IWOMP 2020 OpenMP Heterogeneous Programming Section:

Paper # 3

EXASCALE COMPUTING PROJECT

Toward OpenMP MultiGPU Programming with taskloop and User-defined Schedules

Vivek Kale, Wenbin Lu, Anthony Curtis, Abid Malik, Barbara Chapman and Oscar Hernandez

BROOKHAVEN NATIONAL LABORATORY

Stony Brook University
The State University of New York

OAK RIDGE National Laboratory

Thursday, September 24, 2020

4:30 PM CEST / 8:30 AM PDT

IWOMP 2020 in Austin, Texas (Virtual)

State of Scientific Applications and Supercomputers

Class of Science Applications

- Monte-carlo methods with coarse grained, irregular computations and low communication
- Molecular dynamics with load imbalance and irregular communication
- Machine learning methods, similar to Monte Carlo methods.

Evolution of a Node of a Supercomputer:

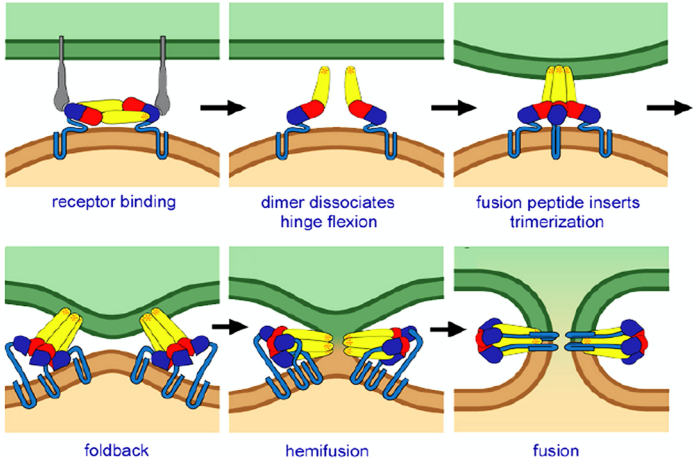
1. 1990s: CPU, i.e., single core
2. 2000s: CPUs, i.e., multicore
3. 2010s: CPUs with GPU, i.e., many-cores
4. 2020: CPUs and GPUs

5. 2030?: multicores + multiple GPUs (manycores) FPGAs + quantum computers

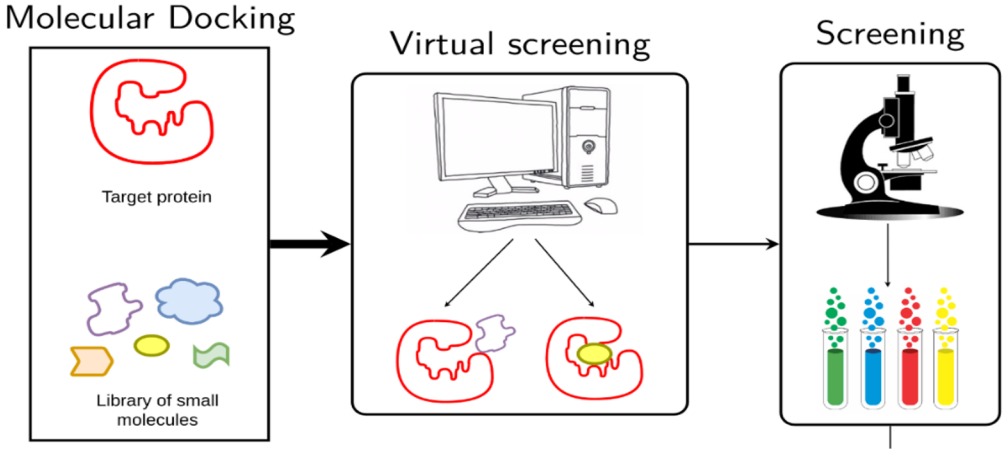
- MPI meant to handle across-node, OpenMP meant to handle within-node
- In these applications, Within-node across-GPU parallelism is especially important, and a programming model like OpenMP is especially beneficial.

Science of AutoDock Application Code

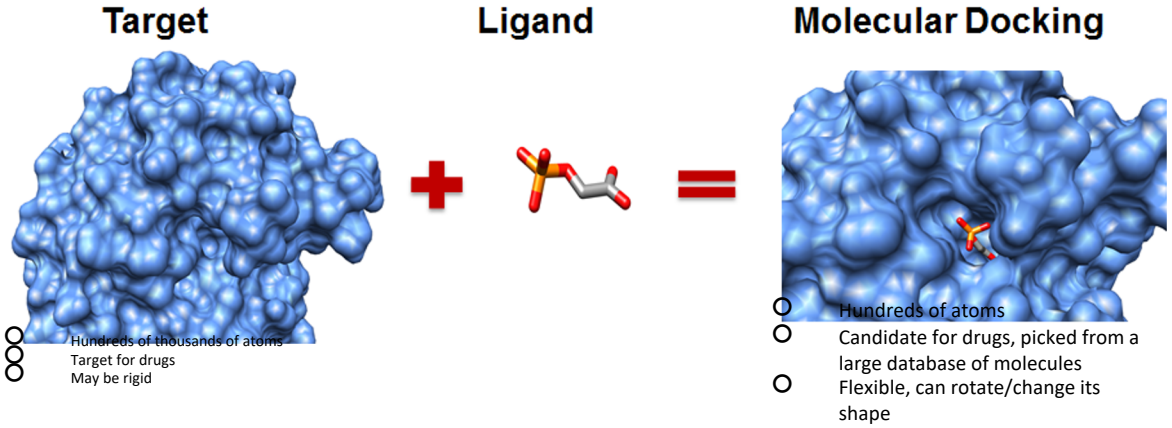
How Viruses Infect Healthy Cells



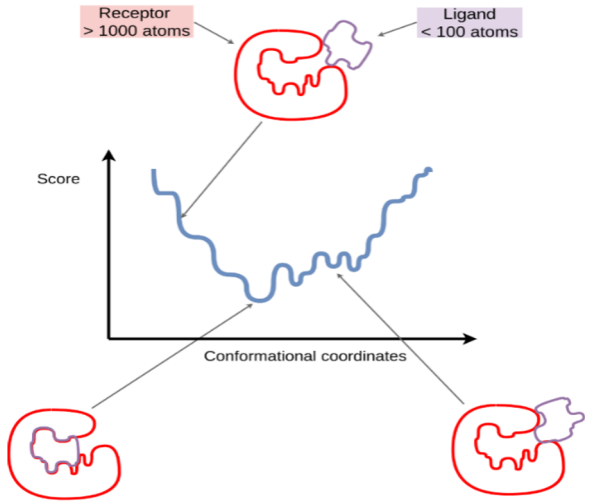
Molecular docking accelerates medical research



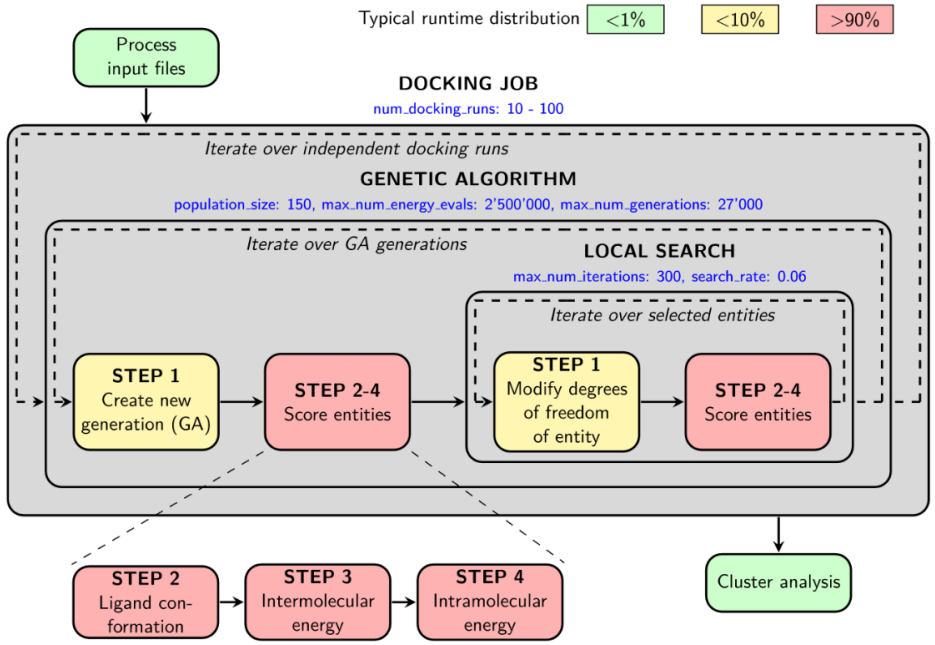
Receptor-Ligand docking



How docking works



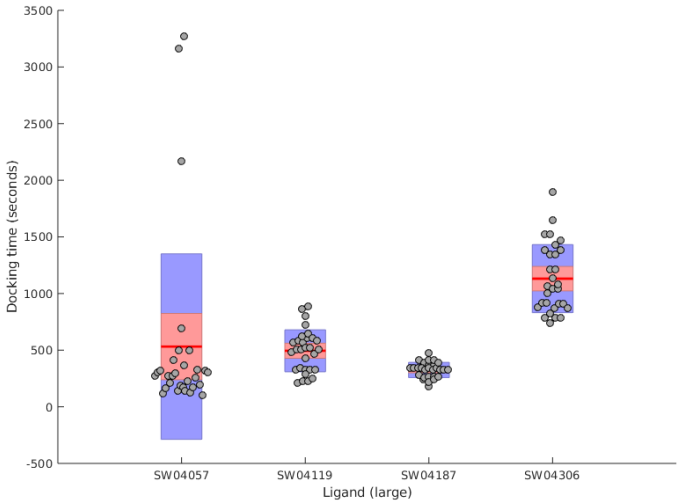
Autodock Genetic Algorithm

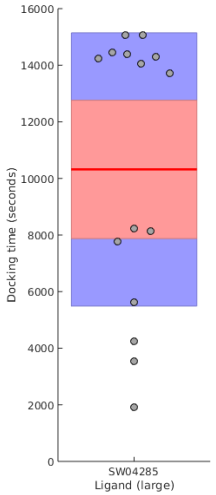


- Receptor proteins are modeled using pre-computed energy grid maps
  - Is constant
  - Saves a lot of time
  - Less accurate
- The number of rotatable bonds in the ligands determines the dimension of the search space
  - Searching through a 40-D space is very difficult!
  - Sometimes you get lucky, other times you don't
- The search will be terminated once one of the following conditions is satisfied
  - The results are clustered around a local minimum (small σ for the current generation)
  - We have reached a large number of generations
  - We have reached a massive number of energy evaluations

AutoDock 4.2 MPI+OpenMP implementation parallelizes across CPUs on the 2nd level of parallelism.

Runs of Autodock on a GPU





Ligand	Rotatable bounds	μ	σ	Coefficient of variation
SW04057	36	531.51s	818.66s	1.54
SW04119	22	494.38s	185.39s	0.37
SW04187	17	324.31s	67.518s	0.21
SW04306	36	1131.1s	300.50s	0.27

→ Expect variations of at least 20%.

- Each GPU kernel/CPU thread handles its own private set of receptor and ligand pairs
  - Completely independent, except for I/O
  - Perfect weak-scaling
- Large variations in docking time creates new issues in multi-GPU setup
  - Static work distribution could lead to idling
  - Motivates need dynamic on-demand multi-GPU scheduling

Other Applications

- DMRG++
  - Main computation is Hamiltonian matrix-vector multiplication
  - Sparse matrix-vector multiplication is source of load imbalanced.
  - The inner matrix vector multiplications can be run on GPUs.
- Monte Carlo Methods, e.g., QMCPack
- Molecular dynamics with load imbalances and irregular communication like miniMD.
- Machine learning, e.g., applications of DoE's CANDLE project.

Representative Benchmark Kernel

```
for (int i = 0; i < arrSize; i++)
  a[i] = 3.0; b[i] = 2.0; c[i] = 0.0;

for (int i = 0; i < numTasks; i++) {
  int work = rand() % probSize;

  output[i] = 0;
#pragma omp target map(to: a[0:n*n], b[0:n*n],
c[0:n*n]) map(tofrom: output[i:1], work) nowait
  {
    const int NN = n * n;
    double work_start = 0;
    for (int j = 0; j < NN; j++)
      c[j] = sqrt(a[j] * b[j]);
    output[i] = c[NN];
  }
}
```

Observations on Benchmark Kernel

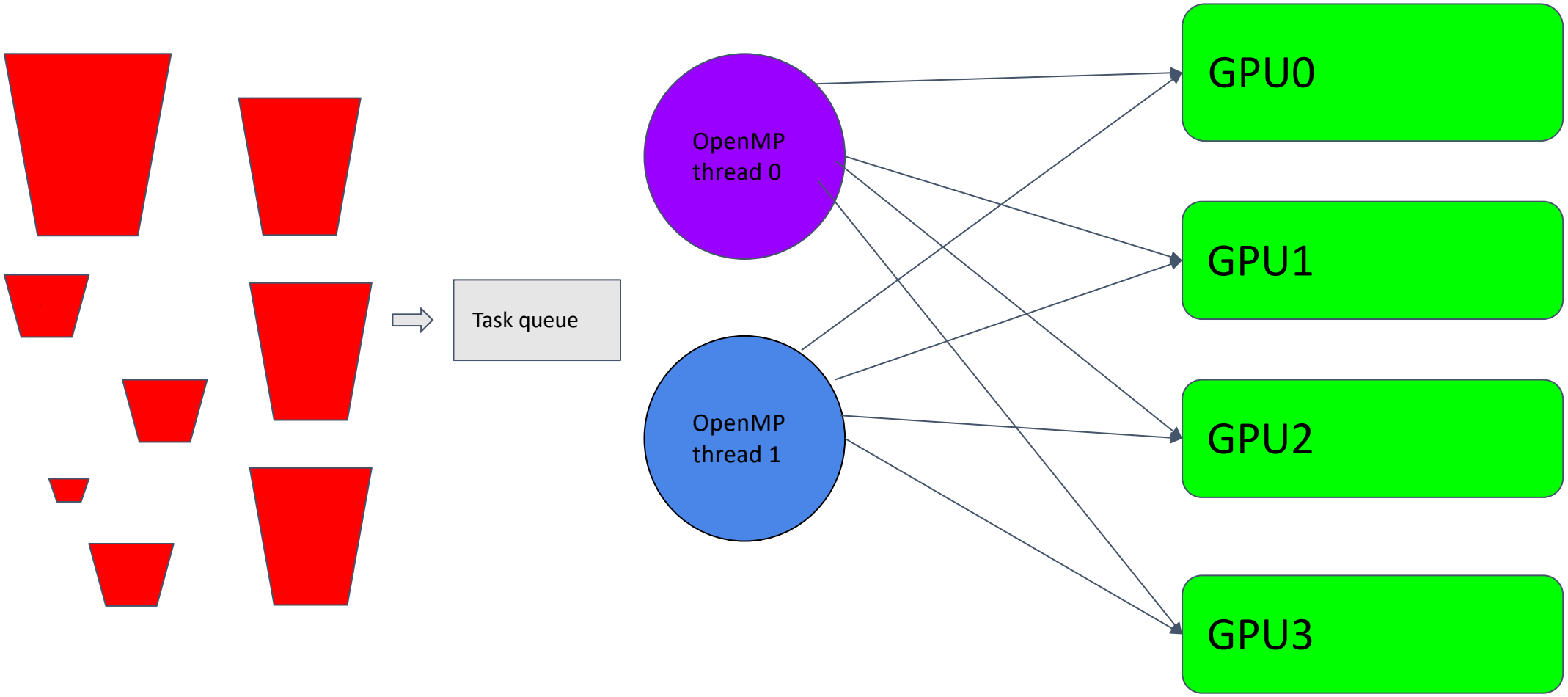
1. Each vector multiplication independent of other, so embarrassingly parallel.
2. Code uses only one GPU on the node. Using multiple GPUs useful considering the baseline performance numbers shown for Autodock.
3. Even if code did use all the GPUs, it wouldn't use the GPUs efficiently due to load imbalance caused by the differently sized computations, in particular given our observations about Autodock.

→ **Objective**: make code use all computational power of the node, specifically, use all the GPUs, all the time, given the application's load imbalance

Key Idea of Our Solution

- A basic way to run OpenMP offload code on multiple GPUs is by pre-assigning each target region of computational work of the application to a device, i.e., GPU, ID.
- To run a set of 100 computations of our benchmark on nodes with 6 GPUs, we can have an OpenMP thread assign the first 17 computations to GPU 0 the next 17 to GPU 1, and so on.
- When running T computations on a node of G GPUs, an OpenMP thread assigns the xth computation to device ID (x*G)/T through adding the parameterized clause `device(x*G/T)` to the `target` construct.
- We call this strategy compact, and it is our baseline strategy.
- **Problem**: load imbalance across, and an under-utilization of, the GPUs of a node.
- **Fix**: Assign, or schedule, computations to GPUs dynamically. To do so, find a way to encapsulate the computations in standardized units of work that can be managed by the OpenMP threads to distribute to the GPUs.
- → We use the OpenMP tasking support already available in OpenMP for this purpose.

Conceptual Idea of Task-to-GPU Strategy



Conceptual Idea of Task-to-GPU Strategy

- OpenMP threads on a CPU manage and schedule an OpenMP task to some GPU in the set of GPUs on a node.
- A `taskloop` construct is applied to the loop that does work in independent outer iterations, each of which contains a target region.
- Red trapezoids: tasks generated from the `taskloop` construct. Grey rectangle represents the queue of `taskloop`.

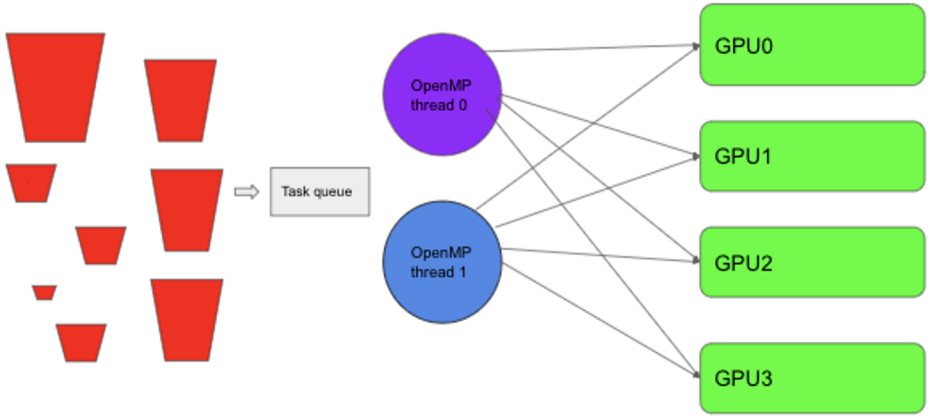


Figure 14: Conceptual diagram of OpenMP threads scheduling tasks to GPUs.

- Each OpenMP thread on the CPU offloads a task of computation in `taskloop` to a particular GPU by dequeueing the next available GPU from a GPU queue, which is stored on the host.
- Note that this GPU queue does not perform cross-GPU synchronizations, thus avoiding GPU-to-GPU communication before each execution of a task.

Code Transformation for Task-to-GPU Scheduling

```
#pragma omp parallel
{
#pragma omp single
  {
#pragma omp taskloop shared(success)
    for (int i = 0; i < numTasks; i++) {
      const int dev = gpu_scheduler_dyn(occupancies, ndevs);
      output[i] = 0;
#pragma omp task depend(out : success[i])
      {
        success[i] = 0;
      }
#pragma omp task depend(inout : success[i])
      {
#pragma omp target device(dev)                                  \
        map(to: a[0:arrSize], b[0:arrSize], c[0:arrSize])       \
        map(tofrom: success[i:1], output[i:1], taskWork[i:1],
    occupancies[dev:1])
        {
          devices[dev]++;
          if (taskWork[i] > probSize) taskWork[i] = probSize;
          const int NN = taskWork[i];
          output[i] = doWork(c, a, b, taskWork[i]);
          success[i] = 1;
        }
      }
#pragma omp task depend(in : success[i])
      {
#pragma omp atomic
        occupancies[dev]--;
      }
    }
  }
}
```

```
inline unsigned gpu_scheduler_X(type1 param1, type2 param2, type3 param3)
{

}
```

Parametrized Interface for UD Task-to-GPU schedule

```
inline unsigned gpu_scheduler_X(loop_record* lr)
{

}
```

Pointer-based Interface for UD Task-to-GPU schedule

User-defined Task-to-GPU Schedules

```
inline unsigned gpu_scheduler_sta(unsigned *occupancies, int taskID, int ngpus, int numTasks)
{
  const unsigned chosen = (unsigned) ((taskID*ngpus)/(float) numTasks);
  #pragma omp atomic
    occupancies[chosen++];
  return chosen;
}
```

Static

```
inline unsigned gpu_scheduler_rrb(unsigned *occupancies, int taskID, int ngpus)
{
  const unsigned chosen = (unsigned) taskID%ngpus;
  #pragma omp atomic
    occupancies[chosen++];
  return chosen;
}
```

Round-robin

```
inline unsigned gpu_scheduler_ran(unsigned *occupancies,, int ngpus)
{
  const unsigned chosen = (unsigned) taskID%ngpus;
  #pragma omp atomic
    occupancies[chosen++];
  return chosen;
}
```

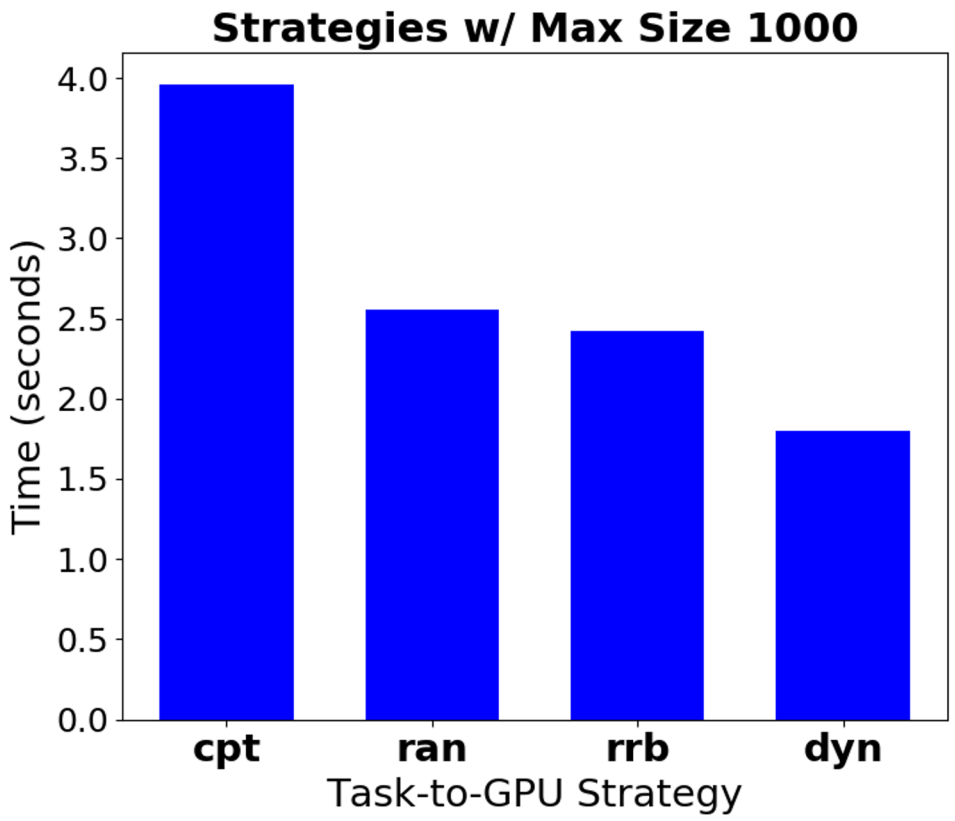
Random

```
inline unsigned gpu_scheduler_dyn(unsigned *occupancies, int ngpus)
{
 short looking = 1;
 unsigned chosen;
 while (looking) {
 for (unsigned i = 0; i <ngpus; i++)
  {
   unsigned occ_i;
   #pragma omp atomic read
              occ_i = occupancies[i];
       if (occ_i == 0) {
             chosen = i;
             occupancies[chosen]++;
             looking = 0;
             break;
      }
    }
   }
  return chosen;
}
```

Dynamic-opt

Results for Task-to-GPU Strategies

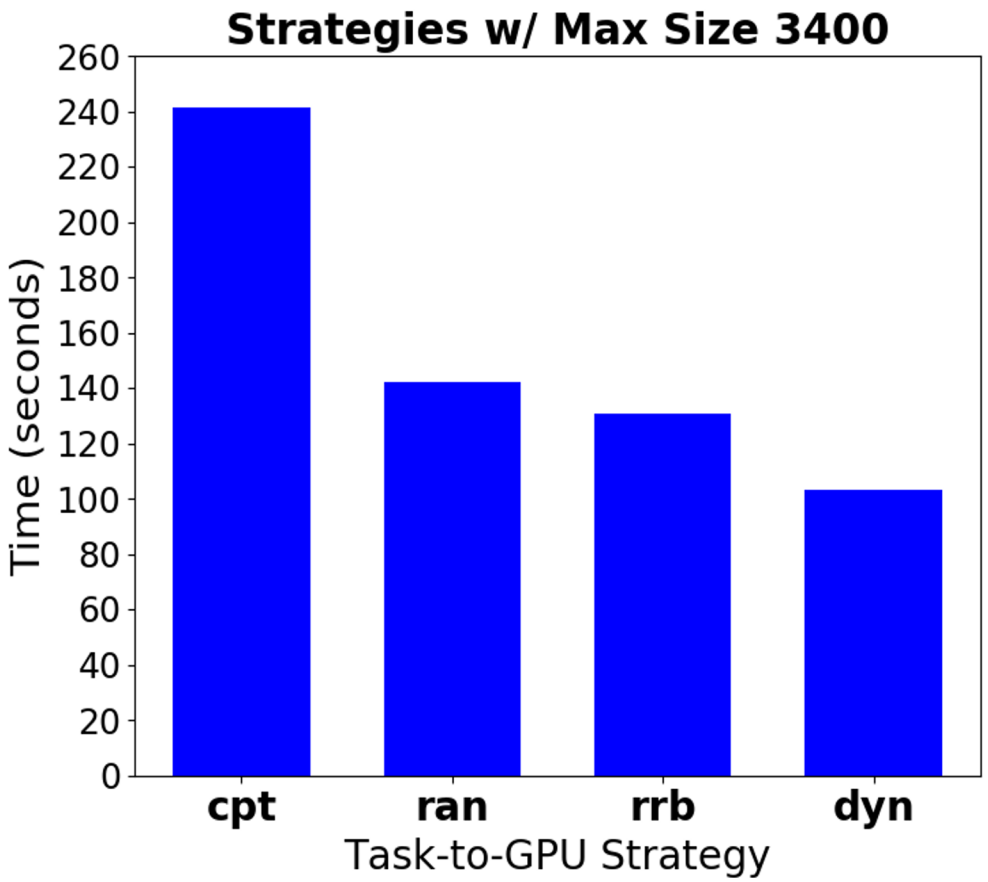
- Ran a uniformly random distribution of a square rooted vector multiplication.
- Ran with Clang 10 with 4 OpenMP thread, one node of Seawulf (42 AMD CPU cores, 8 NVIDIA Tesla Kepler GPUs)



1. Compact has load imbalance so likely performs badly because of that.
2. Round-robin also has bad load imbalance, and similar problem with random.
   - The performance improvement could come from reduced pressure on PCIe
3. Impact of load balancing is 0.5 seconds seen through Dynamic task to GPU scheduling.

Results for Task-to-GPU Strategies

- Ran a uniformly random distribution of a square rooted vector multiplication.
- Again ran with Clang 10 with 4 OpenMP threads, one node of Seawulf (42 AMD CPU cores, 8 NVIDIA Tesla Kepler GPUs).



- Compact again has load imbalance so likely performs badly because of that.
- Round-robin also has bad load imbalance, and similar problem with random.
- Impact of load balancing is 20 seconds seen through dynamic task to GPU scheduling
- → Task-to-GPU scheduling helps improve performance by reducing data movement cost along with handling load imbalance.

Performance Monitoring

- Need to understand
  - GPU utilization
  - Degree of load imbalance
  - Overhead (runtime, scheduler, task affinity)
- Manual instrumentation with CUDA Profiling Tools Interface (CUPTI)
  - CUDA invokes user-defined callbacks to record start/finish timestamps of various events
  - High-level activities categories: **DEVICE, CONTEXT, DRIVER, RUNTIME, MEMCPY, MEMSET, KERNEL, OVERHEAD**
    - Similar to the results shown in nvprof, just not aggregated
  - Low-level events: L2 cache, FLOPS, instructions, warp activity, shared-memory, atomics, etc.

Performance Profiling with CUPTI

Used smaller size due to profiling overheads of CUPTI

Scheduler	**DRIVER**	**MEMCPY**	**OVERHEAD**
Compact	2.3636	0.0739	0.1894
Random	1.6606	0.0852	0.2485
Roundrobin	1.7758	0.0808	0.2083
Dynamic	1.4243	0.0635	0.1717

Extend OpenMP for Task-to-GPU Scheduling

- ***Key question***: how do we extend OpenMP to support of task scheduling for multi-GPUs, for ease of use by application programmers?
- Proposal of `taskloop target` construct : single OpenMP construct to offload asynchronous target regions on multiple device
- Extend `taskloop` construct to handle the scheduling of target regions to GPUs and avoid additional levels of nested tasks → approach could have less overhead and less programmer effort.
- Augment User-defined Schedules proposal for OpenMP to handle OpenMP task-to-GPU scheduling.
- Task-to-GPU affinity supported by specialized affinity clause to improve data locality, specifically less CPU-GPU data movement

Related Work

- Several methods deal with programming multiple accelerators:
  - Spawn multiple processes using MPI on host, with each process dealing with one accelerator[potluri-2012,michael-2014]
  - Spawn multiple threads using OpenMP on host, and each thread deals with one accelerator [guan-2013,jame-2016,michael-2014]
  - Project on OpenMP offload regions to multiple GPUs, comparison with MPI handling the parallelization of work across GPUs on a node.
- Compilers (OpenACC extensions) and languages (XcalableACC[Nakao-2017]) to generate a code for multiple accelerators from a code for a single accelerator automatically. For example, a loop statement that can be divided is composed only of affine access.
- Our work allows for sophisticated scheduling strategies that the user to define within the application code or be automatic.
- Task-to-GPU load balancing and communication:
  - Xu et al. [xu-2016] propose an OpenACC extension to support multiple accelerators.
  - Komoda et al. [komoda-2013] propose another OpenACC extension that supports dividing data and tasks into multiple accelerators.
  - OpenACC extensions for loop partitioning across GPUs.
  - Compiler has a mechanism to keep data consistency on the accelerator memory automatically. The OpenACC extension can be used only before the loop statement. So, the OpenACC extension cannot offload data to an arbitrary device, as in our work.
  - Scogland et al developed directive extensions to support scheduling work on multiple GPUs and multi-cores using a runtime called coreTStar. The extension partitions loop iterations and its data across multiple devices and CPU threads.

Conclusions

- Presented methods to use all GPUs of a node of an HPC cluster efficiently through OpenMP, particularly for applications that are embarrassingly parallel and load imbalanced, which are characteristics of the computational pattern of Monte Carlo Methods and exemplified by the applications Autodock and DMRG++.
- Our solution involves encapsulating each OpenMP target region containing a computation within an OpenMP task, and then having OpenMP threads assign the OpenMP tasks to GPUs on a node through a user-level task-to-GPU schedule.
- Through experimenting with our approach, results provide up to a 57.2% performance improvement → suggest the usefulness of OpenMP tasking across GPUs on a node.

Conclusion

- **Problem:** Explore how to program multiple GPUs within a node by looking at different task-to-GPU scheduling strategies to map computations to multiple devices.
- **Our fix:** Our solution involves using OpenMP's tasking construct task loop to generate OpenMP tasks containing target regions for OpenMP threads, and then having OpenMP threads assign or schedule those tasks to GPUs on a node through a schedule specified by the application programmer, or a use such as a performance engineer helping optimize an application.
- **What we did:** We analyze the performance of our solution using a small OpenMP performance benchmark code representative of the applications with Monte Carlo methods, in particular AutoDock [15] and DMRG++ [13].
- **What we got:** Applying our solution to our benchmark, we improve performance over a standard baseline assignment of tasks to GPUs up to 57.2%.
- **Impact and Extensions:** Based on our results, we suggest OpenMP extensions that could help application programmers have their applications use multiple GPUs per node efficiently through OpenMP.

Ongoing Work

- Experiment with miniApp developed through OpenMP performance benchmarking tool. Experiment with our technique in other miniApps.
- Combines scheduling tasks across GPUs with scheduling of Thread Blocks to Stream Multiprocessors (SMs), i.e., scheduling within aGPU.
- work to propose new extensions in OpenMP, particularly implementing the the LLVM OpenMP compiler and supporting OpenMP implementations that allow users to easily use our approach.
- Investigate impact of and tune the taskloop's grain-size. Also, consider variable task sizes.
- look at using or adapting the affinity clause to easily reduce data movement overheads of our task-to-GPU scheduling strategies. The affinity clause will give a hint to the task scheduler about placing a task on the most appropriate GPU based on the GPU context.

Future Work

- Experiment with miniApp developed through OpenMP performance benchmarking tool.
  - Plan for miniApp development: https://tinyurl.com/miniAppADplan
- Experiment with our technique in combination with MPI or OpenShmem, make adjustments based on information from MPI or OpenShmem.
- Adapting the scheduler based on previous executions of target regions.

Acknowledgements (1)

- This research was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, in particular its sub-projecton Scaling OpenMP with LLVm for Exascale performance and portability (SOLLVE).

- It is also supported in part by NSF project 1409946 ”Compute on DataPath”. This material is based upon work supported by the U.S. Departmentof Energy, Office of Science, Office of Advanced Scientific Computing Research, under contract number DE-AC05-00OR22725.

Acknowledgements (2)

- This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.
- We thank Stony Brook Research Computing and Cyberinfrastructure, and the Institute for Advanced Computational Science at Stony Brook University for access to the high-performance SeaWulf computing system, which was made possible by a $1.4M National Science Foundation grant (#1531492).
- Thanks to Jeremy Smith and Ada Sedova, from Oak Ridge National Laboratory, for providing a small sample of input sets for the Autodock-GPU experiments to help us study the application workload.
- We acknowledge the QMCPACK team at ORNL for discussing their code with respect to application load imbalances.

Works Cited (1)

- [1] OpenMP 5.0 Reference Guide.https://www.openmp.org/wp-content/uploads/OpenMPRef-5.0-1119-01-TSK-web.pdf.
- [2] OpenMP Verification and Validation Suite. https://github.com/SOLLVE/sollve_vv.
- [3] Parallel Computational Pattern: Monte Carle Methods. https://patterns.eecs.berkeley.edu/?page_id=186.
- [4] Perlmutter User Guide. https://www.nersc.gov/systems/perlmutter/.
- [5] Summit User Guide. https://docs.olcf.ornl.gov/systems/summit_user_guide.html.
- [6] The LLVM Compiler Infrastructure. http://llvm.org/.
- [7] Optimizing MPI Communication on Multi-GPU Systems Using CUDA Inter-Process Communication, 2012.

Works Cited (2)

- [8] MACC: An OpenACC Transpiler for Automatic Multi-GPU Use. In Supercomputing Frontiers, page 109–127. Springer International Publishing, Cham, 2018.
- [9] J. Beyer and B. R. de Supinski. IWOMP 2016 Tutorial: OpenMP Accelerator Model. http://iwomp2016.riken.jp/wp-content/uploads/2016/10/tutorial-accelerator.pdf, 2016.
- [10] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In Journal of Parallel and Distributed Computing, 1995.
- [11] J. M. Bull. Measuring Synchronisation and Scheduling Overheads in OpenMP. In Proceedings of First European Workshop on OpenMP, pages 99–105, Lund, Sweden, 1999.

Works Cited (3)

- [12] F. M. Ciorba, C. Iwainsky, and P. Buder. OpenMP Loop Scheduling Re-visited: Making a Case for More Schedules. ArXiv, abs/1809.03188, 2018.
- [13] J. Criado, M. Garcia-Gasulla, J. Labarta, A. Chatterjee, O. Hernandez,R. Sirvent, and G. Alvarez. Optimization of condensed matter physics ap-plication with openmp tasking model. In X. Fan, B. R. de Supinski, O. Sinnen, and N. Giacaman, editors,OpenMP: Conquering the Full Hardware Spectrum, pages 291–305, Cham, 2019. Springer International Publishing.
- [14] S. Donfack, L. Grigori, and A. Gupta. Adapting Communication-Avoiding LU and QR Factorizations to Multicore Architectures. In2010 IEEE International Parallel and Distributed Processing Symposium, pages 1–10, Atlanta, GA, USA, April 2010.
- [15] M. G. M. O. A. J. Huey, R. and D. S. Goodsell. A semiempirical free energy force field with charge-based Desolvation. Journal of Computational Chemistry, pages 28: 1145–1152, 2007.
- [16] J. Guan and S. Yan and J. M. Jin. An OpenMP-CUDA Implementation of Multilevel Fast Multipole Algorithm for Electromagnetic Simulation on Multi-GPU Computing Systems, 2013

Works Cited (4)

- [17] L. Kale and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, Proceedings of OOPSLA'93, pages 91–108. ACM Press, September 1993.
- [18] V. Kale, C. Iwainsky, M. Klemm, J. H. M. Kornd˙orfer, and F. M. Ciorba.Toward a Standard Interface for User-Defined Scheduling in OpenMP. In International Workshop on OpenMP, pages 186–200. Springer, 2019.
- [19] J. Kim, A. D. Baczewski, L. Zhao. QMCPACK: an Open Source ab initio Quantum Monte Carlo Package for theElectronic Structure of Atoms, Molecules and Solids.Journal of Physics:Condensed Matter, 30(19):195901, apr 2018.
- [20] T. Komoda, S. Miwa, H. Nakamura, and N. Maruyama. Integrating Multi-GPU Execution in an OpenACC Compiler. InIn 2013 42nd InternationalConference on Parallel Processing, page 260–269, 2013.
- [21] C. B. Leopold Grinberg and R. Haque. Hands on with openmp4.5 and uni-fied memory: Developing applications for ibm's hybrid cpu + gpu systems(part ii), 2017.

Works Cited (5)

- [22] Morris, G. M., Huey, R., Lindstrom, W., Sanner, M. F., Belew, R. K., Good-sell, D. S. and Olson, A. J. Autodock4 and AutoDockTools4: AutomatedDocking with Selective Receptor Flexiblity, 2009.[23] M. Nakao, H. Murai, H. Iwashita, A. Tabuchi, T. Boku, and M. Sato. Im-plementing Lattice QCD Application with XcalableACC Language on Ac-celerated Cluster. page 429–438, 2017.
- [24] O. Trott, A. J. Olson. AutoDock Vina: Improving the Speed and Accu-racy of Docking with a New Scoring Function, Efficient Optimization andMultithreading, 2010.[25] T. R. Scogland, W.-C. Feng, B. Rountree, and B. R. Supinski. CoreTSAR: Adaptive Worksharing for Heterogeneous Systems. In Proceedings of the29th International Conference on Supercomputing - Volume 8488, ISC 2014,page 172–186, Berlin, Heidelberg, 2014. Springer-Verlag.
- [26] P. Tandon and D. E. Rosner. Monte Carlo Simulation of Particle Aggre-gation and Simultaneous Restructuring.Journal of Colloid and InterfaceScience, 213(2):273 – 286, 1999.
- [27] M. Wolfe. Scaling OpenACC Applications Across Multiple GPUs, 2014.
- [28] Xu, Rengan and Tian, Xiaonan and Chandrasekaran, Sunita and Chapman, Barbara. Multi-GPU Support on Single Node Using Directive-basedProgramming Models, January 2016

Questions?