OAK RIDGE National Laboratory

OpenMP Offloading For Density Matrix Renormalization Group Hamiltonian Application Kernel

Wael Elwasif, Ed D'azevedo,
Arghya Chatterjee, Gonzalo Alvarez,
Oscar Hernandez

ORNL

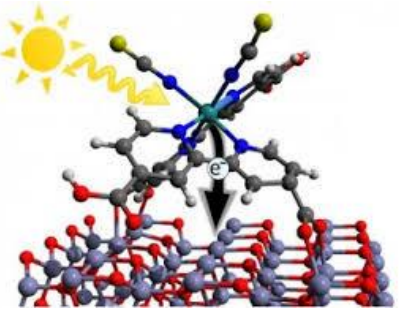
This work has been supported by the US Department of Energy's Office of Science's Scientific Discovery through Advanced Computing (SciDAC) project and Oak Ridge National Laboratory. ORNL is managed by UT-Battelle, LLC for the U. S. Department of Energy under Contract No. DE-AC05-00OR22725.

U.S. DEPARTMENT OF ENERGY

Overview

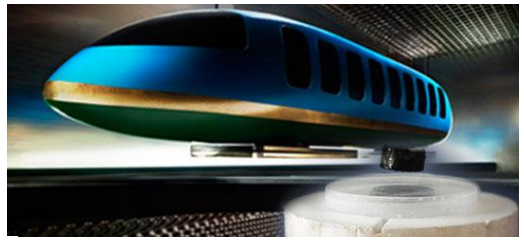
- Application background
- Key computational Kernel : Hamiltonian Application
- GPU Offloading using Batched GEMM
- Scaling out:
  - Approach 1: Hybrid OpenMP CPU- (many)GPU tasks
  - Approach 2: Single GPU compute task
- Conclusions and lessons learned

Domain: Nanoscale Modelling

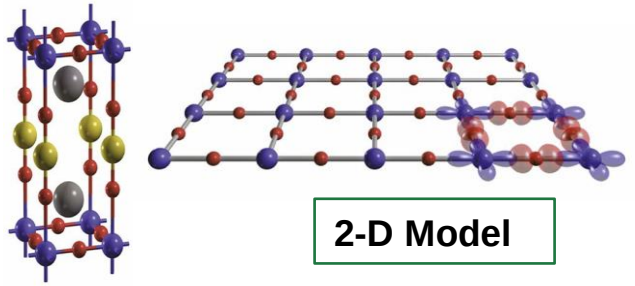


- Density Matrix Renormalization Group (**DMRG**)
  - Code : dmrg++
  - https://github.com/g1257/dmrgpp
  - Originally limited to essentially 1-D problems

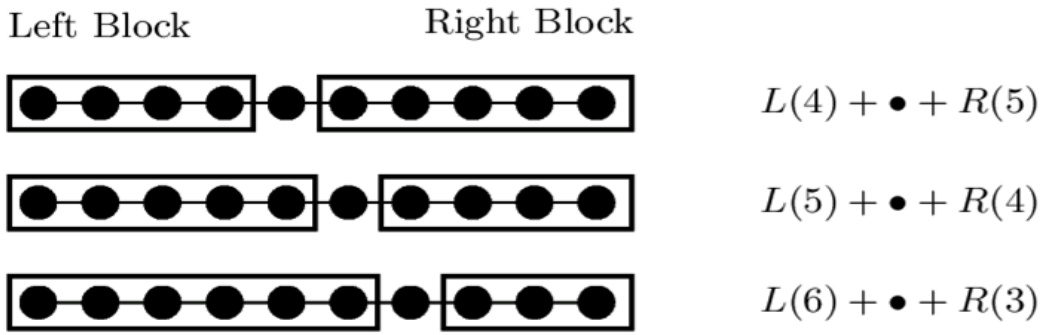
- Goal : Extend to 2-D larger problems
  - Exponential growth in runtime as number of quantum states increases



- Key computational kernel:
  - Hamiltonian application

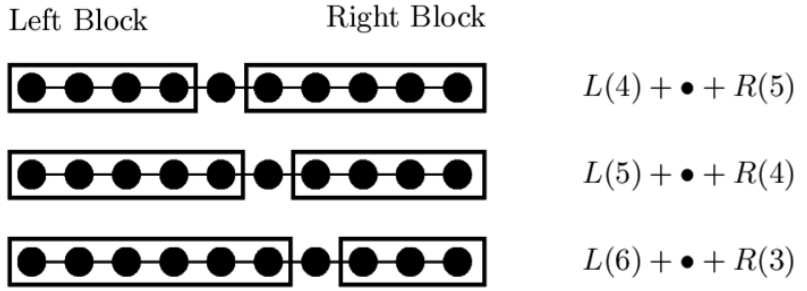


Background



- Large Eigenvalue problem
- Hamiltonian matrix on "sites"
- 4 possible states for each site (Hubbard Model)
  - 0 ⇒ No particle 1 ⇒ 1 "***up***" particle
  - 2 ⇒ 1 "***down***" particle 3 ⇒ 1 "***up***" particle and 1 "***down***" particle
  - *Different quantum models have different number of states*
- Block ⇒ Finite set of sites
- L(4) ⇒ 4 sites in Left block ⇒ 4^{10} potential configurations
- Algorithm proceeds in sweeps
  - Maximum computational load in the middle of the system
  - This is where we focus our efforts

Hamiltonian matrix



- https://g1257.github.io/dmrgPlusPlus/DmrgComputational.pdf

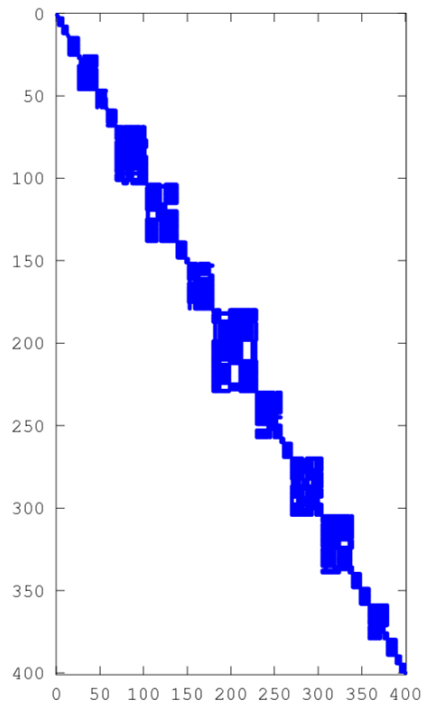
$$H' = H_L \otimes I_R + I_L \otimes H_R + \sum_{\gamma=0}^{\gamma<\Gamma} c_L^\gamma \otimes c_R^\gamma$$

size(H_L) is the number of basis on Left block

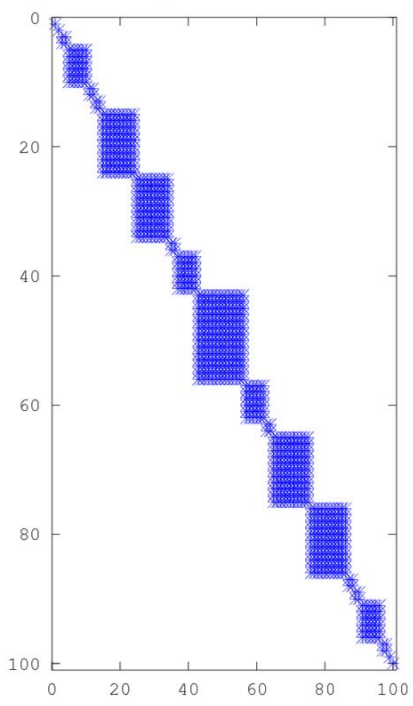
- For H_L (400x400) and H_R (100x100) ⇒ ***H' (40000x40000)***
- H_L, H_R can be reordered to be block diagonal
  ⇒ so H' is also block diagonal

Left and Right Hamiltonian

- **Key Idea:**
  Reorder states based on quantum number (n_{up}, n_{down}) to expose block pattern
- Hamiltonian represented as sum of Kronecker products of smaller dense matrices



Left Hamiltonian (H_L)



Right Hamiltonian (H_R)

Algorithm Overview

- H' can be permuted to have only block submatrices on the main diagonal
  - conserve "symmetry" quantum number ($target_{up}$, $target_{down}$)
- Compute lowest eigenvalue and eigen-vector on one of the diagonal block that correspond to a given target "quantum number" ($target_{up}$, $target_{down}$) using **iterative** Lanczos algorithm
- Form "density matrix" (X*X') using converged eigenvector and compute eigen-decomposition of density matrix (or SVD)
- Truncate quantum states from 4*M to M states
  - Based on projection to dominant eigen-vectors of density matrix → discard small eigen-values

Kronecker Product

- vec(Y) = kron(A,B) * vec(X) computed as **Y = B * X * A^T**
  - **X** reshaped into conformable matrix for multiplication
  - **Y** reshaped into a vector after multiplication
- Efficient ***O(N^3)*** algorithm
  - Expanding the product will take ***O(N^4)*** operations.
  - Savings in both memory and FLOPS

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & \cdots & a_{1n}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & \cdots & a_{mn}\mathbf{B} \end{bmatrix}$$

Hamiltonian Application

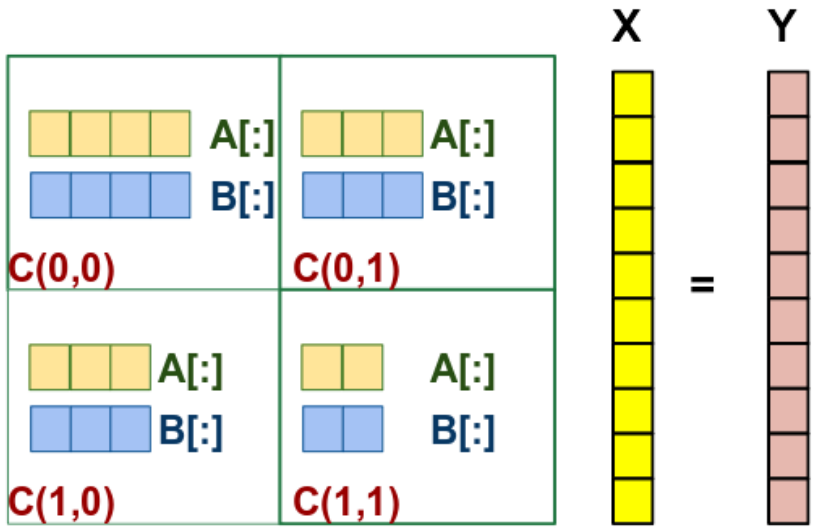
$$C[I,J] = \sum_k A_{IJ}^{(k)} \otimes B_{IJ}^{(k)}$$

$$Y[I] = \sum_J C[I,J] * X[J]$$

$$C[I,J] * X[J] = \left(\sum_k^K A_{IJ}^{(k)} \otimes B_{IJ}^{(k)} \right) * X[J]$$

$$= \sum_k^K (B_{IJ}^{(k)} * X[J] * (A_{IJ}^{(k)})^t)$$

$$= \sum_k^K (W_{IJ}^{(k)} * (A_{IJ}^{(k)})^t)$$



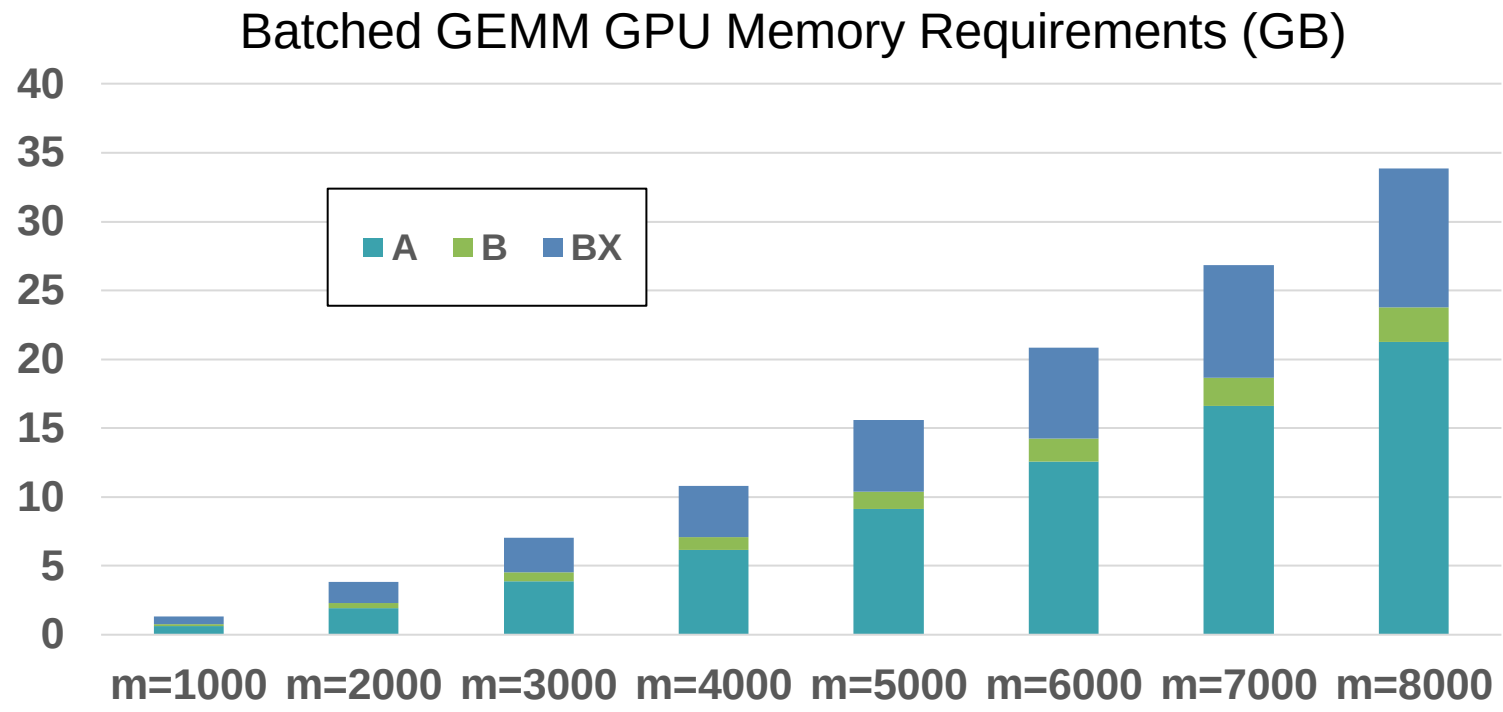
- NxN block partitioned sparse matrix
- Each sub-matrix (or cell) of C{i,j} is sum of Kronecker products
- Load balance : Number of entries and size of Ai and Bi in each cell

Scaling Batched GEMM implementation

128 Sites, 2 Operators per site

Approach

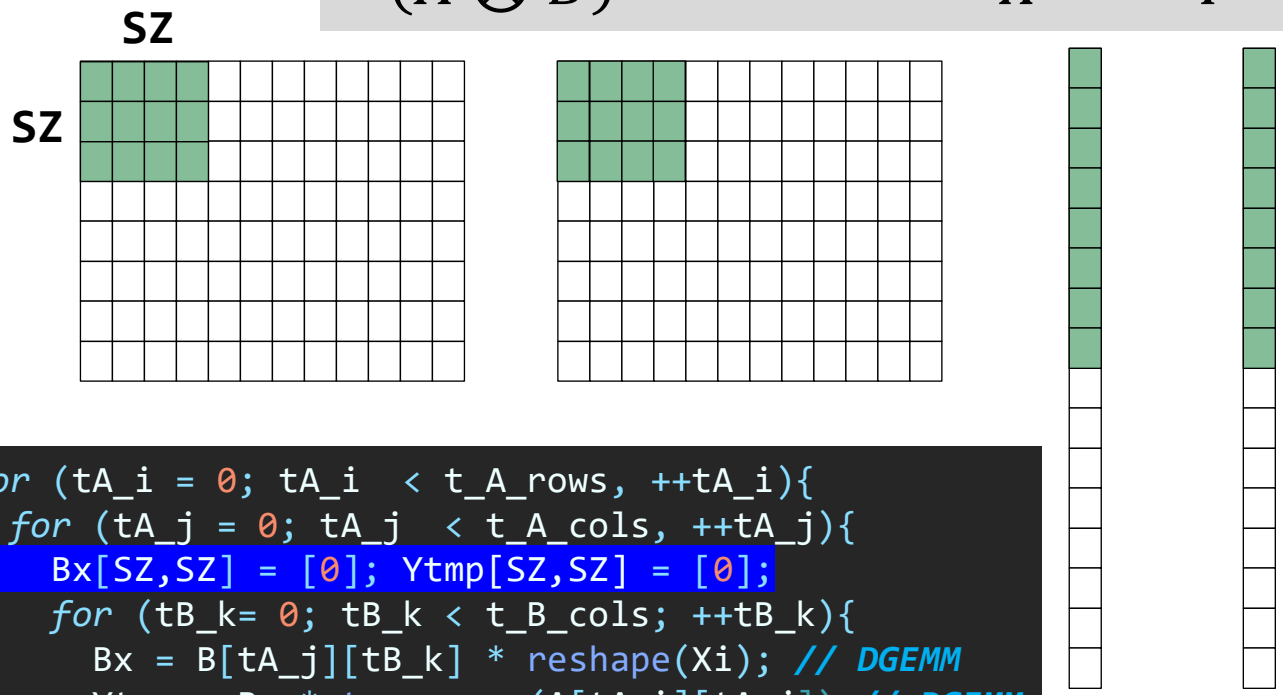
- Split work between CPU and GPU
- Use tiling to reduce size of needed scratch space



M = Total number of quantum states

Tiling for Kronecker Multiplication

$$(A \otimes B) \quad * \quad X \quad = \quad Y$$




```
1.  for (tA_i = 0; tA_i  < t_A_rows, ++tA_i){
2.    for (tA_j = 0; tA_j  < t_A_cols, ++tA_j){
3.      Bx[SZ,SZ] = [0]; Ytmp[SZ,SZ] = [0];
4.      for (tB_k= 0; tB_k < t_B_cols; ++tB_k){
5.        Bx = B[tA_j][tB_k] * reshape(Xi); // DGEMM
6.        Ytmp = Bx * transpose(A[tA_i][tA_j]);// DGEMM
7.        Y += reshape(Ytmp);
8.      }
9.    }
10. }
```

Pseudo Code for Tiled Kronecker multiplication

- Full scratch size: $\boldsymbol{f(size(B),\ size(X))}$
- Tiled scratch size: $\boldsymbol{2 * SZ * SZ}$
  - Per execution thread
- $\boldsymbol{SZ = 1}$: Full Kronecker Product, compute $\boldsymbol{O(N^4)}$
- $\boldsymbol{SZ = size(B)}$: Full scratch space, compute $\boldsymbol{O(N^3)}$
- In-place `reshape()` and `transpose()` operations
- Small-size optimized DGEMM for maximum performance

Approach 1 : One task per Kronecker multiplication kernel

- Sort kernels to offload "heavy weight" kernels first
- Fit as many kernels on GPU as possible
  - Copy A's and B's only on first iteration ⇔ remain constant
  - X and Y copied each iteration
- Run the rest in separate tasks on OpenMP CPU threads
- OMP Features:
  - Task loops
  - Concurrent offloading from multiple CPU threads (clang, XL only)
  - **`target nowait`** to not block calling thread (XL only)

Tasks for Hybrid CPU-GPU Execution

- Separate **`taskgroup`** for the GPU with barrier to copy consistent **`Ygpu`**
- **`grainsize > 1`** for CPU tasks since CPU tasks will be finer granularity
- **`map_copy_AB()`** gathers A's and B's into contiguous CPU memory for single CPU-GPU data transfer – ***need to reconstruct GPU Array data structure***
- Sorting computes number of OMP teams for each GPU – based on tile size

Pseudo Code for Task Implementation

```
#pragma omp taskgroup
  {
#pragma omp parallel
    {
#pragma omp single nowait
      {
#pragma omp task untied  // CPU master task
        {
#pragma omp taskloop untied grainsize(N) nogroup
          for (int i = 0; i < CPU_kernels.size(); ++i)
              kronCPU(A[i], B[i], X, Ycpu, xstart[i], ystart[i]);
        }
#pragma omp task  // GPU master task
        {
          if (iter == 0)   // Copy A, B to GPU only on first iteration
              map_copy_AB ();
#pragma omp taskgroup
          {
#pragma omp taskloop grainsize(1) nogroup
           for (int i = 0; i < GPU_kernels.size(); ++i)
            KronGPU(A[i], B[i], X, Ygpu, xstart[i], ystart[i], nteams[i]);
          } // GPU task group – barrier
#pragma omp target exit data map(from : Ygpu[:Ygpu.size()])
        } // Master GPU task
      } // omp single
    } // omp parallel
  } // omp task group
Ycpu += Ygpu; // Accumulate GPU into CPU
```

The GPU kernel : issues for portability and performance

- With optional deferred execution of **`nowait`**, calling CPU thread blocks impacting execution of CPU tasks
- Limits on how big, and where will **`BX`** and **`Ytmp`** be allocated?
  clang : Global memory ?
  XL : Shared Memory ?
- Team-level BLAS libraries ??

```
#pragma omp target nowait
{
#pragma omp teams distribute collapse(4) num_teams(nteams)
   {   // Tiling loop nest
     for (int iastart = 0; iastart < A.nrows(); iastart += SZ) {
       for (int ibstart = 0; ibstart < B.nrows(); ibstart += SZ) {
         for (int jastart = 0; jastart < A.ncols(); jastart += SZ) {
           for (int jbstart = 0; jbstart < B.ncols(); jbstart += SZ) {
           ...
             T BX[SZ * SZ] = [0], Ytmp[SZ * SZ] = [0]; // Compiler choice?
           ...
#pragma omp parallel for collapse(2)
// User defined tile DGEMM BX = B[TI_B,TJ_B] * X[]
           ...
#pragma omp parallel for collapse(2)
// User defined tile DGEMM Ytmp = Bx[] * transpose(A[TI_A, TJ_A])
           ...
           Y += Ytmp;  // Atomic update
           }
    ...
}
```

Pseudo Code for GPU kernel

Case Study

- H/W
  - Intel Xeon E5-4640v4, 12x4 cores running at 2.1GHz with 512GB of DDR4-2400 ECC memory.
    GPU : **12 GB** PCIe 3.0 V100 Titan V GPGPU.
- Synthetic mini-app
  - Compiler : **clang -12 (trunk)**
  - Compare with batched GEMM implementation based on UVM & MAGMA library (**BG_GPU)** and CPU-only OMP full batched GEMM **(BG_CPU)**
  - Test no offloading (**CPU**), offload using **5GB**, **8GB**, and **10GB** of GPU memory.
- Simulated System using 128 sites, 2 operators per site
  - **M6000** : 6000 Saved Quantum States
  - **M8000** : 8000 Saved Quantum States

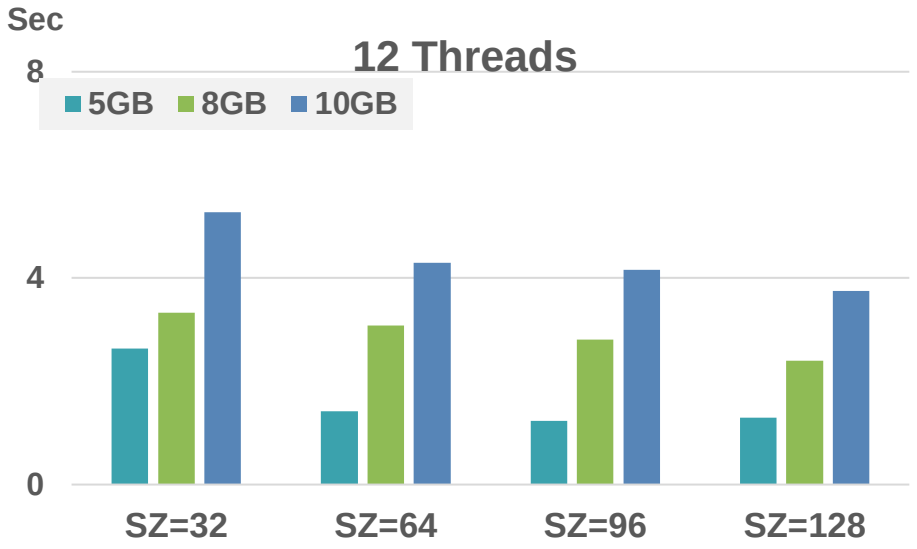
System	Total Kron Ops	Memory (A , B)	Memory range
M6000	462,722	14.24 GB	5 – 59,650 Bytes
M8000	518,162	23.78 GB	5 – 106,097 Bytes

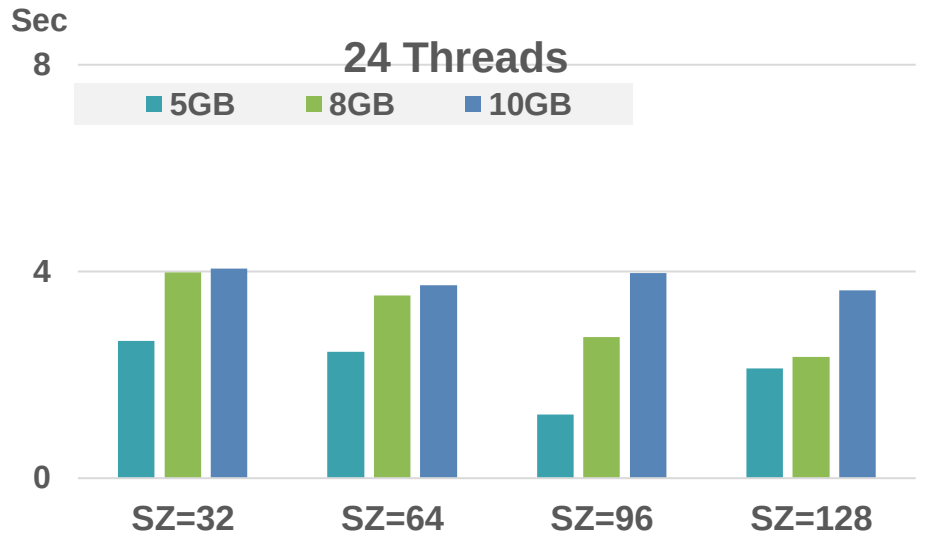
- Measured metric
  - Iterative solve runs for > 20 iterations
  - We measure average runtime for iterations 1,2 : capture steady state performance

Larger overhead for smaller problem

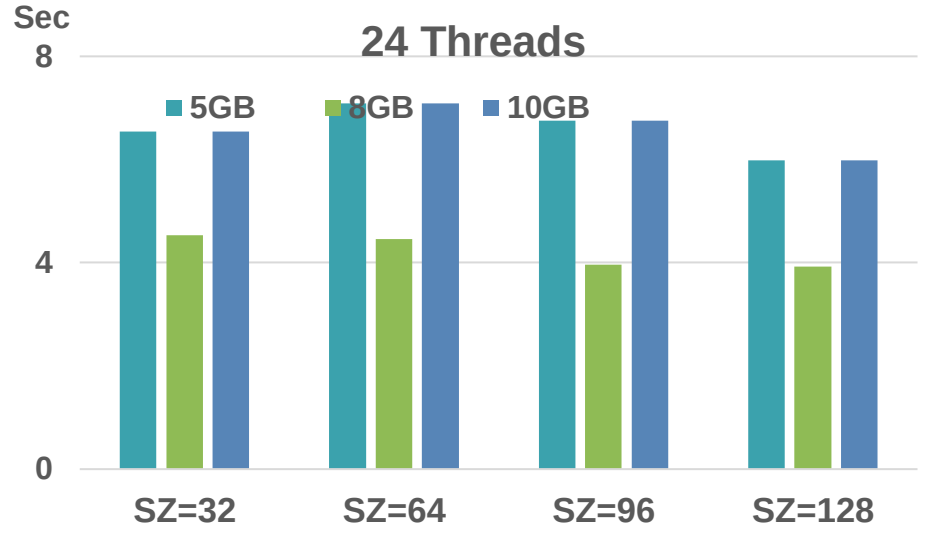
$$T_0 - (T_1 + T_2)/2$$

M8000



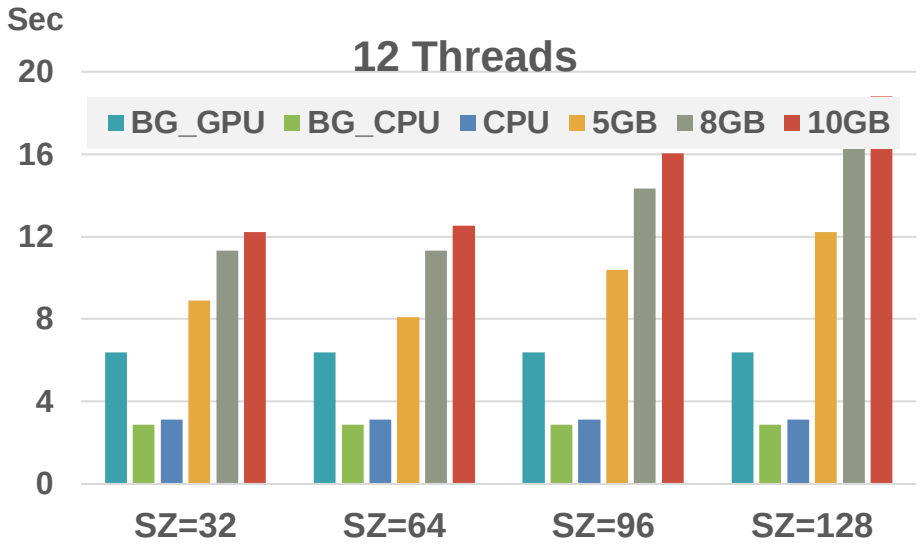


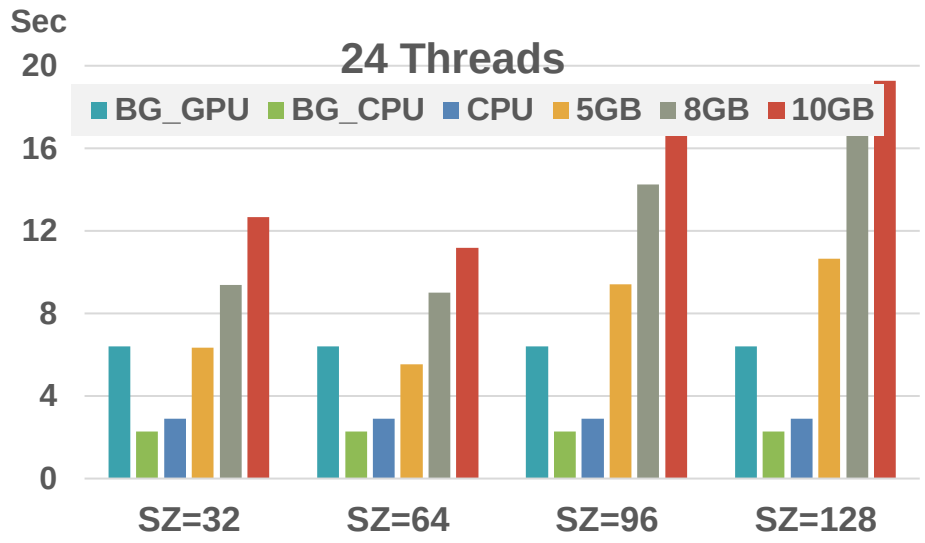
M6000



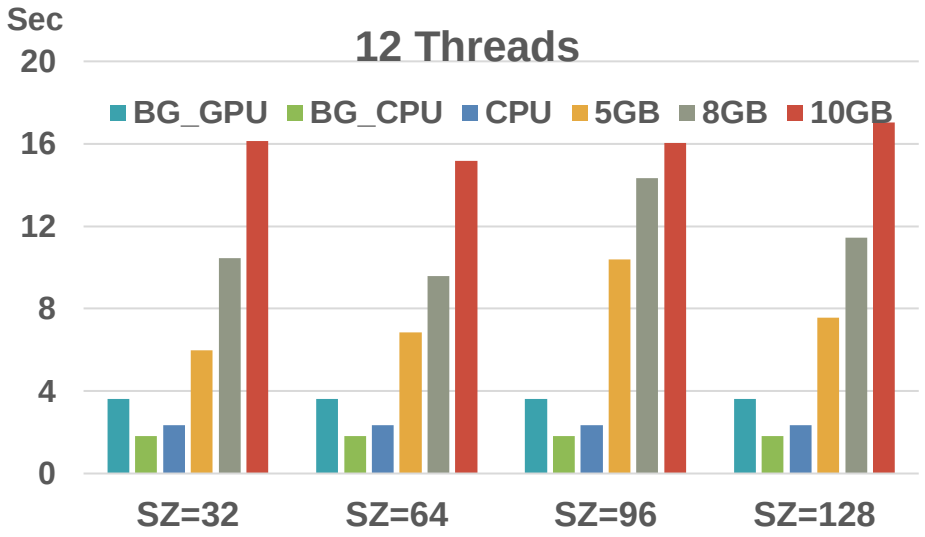
Task-Based approach fails to improve iteration run time

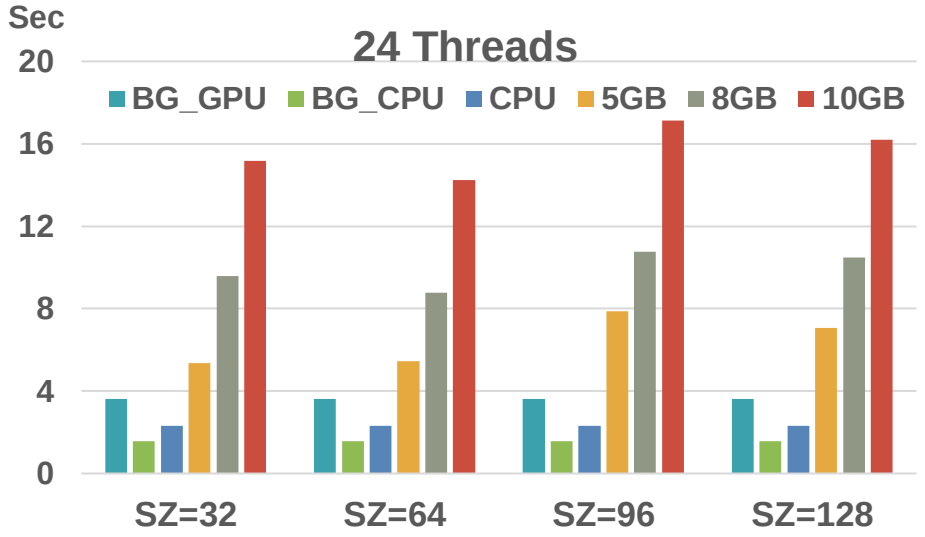
M8000



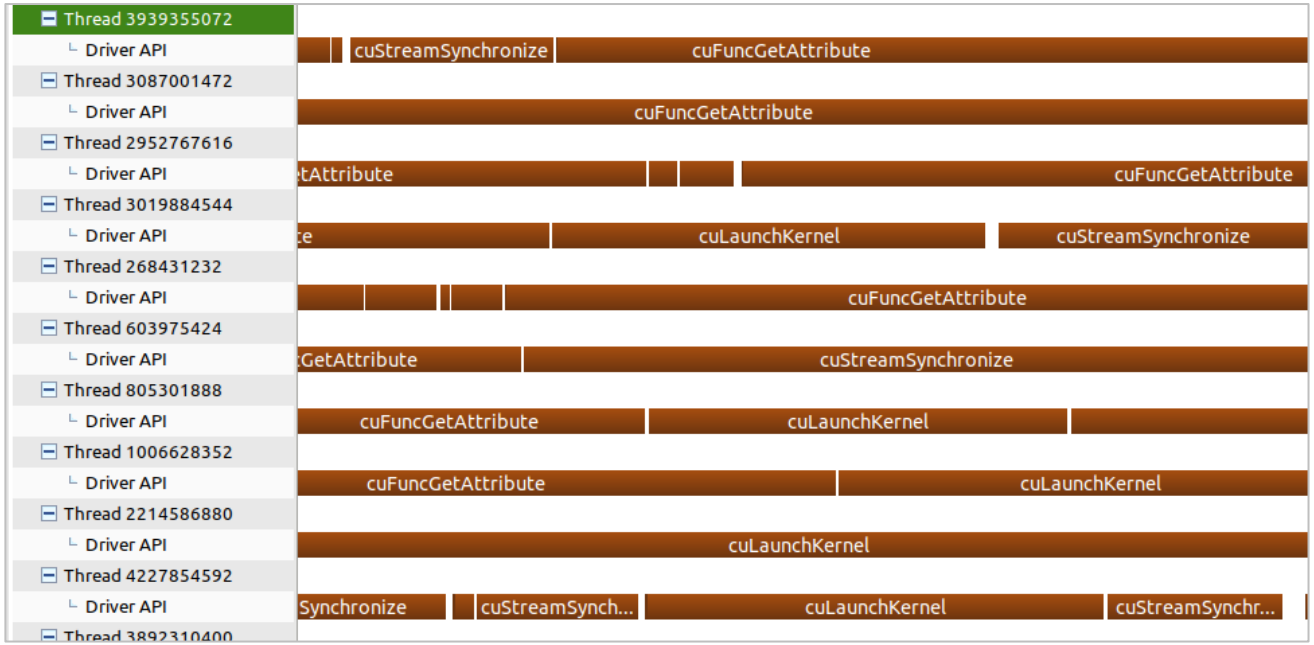


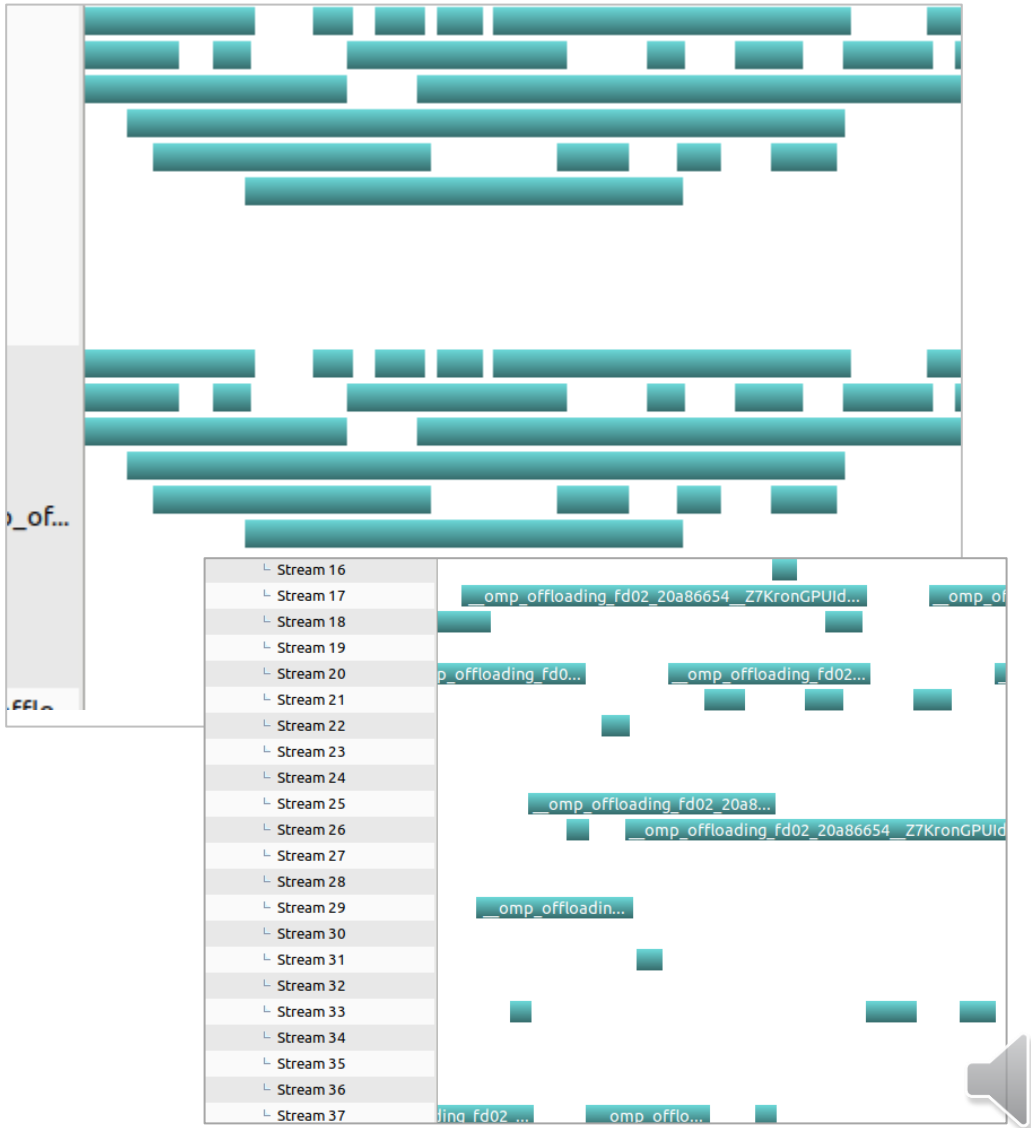
M6000





`nowait` needed for tasking to work





- Managing offloaded tasks interferes with the ability of the CPU threads to do useful computational work
- Simply using more threads does not fix this issue
- “smart” thread management on CPU ??
  - Separate CPU “workers” from CPU “offloaders”

Approach 2: Avoid GPU tasking

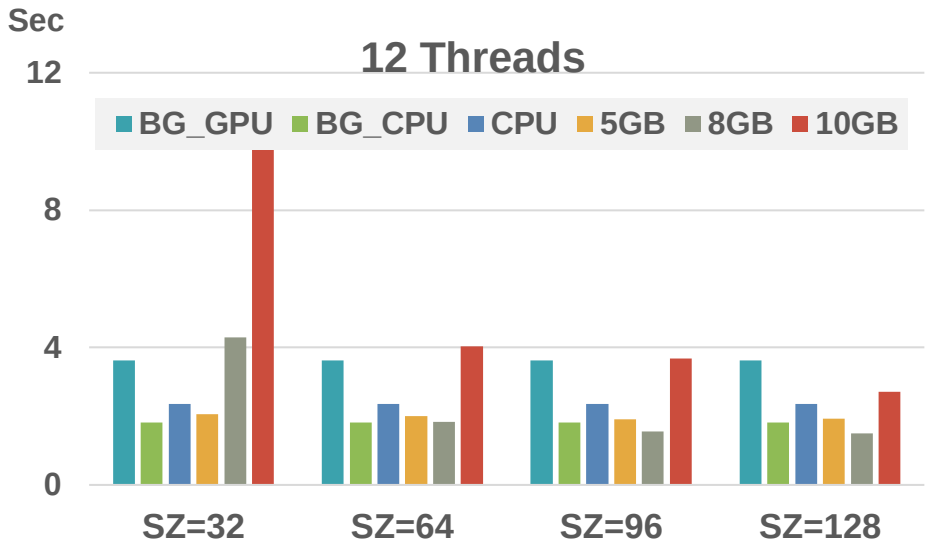
```
1.  #pragma omp teams distribute num_teams(NTEAMS) collapse(3)
2.  {
3.   for (int ik = 0; ik < n_kronGPU; ik++)
4.    for (int iblock = 0; iblock < max_nblocks_ia; iblock++)
5.      for (int jblock = 0; jblock < max_nblocks_ib; jblock++) {
6.          ...
7.          if (out_of_bounds(ik, iblock, jblock)
8.             continue;
9.          ...
10.         scratch_offset = omp_get_team_num() * SZ * SZ;
11.         BX = &BX_ALL[scratch_offset];     // Per team BX[]
12.         Ytmp = &Ytmp_ALL[scratch_offset]; // Per team Ytmp[]
13.         ...
14.         A = ALL_A[ik]; B = ALL_B[ik];
15.         for (ja = 0; ja < A[ik].ncols(); ja += SZ) {
16.           for (jb = 0; jb < B[ik].ncols(); jbs += SZ) {
17.             // Tiling logic ...
18.             BX[] = [0];
19. #pragma omp parallel for collapse(2)
20.             // BX = B[:,:] * reshape(Xi)
21.             Ytmp[] = [0];
22. #pragma omp parallel for collapse(2)
23.             // Ytmp = BX * transpose(A[:,:])
24.             Y[] += Ytmp[];
25.           }
26.         }
27.     }
28. }
```

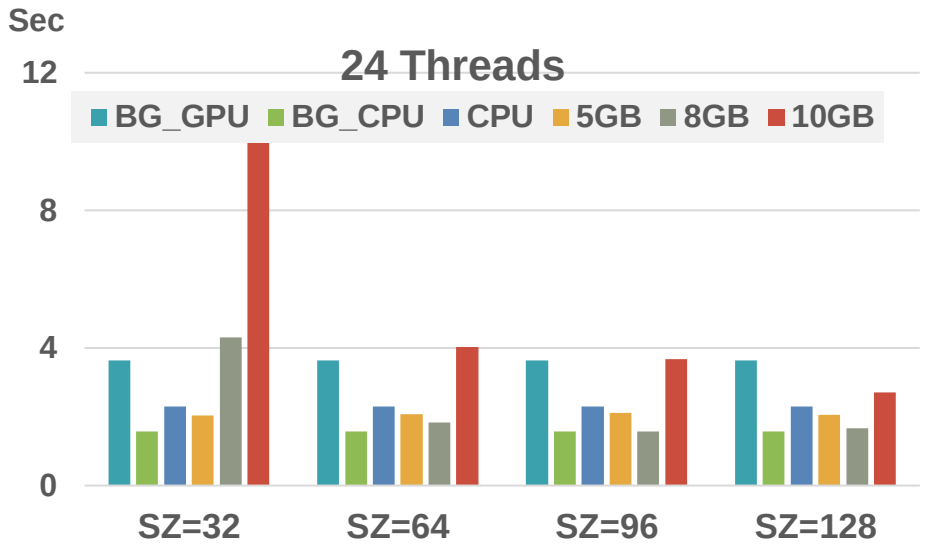
Pseudo Code for Batched Kronecker multiplication

- ***CPU tasks unchanged***, use single compute target region for all GPU multiplications
- Fixed global scratch size
  $$\mathbf{NTEAMS} * \mathbf{2} * \boldsymbol{SZ} * \boldsymbol{SZ}$$
- Fixed outer loop over maximum number of tiles for any kernel
  - Continue when out of bound
- Pre-allocated single global scratch space for **BX** and **Ytmp** (lines 10-12).
- Number of teams and tile size affect storage for A, B.
  - Insignificant practical impact

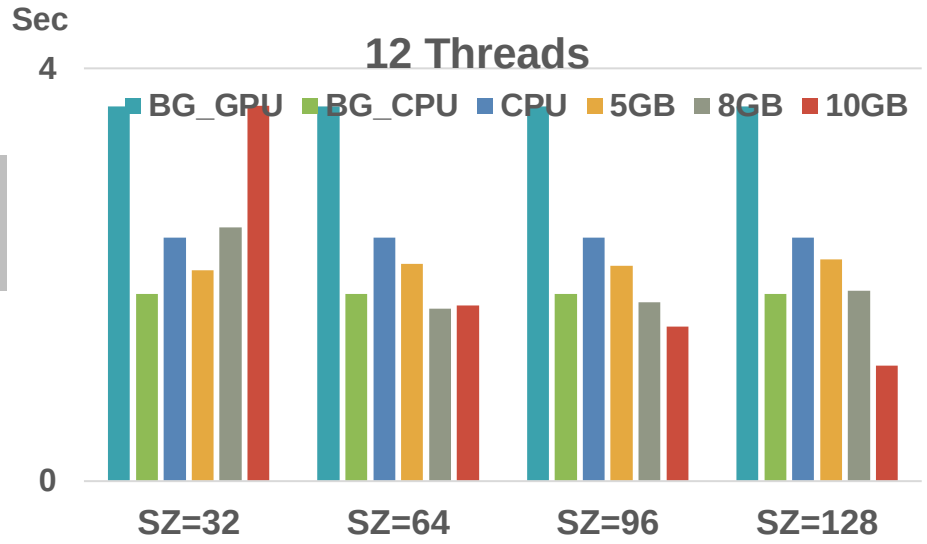
Complex parameter interaction for performance

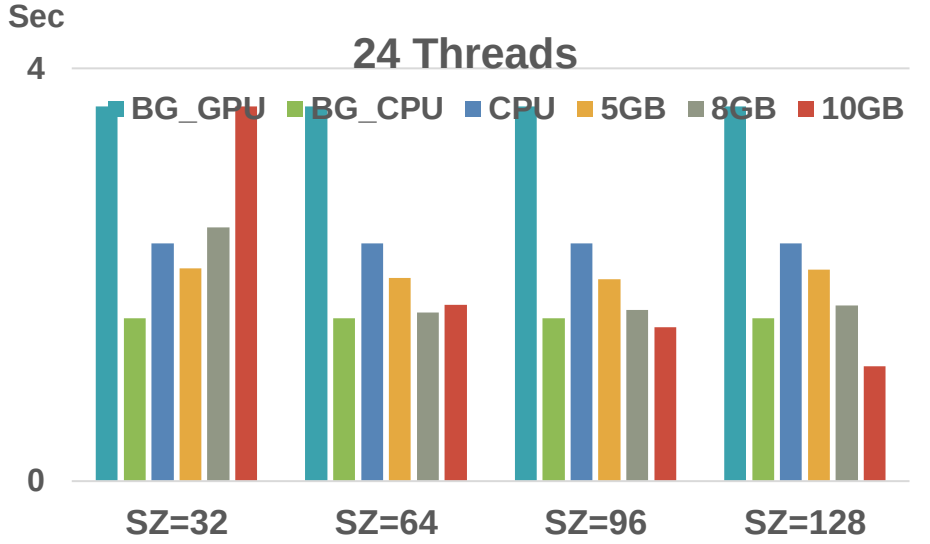
M6000 1000 Teams





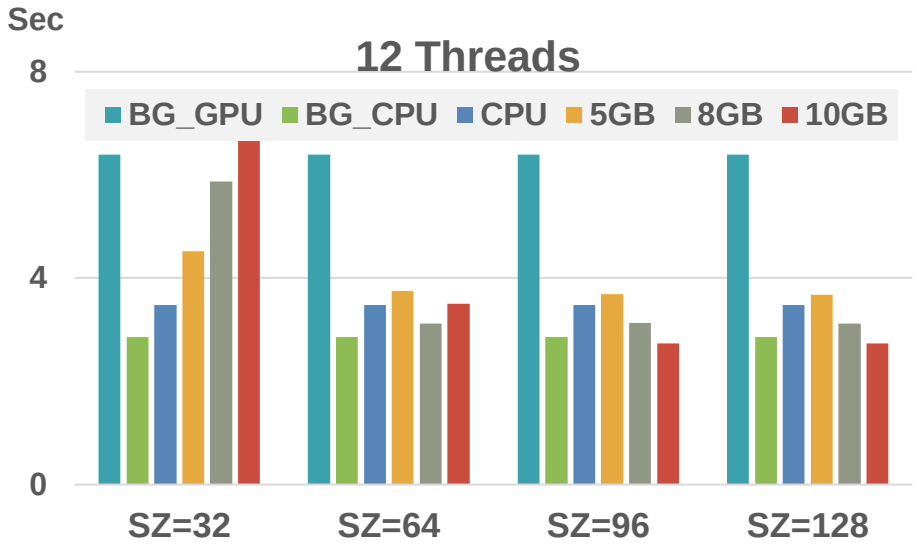
M6000 5000 Teams

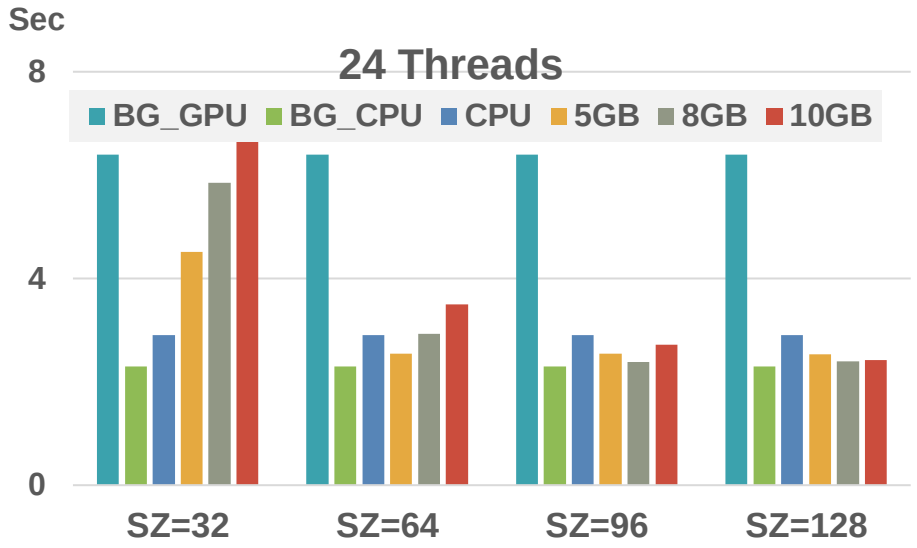




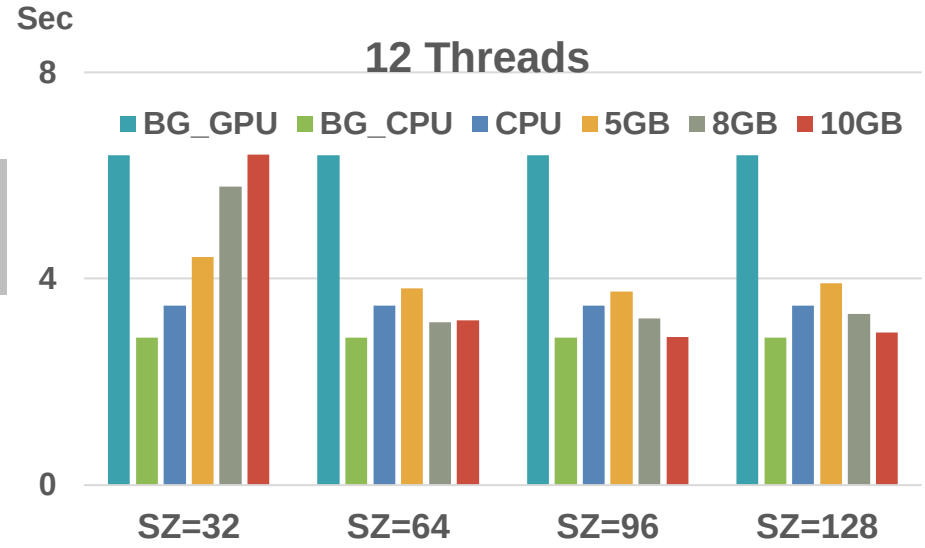
Changes with larger problems : CPU impact

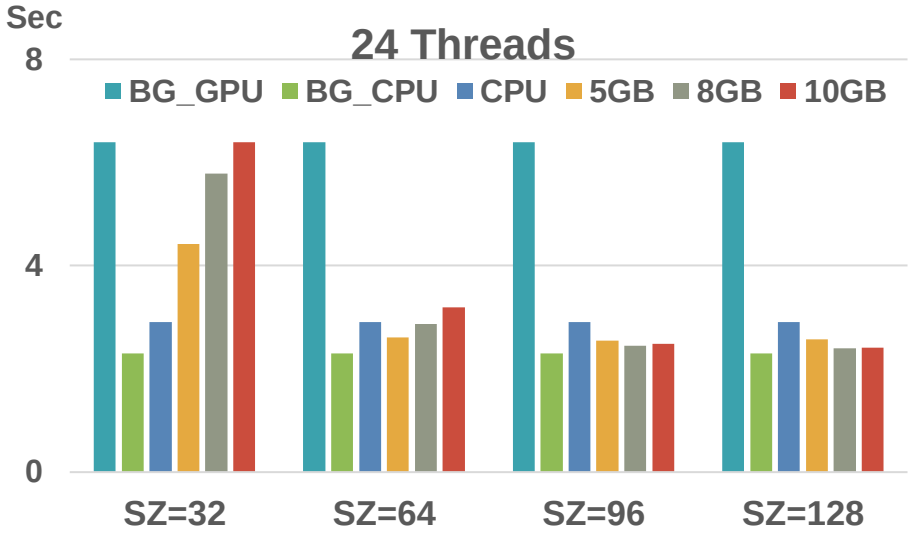
M8000 1000 Teams





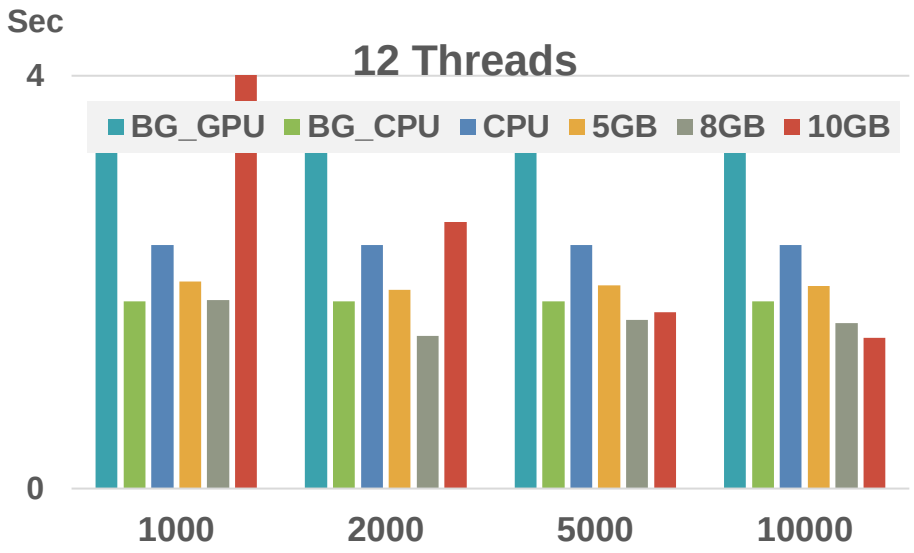
M8000 5000 Teams

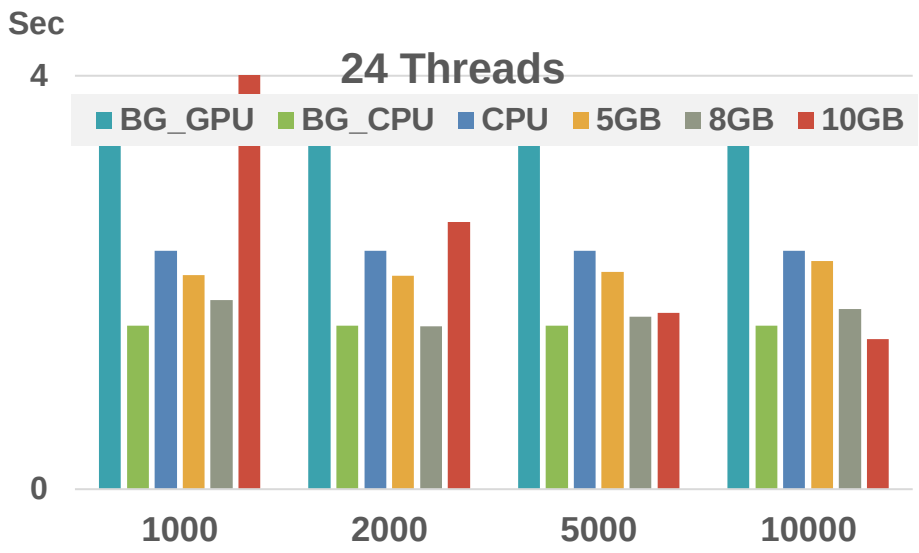




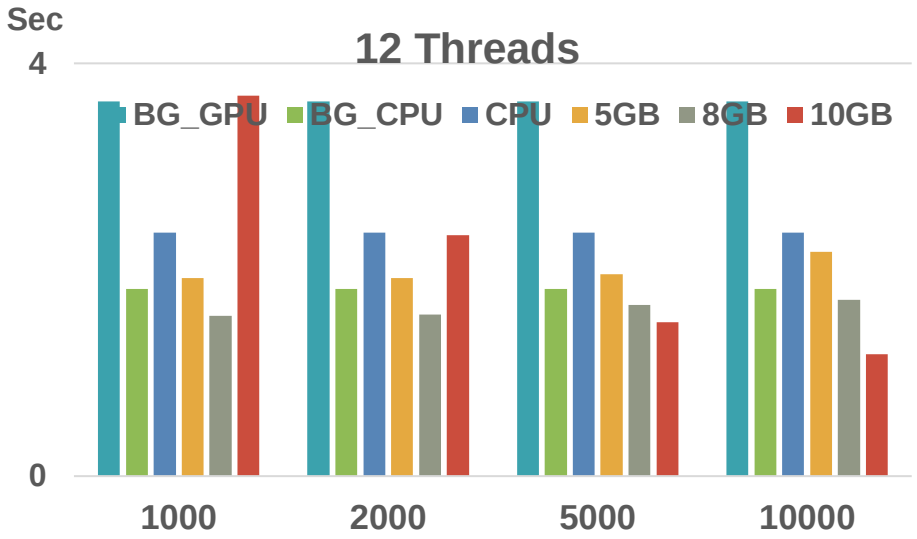
More teams for better performance ?

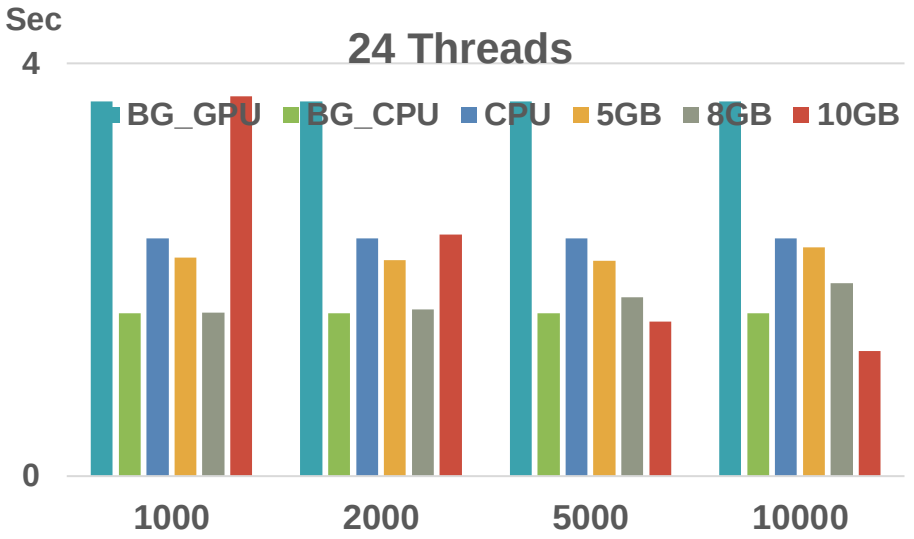
M6000
SZ = 64





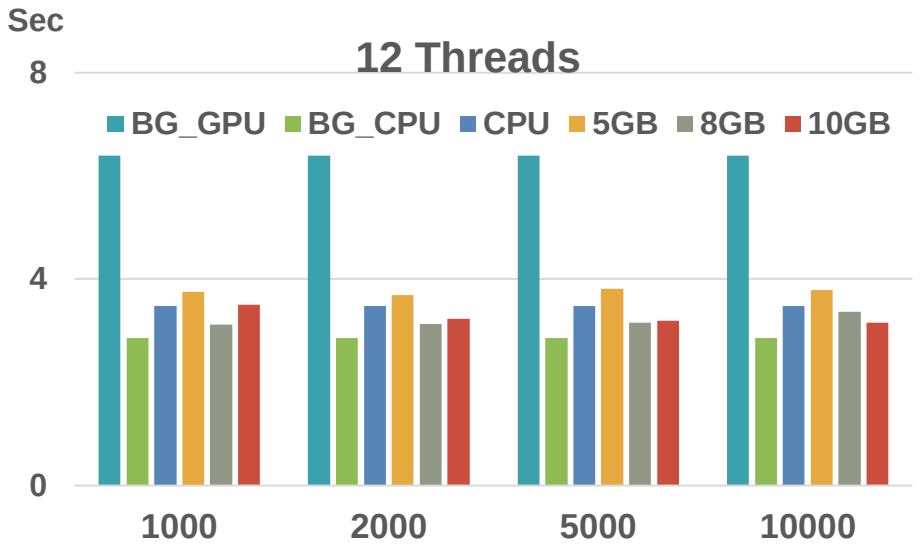
M6000
SZ = 96

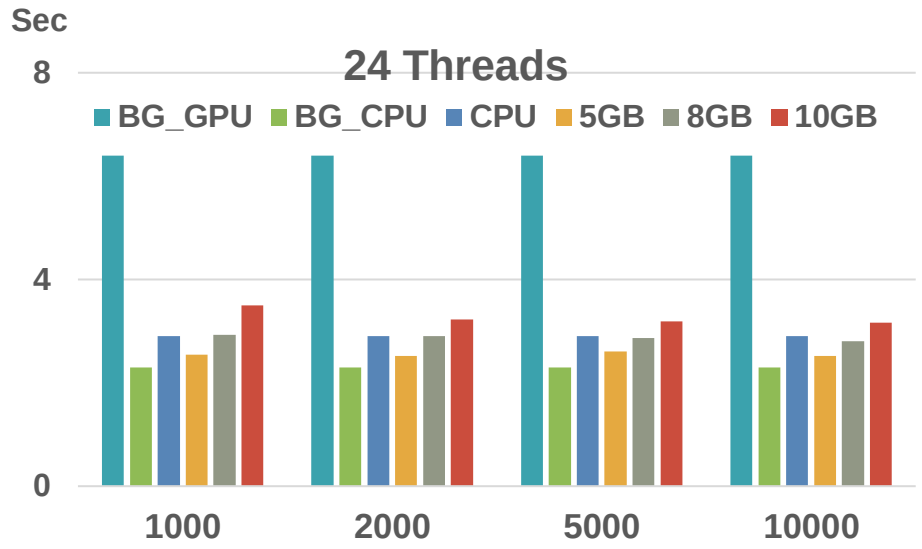




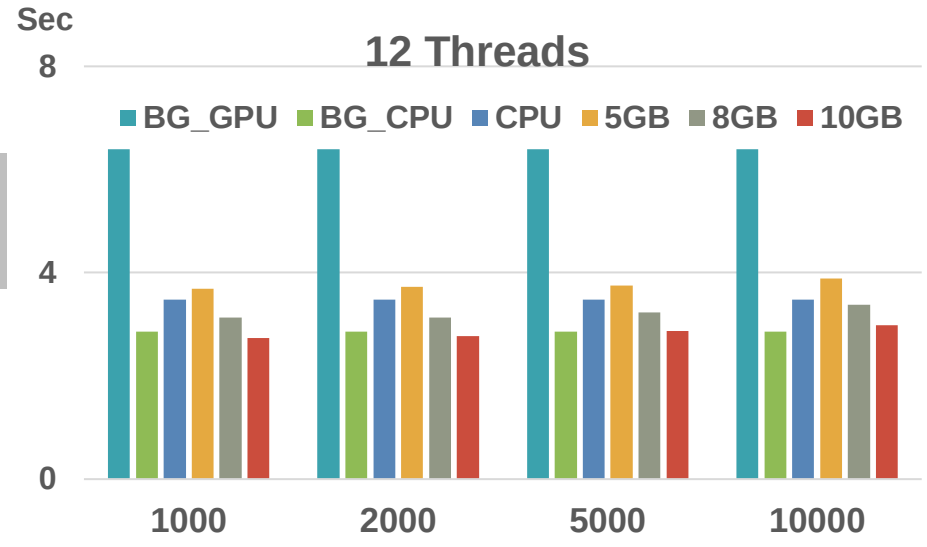
Minimal impact for larger problems

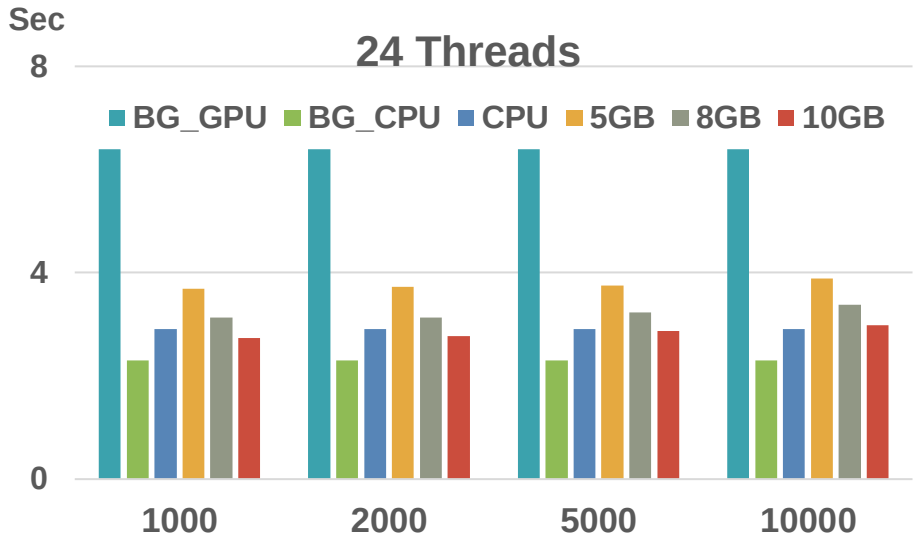
M8000
SZ = 64





M8000
SZ = 96





Conclusions & Lessons Learned

- Portable efficient asynchronous OpenMP GPU tasking still a work in progress
  - Optional `nowait` and multithreaded, concurrent offloading implementation support.
- Implementation defined GPU memory allocation choices need better documentation
  - Memory management in OpenMP 5.0 will help, need documentation.
- Maximum offloading may not be always the best option
  - CPUs can be better at small granularity tasks
- Team-level math (BLAS ..) libraries needed to support "complex" nested parallelism patterns.

Thank You