Optimization of GPU Kernels for Sparse Matrix Computations in Hypre

Chaoyu Zhang[1], Ruipeng Li[2](advisor), Hai Jiang[1](advisor)

[1]Arkansas State University, Department of Computer Science, Jonesboro, AR
[2]Lawrence Livermore National Laboratory, Center for Applied Scientific Computing

Introduction

The acceleration of sparse matrix computations on GPUs can significantly enhance the performance of iterative methods for solving linear systems. In this work, we consider the kernels of Sparse Matrix Vector Multiplications (SpMV), Sparse Triangular Matrix Solves (SpTrSv) and Sparse Matrix Matrix Multiplications (SpMM), which are often demanded by Algebraic Multigrid (AMG) solvers. With the CUDA and the hardware support of the Volta GPUs on Sierra, the existing kernels should be further optimized to fully take the advantage of the new hardware, and the optimizations have shown significant performance improvement. The presented kernels have been put in HYPRE for solving large scale linear systems on HPC equipped with GPUs. These shared-memory kernels for single GPU are the building blocks of distributed matrix operations required by the solver across multiple GPUs and compute nodes. The implementations of these kernels in Hypre and the code optimizations will be discussed.

Sparse Matrix-Vector Multiplication

Optimizations:

- Self-tuning number of threads per row

Based on rNNZ (average number of non-zeros per row) to choose a suitable number of threads.

Fixed size: K=16 threads per row →

```
If (rNNZ <  2)                 K=2;
If (rNNZ >  2 && rNNZ<4)       K=8;
If (rNNZ >  4 && rNNZ<8)       K=8;
If (rNNZ >  8 && rNNZ<16)      K=16;
If (rNNZ > 16 && rNNZ<32)      K=32;
```

- Warp reductions with shuffle instructions without shared memory

Using shuffle instructions to replace shared memory access in order to improve reduction operations. The instructions are directly working on threads' registers instead of the shared memory, which combine the synchronized write and read. Thus, shuffle instructions provide faster data exchanges in warps and memory access with lower latency and higher bandwidth.

```
//shared memory reduction
r[threadIdx.x] = sum;
   r[threadIdx.x] = sum = sum + r[threadIdx.x+8];
   r[threadIdx.x] = sum = sum + r[threadIdx.x+4];
   r[threadIdx.x] = sum = sum + r[threadIdx.x+2];
   r[threadIdx.x] = sum = sum + r[threadIdx.x+1];
   if (lane == 0)
     d_y[row] = r[threadIdx.x];
```

→

```
#pragma unroll
    for (HYPRE_Int d = K/2; d > 0; d >>= 1)
    { sum +=
    __shfl_down_sync(HYPRE_WARP_FULL_MASK, sum, d);}
    if (grid_group_id < n && group_lane == 0)
    { d_y[grid_group_id] = sum;}
```

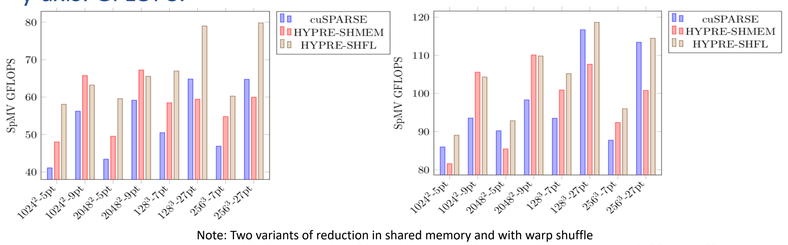
- First add during Load and Algorithm cascading

First Add During Load: each thread loads and adds two elements on the first level of reduction
As Figure shown above, Half threads are idle at the first loop iteration, which could be used for loading data.
With two loads and first add of the reduction:

```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
```

Algorithm cascading : combine sequential and parallel reduction.

- Performance of SpMV kernel On P100 (left) and V100 (right) with y-axis: GFLOPS.



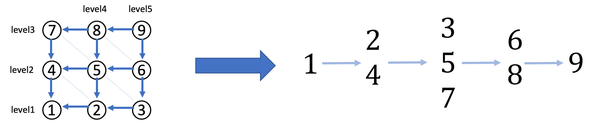
Note: Two variants of reduction in shared memory and with warp shuffle
We tested our kernels on the Nvidia Pascal and Volta GPU architectures available at LLNL, namely Nvidia P100 and V100.

Sparse Triangular Matrix Solve

Sparse triangular solve with a dense right-hand-side vector is traditionally a sequential algorithm in general. However, the level scheduling method reorganizes the solution vector in levels, where within each level the computation can be parallelized. Another approach named element scheduling analyzes the underlying DAG of the triangular matrix and thus can enjoy a finer level parallelism.

- Level Scheduling: parallelism within levels of dofs

The idea is to group the unknowns into different levels, where the first level consists of the nodes in the graph with zero in-degree and unknowns in any level should only depend on those in the previous levels. Therefore, the system can be solved level by level and the unknowns within the same level can be computed simultaneously.



Level scheduling for the 5-point stencil operator a 2-D regular grid

Total computation time T consists of kernel launch time, kernel execution time and synchronization time in (1):

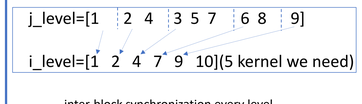
$$T = \sum_{i=1}^{N}(t_O^{(i)} + t_C^{(i)} + t_S^{(i)}) \quad (1) \qquad T = \sum_{i=1}^{M}(t_O^{(i)} + t_S^{(i)}) + \sum_{i=1}^{N}(t_C^{(i)}) \quad (M < N) \quad (2)$$

N is the number of kernel launch, $t_O^{(i)}$ is kernel launch time , $t_C^{(i)}$ is kernel execution time, $t_S^{(i)}$ is synchronization time. The idea is to reduce the number of kernel launches, so that T can be improved as shown in (2).

- Optimization: reducing the number of CUDA kernel launches

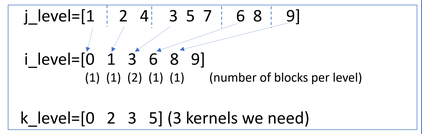
Remove unnecessary inter-block synchronizations and substitute with the intra-block __syncthreads().

Original design:



inter-block synchronization every level

Optimization:



reduced inter-block synchronization

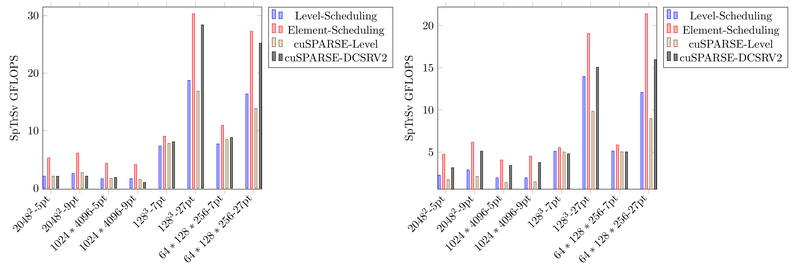
As figure shown above, suppose each of first three levels only need one block, we just kernel launch once and use __syncthreads() on GPU side. Thus, the original Sparse Triangular solve time was $T_1 = \sum_{i=1}^{5}\left(t_O^{(i)} + t_C^{(i)} + t_S^{(i)}\right)$, after optimization, $T_2 = \sum_{i=1}^{3}(t_O^{(i)} + t_S^{(i)}) + \sum_{i=1}^{5}(t_C^{(i)})$. in engineer and application, M usually smaller than N which means performance will be significant enhanced.

- Element Scheduling:

The algorithm of element scheduling is based on a counter scheme for the dependencies of each known and so enjoys a finer level of parallelism and a more aggressive scheduling, compared with the level scheduling approach. The element scheduling algorithm can avoid global synchronizations across CUDA blocks and there is a finer-level concurrency in this algorithm compared with the level scheduling. The down side of this algorithm is that it requires column access of the matrix. Since we assume the sparse matrix is available in the row CSR format, so matrix transposition is needed for these kernels. Therefore, the memory requirement is doubled.

- Performance of Gauss-Seidel relaxation on P100 (left) and V100 (right) with y-axis: GFLOPS.

Note: Analysis time of Hypre's Gauss-seidel relaxation is significant faster than cuSPARSE
The algorithm of DCSRV2 solver in cuSPARSE has not been published, so it is unknown.



Sparse Matrix-Matrix Multiplication

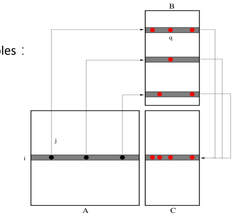
We consider sparse matrix–matrix multiplication C = AB in the CSR format. The goal is to exploit all the levels of parallelism available in the computation with a low memory requirement. Our GPU implementation is based on hash table as the *sparse accumulator*, which allows the most concurrency and the efficiency in memory use. This kernel is a single warp works on a row of the product matrix. Currently, a simple static scheduling approach is used, whereas a more complicated dynamic approach using a shared job queue to schedule warps is under the investigation.

- Optimization Method and future work:

Accumulation rows with hash tables:

- Each row i has a hash table T_i, which should be *large enough* to hold $C_{i,:}$
- Each red dot corresponds to an insertion into T_i with **(Key, val) = (q, $a_{i,j}b_{j,q}$)**
- All the insertions can be executed concurrently
- Need to serialize accesses to the same position of T_i

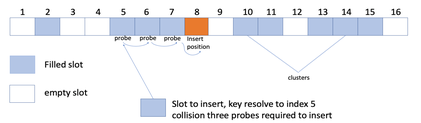
Accumulating sparse rows with hash tables :



Hash-Search-and-Insert: if q is found, update the value by adding $a_{i,j}b_{j,q}$; otherwise insert the pair.

- Hash Functions and Probings

For hash table with s slots, Hash table
$h'(key) = key \bmod s$ Linear
probing: $h(key, i) = (h'(key) + i) \mod s$



- A complete hash-table based SpMM algorithm workflow

1) Row NNZ estimation: allocate "reasonable-sized" hash tables for 2);
2) Symbolic analysis: compute row counts (bounds) and row pointers, and allocate "adequate-sized" hash tables for 3);
3) Numeric multiplication: compute the column indices and values;
4) Post-processing: remove the gaps between rows computed from 3);

Remarks:

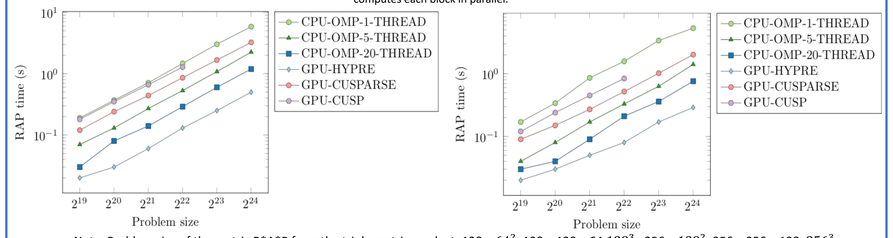
- If have prior knowledge about row NNZ, can omit 1)
- If memory is not an issue or problem is very "structured", maybe omit 1) and 2) and directly perform 3) with the naive upper-bound
- On return of 2), can know if row counts are exact. If so, omit 4)
- If just want *exact* row NNZ, run 2) twice

- SpMM Kernel Memory:

1) The memory is indeed very limited on the current GPUs. The out-of-memory issue has not been addressed in this work, We always assume the local problems can fit on a GPU.
2) This is a reasonable assumption in the context of parallel AMG solvers, which is able to solve large-scale systems that can have billions of degrees of freedom.
3) The global system is distributed across a large number of compute nodes with multiple GPUs on each node. As we showed, the size of the local problems can be up to several million, which is usually quite large for the settings of real numerical simulations.

- Performance of CSR SpMM 3-D 7-pt Laplacians on P100 (left) and V100(right) with y-axis: time(log-scale).

Note: The CPU used was IBM Power9, which has 40 cores. The CPU code was parallelized with OpenMP. The CPU SpGEMM for C=AB partition the rows of C into equally sized row blocks and each thread computes each block in parallel.



Note: Problem size of the matrix R*A*P from the triple matrix product: 128×64^2, $128 \times 128 \times 64$,128^3 , 256×128^2, $256 \times 256 \times 128$, 256^3
Performance: 2.4x (ray) and 2.6x (lassen) faster than CPU (OpenMP 20-THREAD)

Conclusion

- SpMV, SpTrSv and SpMM are important computation kernels for AMG solve and setup phase.
- Well designed, implemented and optimized GPU kernels can outperform their CPU counterparts by 1 or 2 orders of magnitude.
- Existing SpMV algorithm was optimized to take advantage of cutting-edge features of the latest GPUs. Two parallel SpTrSv methods based on different scheduling schemes have been developed and implemented. Hash-table based parallel SpMM approach was developed.
- Better performance was obtained compared with the state-of-the-art cuSPARSE library.

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

IM Review and Release number: LLNL-POST-782245