A Study of Sparse-Matrix Vector Multiplication (SpMV) on Different Architectures and Libraries

Naveen Anand Subramaniam, Omkar Deshmukh, Vennila Megavannan, Dan Negrut

July 6, 2015
Abstract

With the advent of parallel processing architectures and a steep increase in parallelism found among the recent applications, GPGPUs have gained attention with respect to their importance in the execution of these applications. In this document, we specifically analyze Sparse-Matrix Vector Multiplication (SPMV) across different architectures, libraries and matrix formats. The experimental platforms include but are not limited to GTX770, Tesla K40c, Tesla K20Xm while the different libraries we have used are CUSP, cuSPARSE, VexCL, ViennaCL and MKL. The purpose of this effort is to identify several trade-offs with respect to architectures and libraries while also accounting for density, size and number of non-zeros (NNZ) of the sparse matrices we worked with.

Keywords: SPMV, GPU, CUDA, OpenCL, cuSPARSE, MKL, VexCL
Contents

1 Introduction 3

2 Experimental Setup 3

3 Results 4
   3.1 CUSP ......................................................... 4
   3.2 ViennaCL ..................................................... 4
   3.3 cuSPARSE .................................................... 5
   3.4 VexCL ......................................................... 5
   3.5 MKL ......................................................... 5

4 Discussion 5
1 Introduction

ViennaCL is a free open-source linear algebra library for computations on many-core architectures (GPUs, MIC) and multi-core CPUs. The library is written in C++ and supports CUDA, OpenCL, and OpenMP. In addition to core functionality and many other features including BLAS level 1-3 support and iterative solvers, the latest release family ViennaCL 1.6.x provides fast pipelined iterative solvers including fast sparse matrix-vector products based on CSR-adaptive, a new fully HTML-based documentation, and a new sparse matrix type. Also, a Python wrapper named PyViennaCL is available [4].

The cuSPARSE library contains a set of basic linear algebra subroutines used for handling sparse matrices. It is implemented on top of the NVIDIA CUDA runtime (which is part of the CUDA Toolkit) and is designed to be called from C and C++. The cuSPARSE library allows developers to access the computational resources of the NVIDIA graphics processing unit (GPU). It does not auto-parallelize across multiple GPUs [2].

Cusp is a library for sparse linear algebra and graph computations on CUDA. Cusp provides a flexible, high-level interface for manipulating sparse matrices and solving sparse linear systems [1].

VexCL is a vector expression template library for OpenCL/CUDA. It has been created for ease of GPGPU development with C++. VexCL strives to reduce the amount of boilerplate code needed to develop GPGPU applications. The library provides convenient and intuitive notation for vector arithmetic, reduction, sparse matrix-vector products, etc. Multi-device and even multi-platform computations are supported [3].

The Intel Math Kernel Library (Intel MKL) accelerates math processing routines that increase application performance and reduce development time. MKL includes highly vectorized and threaded Linear Algebra, Fast Fourier Transforms (FFT), Vector Math and Statistics functions.

2 Experimental Setup

The GPUs that were used to test the matrices were Tesla K40c, Tesla K20Xm and GTX770 for the CUSP, cuSPARSE, VexCL and ViennaCL libraries. For the MKL library, the performance was tested on Intel Xeon E5-2690 and Intel Core i7-5960X.

The matrices taken into consideration, listed in Table 1, are from the Florida Matrix Collection [9] and vary in size from 0.05 million to 24 million. The matrices are square. The density of the matrices are calculated as a percentage value of the NNZ to the total size of the matrix; i.e., $\frac{NNZ \times 100}{(Rows \times Columns)}$. The values are all double precision floats with 8-byte storage size in memory.

For the CUSP and ViennaCL libraries, the performance of the libraries in four formats of the matrices were considered: HYB, COO, CSR, ELL [7]. cuSPARSE was tested in HYB and CSR formats while VexCL and MKL were measured only in the CSR format.
Table 1: Matrices used from Florida Matrix Collection.

<table>
<thead>
<tr>
<th>Name</th>
<th>Dimension</th>
<th>NNZ</th>
<th>Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>rajat26.mtx</td>
<td>51032</td>
<td>249302</td>
<td>0.009572835</td>
</tr>
<tr>
<td>Lin.mtx</td>
<td>256000</td>
<td>1011200</td>
<td>0.001542969</td>
</tr>
<tr>
<td>rajat30.mtx</td>
<td>643994</td>
<td>6175377</td>
<td>0.001489018</td>
</tr>
<tr>
<td>webbase-1M.mtx</td>
<td>1000005</td>
<td>3105536</td>
<td>0.00031055</td>
</tr>
<tr>
<td>thermal2.mtx</td>
<td>1228045</td>
<td>4904179</td>
<td>0.00032519</td>
</tr>
<tr>
<td>kkt_power.mtx</td>
<td>2063494</td>
<td>8130343</td>
<td>0.000190942</td>
</tr>
<tr>
<td>wikipedia-20061104.mtx</td>
<td>3148440</td>
<td>39383235</td>
<td>0.000397302</td>
</tr>
<tr>
<td>venturiLevel3.mtx</td>
<td>4026819</td>
<td>8054237</td>
<td>4.96707E-05</td>
</tr>
<tr>
<td>rajat31.mtx</td>
<td>4690002</td>
<td>20316253</td>
<td>9.23629E-05</td>
</tr>
<tr>
<td>greatbritain_osm.mtx</td>
<td>7733822</td>
<td>8156517</td>
<td>1.36369E-05</td>
</tr>
<tr>
<td>delaunay_n23.mtx</td>
<td>8388608</td>
<td>25165784</td>
<td>3.57627E-05</td>
</tr>
<tr>
<td>germany_osm.mtx</td>
<td>11548845</td>
<td>12369181</td>
<td>9.27393E-06</td>
</tr>
<tr>
<td>hugebubbles-00020.mtx</td>
<td>21198119</td>
<td>31790179</td>
<td>7.07454E-06</td>
</tr>
<tr>
<td>road_usa.mtx</td>
<td>23947347</td>
<td>28854312</td>
<td>5.03148E-06</td>
</tr>
</tbody>
</table>

3 Results

3.1 CUSP

Figures 1-4 illustrate the performance of SpMV using the CUSP library with a CUDA backend on the different GPU architectures and different matrix formats. It is observed that the Tesla K20Xm is generally performing the best while Tesla K40c is the second good. GTX 770 appears to be the worst of the three. In the ELL format, a few matrices would not run as they seemed to run out of memory on the device. The HYB format proves to be the best among the different matrix types with an average flop rate of 5.524 GFlops on the Tesla K20Xm machine.

3.2 ViennaCL

Figures 5-8 illustrate the performance of SpMV using the ViennaCL library with an OpenCL backend on different GPU architectures and matrix formats. It is observed that the Tesla machines are performing similarly for the different formats of the matrices while GTX770 is particularly slow or out of memory. In the HYB format, Tesla K20Xm is better than the Tesla K40c by an order of magnitude. An interesting observation is the increase in flop rate for the webbase-1M matrix in the COO format, owing to the reason that the matrix may be specifically well suited for this format type. The HYB format proves to be the best among the different matrix types with an average flop rate of 4.255 GFlops on the Tesla K20Xm machine.
3.3 cuSPARSE

Figures 9 and 10 illustrate the performance of SpMV using the cuSPARSE library with a CUDA backend across different GPU architectures and matrix formats. In the HYB format, it is observed that the GPU architectures are performing similarly while in the CSR format, the Tesla machines are better than the GTX770. The HYB format proves to be the best among the different matrix types with an average flop rate of 7.585 GFlops on the Tesla K20Xm machine.

3.4 VexCL

Figure 11 illustrates the performance of SpMV using the VexCL with an OpenCL backend library on different GPU architectures and with a CSR matrix format. It is observed that the GTX770 machine is better than the Tesla machines with an average flop rate of 4.922 GFlops. It is also observed that Intel Xeon CPU flop rate is fairly consistent and is even better than the Tesla machines. Although we present the data in the CSR format, VexCL has its own internal representation of matrices using the VexCL::SpMat structure.

3.5 MKL

Figure 12 illustrates the performance of SpMV using the MKL library with an OpenMP backend on different CPU architectures and with a CSR matrix format. It is observed that the Intel Xeon architecture is better than the Intel-i7 architecture for all matrices in the CSR format. The average flop rate on the Intel Xeon architecture was found to be 12.286 GFlops.

4 Discussion

Li et al. [10] analyzed and showed the performance for almost a similar case of SpMV on GPUs. They report that the performance is generally better using the cuSPARSE library when compared to the CUSP library in the CSR format. In the same CSR format, ViennaCL library proves to be better than the CUSP library. The ELL format fails to run on a few matrices owing to size constraints. In the HYB format, the CUSP and ViennaCL library are almost as good as each other. We noted that their conclusions are in line with the results we have obtained.

Mazhar et al. [11] studied the performance of SpMV using the VexCL library on different GPU and CPU architectures and observed that the performance on the GTX770 machine is around 15GFlops. It is also seen that the VexCL SpMV performance is better on GTX770 than on most of the other architectures.

Bell et al. [7] showed that the flop rate for double precision SpMV is around 10 GFlops using the CSR and COO matrix format using CUDA backend.
Toledo et al. [13] improved the performance of SpMV in the CSR format to around 2.5 times by using block matrices and reducing load instructions. They also use prefetching mechanisms and reordering instructions to reduce cache misses as first proposed by Das et al. [8].

Pinar et al. [12] studied the performance of SpMV and showed that the performance can be improved by storing the matrices in Blocked Compressed Row Storage (BCRS) and Fixed-size blocking formats. Vuduc et al. [14] further improved the BCRS format by using an Unaligned Blocked Compressed Row Storage (UBCRS) format. A performance improvement ranging from 1.26x to 2.1x was observed.

Ashari et al. [6] proposed a new matrix format called Adaptice CSR (ACSR) that overcomes the limitations of the CSR format and provides better performance for SpMV. The challenge with the CSR format is that a fixed number of threads is assigned for each row which works for matrices with low standard deviation in NNZ per row. For matrices with high standard deviation in their NNZ per row, different number of threads should work in different rows. They solve this load imbalance issue using the ACSR format and obtain an average performance improvement of 1.18x and an upper limit of 1.75x over the HYB format for double precision SpMV.

Ashari et al. [5] also showed improvement in performance over the COO and ELL formats using their Blocked Row-Column (BRC) format. The ELL format suffers from the limitation that it works well only for matrices with similar numbers of NNZ per row. Otherwise, it pads with zeros and does redundant computation. COO performs a lot of atomic updates owing to a fine grained element-to-thread mapping which causes resource contention. They propose the BRC format in which they do not pad the entire matrix but split them into blocks and pad them based on the first row and column in the block which reduces memory usage and redundant computation. They obtain an average speed-up of 1.08x to 4.91x over the other conventional matrix formats.

References


Figure 1. Performance of SpMV with CUSP library, HYB matrix format

Figure 2. Performance of SpMV with CUSP library, COO matrix format
Figure 3. Performance of SpMV with CUSP library, CSR matrix format

Figure 4. Performance of SpMV with CUSP library, ELL matrix format
Figure 5. Performance of SpMV with ViennaCL library, HYB matrix format

Figure 6. Performance of SpMV with ViennaCL library, COO matrix format
Figure 7. Performance of SpMV with ViennaCL library, CSR matrix format

Figure 8. Performance of SpMV with ViennaCL library, ELL matrix format
Figure 9. Performance of SpMV with cuSPARSE library, HYB matrix format

Figure 10. Performance of SpMV with cuSPARSE library, CSR matrix format
Figure 11. Performance of SpMV with VexCL library, CSR matrix format

Figure 12. Performance of SpMV with MKL library, CSR matrix format