sub-categories C1 and C2. The first subgroup, C1, is composed of small, sparse matrices such as `olm2000`, `lnsp131`, `lnsp511`, `coater1`, and `ex25`. Some of these matrices have only a few hundred or few thousand non-zero elements. Similarly to Group B, due to the small size of these matrices, the few non-zero elements have a higher probability of being clustered in contiguous memory locations, benefiting from a larger number of threads per team and wider vector width. The only difference between this subgroup and group B is that subgroup C1 has far smaller matrices. The Fixed approach performs very slightly worse than Group B, averaging about 10% slower when compared to the Oracle due to misuse of resources. The Oracle choice for `lnsp131` is {16, 2}, taking advantage of a higher number of threads than the Fixed approach but also reducing the vector width. The second subgroup, C2, is composed of two very dense matrices having more than a 30 million non-zero elements such as `RM07R`, and `fluorem`. The bandwidth explains the performance for the autotuner and Fixed approach. These matrices do not fit in cache, and the bottleneck becomes more dependent on the bandwidth to the higher-level memory subsystem. Thus the SpMV kernel becomes bandwidth bound compare to compute bound. Choosing the best configuration to maximize compute resources becomes less necessary to the overall performance. The autotuner suboptimal choices suffer from the least overhead penalties. Figure V shows bandwidth profiling metrics taken using Intel Vtunes for these two matrices with and without HBM enabled.

The results for KNL-Flat using the HBM memory mode is

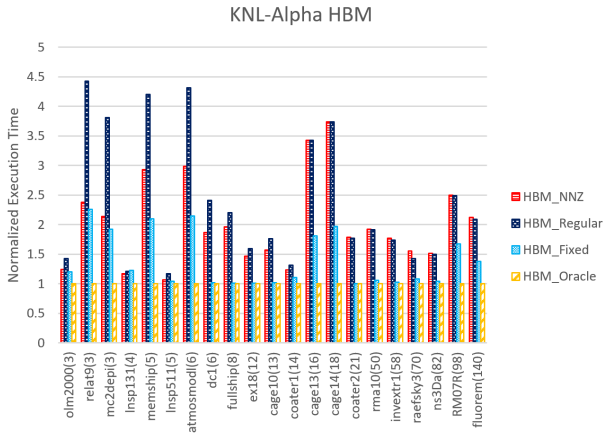

Fig. 3: Results for the Knights Landing architecture using the Flat HBM memory configuration, (lower is better). All results are normalized based on the Oracle. The autotuner strategies are explained in the first paragraph of Section IV

shown in Figure 3. The behavior of the autotuner and Fixed approach behave very similarly as the DDR4 mode except for four matrices. The four matrices that behave differently are `cage13`, `cage14`, `RM07R`, and `fluorem`. These matrices can be characterized as vast and dense enough to be affected by the bandwidth to the memory subsystem. It is exactly for this kind of matrix that the type of memory we are using is of significance. If we look at Table III we can see the difference in peak bandwidth between the DDR4 and HBM memory in the KNL. When the performance of a matrix is bottlenecked primarily by the bandwidth then choosing suboptimal configurations of threads and vector width won't suffer from a significant penalty. Because the Bandwith drastically increases from 90GB/s to 400GB/s for the HBM mode, then the bottleneck of the performance goes back to compute. In this mode, choosing a suboptimal configuration that does not map well to the available resources will suffer from significantly performance penalty. This explains the performance deterioration for the autotuner and Fixed approach for those four matrices when switching between DDR4 and HBM memory mode. Finally, we compare the performance of the Oracle between the DDR4 and HBM memory. We see that as stated above the matrices that depend heavily on the bandwidth perform much better under the HBM. For the two largest and matrices the performance improvement between DDR4 and HBM matches the ratio of bandwidth between the two memory modes, about 4X. What this means for the autotuner is that taking in consideration the overall bandwidth of the system as well as the density and size of the matrix could be of value.

C. Autotuner selection

To emphasize the importance of using an autotuner to find the most optimal set of parameters we present Table VI. In this table, we have picked a subset of the matrices

TABLE VI: Best Performing Selection by the Autotuner

Matrix	# Of Non Zeros Per-row	Pascal		KNL DDR4		KNL HBM	
		Team Size	Vector Width	Team Size	Vector Width	Team Size	Vector Width
relat9	3	128	2	1	1	1	2
lnsp131	4	32	32	16	2	16	1
atmosmodl	6	64	4	1	4	1	1
fullship	8	32	32	4	1	4	4
cage10	13	32	4	4	16	2	8
coater1	14	32	8	16	2	2048	8
coater2	21	64	8	2048	16	4	2
rma10	50	32	16	4	4	2	4
raefsky	70	32	16	2	8	4	1
fluorem	140	32	32	8	16	1	2

used for out experiments, and for each matrix we show the most optimal team size and vector width for each architecture and configuration. What we can clearly see from the table is that each architecture had a substantially different set of optimal parameters. Even the two KNL configurations, which as expected have the most similar set of parameters, have the same parameters less than half the time. What this would mean is that for the most optimal application performance, the programmer would have to re-tune his or her application when porting it between configurations of the same architecture. Lastly, the Pascal's optimal parameters show in GPU's preference for a large team size when compared to the KNL type devices. We also see that the vector width of the Pascal, and KNL Flat using only the DDR4 memory increases as the as the number of non-zeros per row increases, showing how the autotuner adapts to matrix features. The trend is not particularly true for KNL Flat with HBM enable as it may depend more heavily on other matrix features such as matrix size. This means that portability across and within architectures has to be done by taking into account small hardware details and data characteristics, highlighting the need for automated adaptive runtime tools.

V. Conclusion

The broad range of optimal parameters found during our experiments shows an uneasy problem – that in order to obtain the best performing SpMV operations, each instance of the kernel may require considerable tailoring. We propose to extend the current set of KokkosP-based performance tools with an autotuner that iterates over possible candidate parameters that are fed to generic Kokkos parallel patterns. In this paper we have shown that for the Pascal architecture, after about 100 iterations, a near optimal choice can be made by the autotuner that is, in many cases, beneficial to the overall performance. This behaviour, was most commonly encountered with very large sparse matrices. For KNL, we showed that in most cases the autotuner suffers from substantial overhead. Only when the matrix was of a very small dimensions did the autotuner perform on par with the Fixed approach. This outcome reflects experience that the basic configuration of parallel kernel execution is much more important on the GPU where poorly tuned parameters can have a significant effect on execution times. This strongly motivates the need to have simple, low overhead methods for tuning dynamic properties of the execution at runtime. For both architectues we have identified matrix characteristics that affect the performance of the autotuners. Finally, this paper highlights the need for automated tools by presenting the large variability in optimal parameters among and within different architectures for the same and different matrices.

Acknowledgment

Sandia National Laboratories is a multi mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energys National Nuclear Security Administration under contract DE-NA0003525.

We are grateful for the help and support of our colleagues and system administrators in Sandia's NNSA/ASC Advanced Architecture Testbed project.

References

[1] NVIDIA NSight 3.0 User Guide.

[2] NVIDIA CUDA Programming Guide, 5 March 2015, 2015.

[3] N. Bell and M. Garland. Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 18. ACM, 2009.

[4] L. Dagum and R. Menon. OpenMP: an Industry Standard Api for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[5] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[6] H. C. Edwards, D. Sunderland, V. Porter, C. Amsler, and S. Mish. Manycore Performance-Portability: Kokkos Multidimensional Array Library. *Scientific Programming*, 20(2):89–114, 2012.

[7] H. C. Edwards, C. R. Trott, and D. Sunderland. Kokkos: Enabling Manycore Performance Portability through Polymorphic Memory Access Patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014.

[8] K. Gregory and A. Miller. C++ AMP: Accelerated Massive Parallelism with Microsoft Visual C++. 2014.

[9] R. D. Hornung and J. A. Keasler. The RAJA Poratability Layer: Overview and Status. Technical Report LLNL-TR-661403, Lawrence Livermore National Laboratory, CA, USA, 2014.

[10] J. Jeffers and J. Reinders. *Intel Xeon Phi Coprocessor High-Performance Programming*. Newnes, 2013.

[11] J. Reinders. *VTune Performance Analyzer Essentials*. Intel Press, 2005.

[12] A. Sodani. Knights landing (KNL): 2nd Generation Intel Xeon Phi Processor. In *Hot Chips 27 Symposium (HCS), 2015 IEEE*, pages 1–24. IEEE, 2015.

[13] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu. Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro*, 36(2):34–46, 2016.

[14] J. E. Stone, D. Gohara, and G. Shi. OpenCL: a Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in science & engineering*, 12(1-3):66–73, 2010.

[15] C. R. Trott, M. Hoemmen, S. D. Hammond, and H. C. Edwards. Kokkos: the Programming Guide. 2015.

[16] R. W. Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, University of California, Berkeley, 2003.

[17] S. Wienke, P. Springer, C. Terboven, and D. an Mey. OpenACC First Experiences with Real-World Applications. In *European Conference on Parallel Processing*, pages 859–870. Springer, 2012.