ECE 563 Programming Parallel Machines
Large Project
Kumaraguru Sivasankaran

Note: unit of time considered is seconds. All experiments done in HalsteadGPU cluster which hosts 4 nodes with 20 cores per node, 2 GPUs per node. Detailed configuration details available here: https://www.rcac.purdue.edu/compute/halsteadgpu
GPU model: Tesla P100-PCIE-16GB with 56 SMs and 1024 max. number of threads.
Cuda version: 8.0

Motivation:

Increasing requirements of computational power requires us to use multiple compute devices for achieving performance. A heterogenous platform like CPU+GPU is a preferred solution for General purpose hardware accelerators. Usually, the CPU takes in the preprocessing part and prepares to feed the high throughput of GPU which is an application overload. Given the complexity of the memory management in CPU-GPU architecture, it is important to study the combination of GPU-CPU. While the possible advantages of heterogeneity are overall increase in performance, better work load distribution, exploiting concurrency, data communication, energy efficiency etc. the major challenges remain to be programming and workload partitioning.

There are two programming models available for heterogeneous computing (mix of different processors on same platform):

1. Mix of programming models like using one for CPUs (any general programming) and one for GPUs (usually CUDA) which are then combined using a Message Passing Interface (MPI).
2. Single unified programming model like using high level models- OpenMP 4.0, OpenACC

While the first one offers flexibility and useful for enhancing performance, the second one offers productivity and easy to implement. Through this work we focus on the first paradigm and will be the subject of discussion throughout the report. We will analyze some of the critical parameters that help to achieve peak performance using a combined architecture. A small portion is devoted to isoefficiency, karp-flatt analysis but a thorough study is beyond the scope of this report.

Problem Statement:

To study the performance of hardware accelerators, we turn to some classic problems which can give clear differentiation and offers flexibility in implementation. In physics, the n-body problem is the problem of predicting the individual motions of a group of objects, may be celestial or particles, interacting with each other based on some fundamental forces. Originally, this was motivated after the desire to understand and predict the motion of sun, moon and stars. But considering the complexity, it is more difficult to solve. But today's world, solving this problem is solution to many applications in machine learning like finding sum of kernel functions over all pairs of objects, where the kernel function depends on the distance between

objects in parametric space. The popular algorithms like Support Vector Machines (SVM), K-Nearest Neighbors etc. are good examples of problem under study.

In the simplest words, the problem can be defined as follows: "Given the instantaneous position and velocity of the n-bodies, predict the interactive forces acting on them and consequently, predict the motions of the objects in the future."

Using power series solution, we have the formula, after simplifying from Newton's law of gravity,

$$a = G \sum_{i=1,j=1}^{i=n,j=n} \frac{m_j * (x_j - x_i)}{\left((x_j - x_i)^2 + \epsilon^2\right)^{\frac{3}{2}}}$$

Where ϵ is a softening factor to allow for i=j and preserving numerical stability.
Considering particles of unit mass, we can get the summation of forces acting per particle and integrate it over time to get velocity and position. For n bodies, this comes down to $O(n^2)$ for the force calculation and O(n) for integrating the position. Considering the time constraints, this algorithm suited to fit all the below implementation methods and hence, chosen for study. Efficient and complicated algorithms exist for solving the same problem but does not align with motivations of this study.

Implementation details and important results:

Following are the cases considered in this study for a fixed problem size of 2^12:

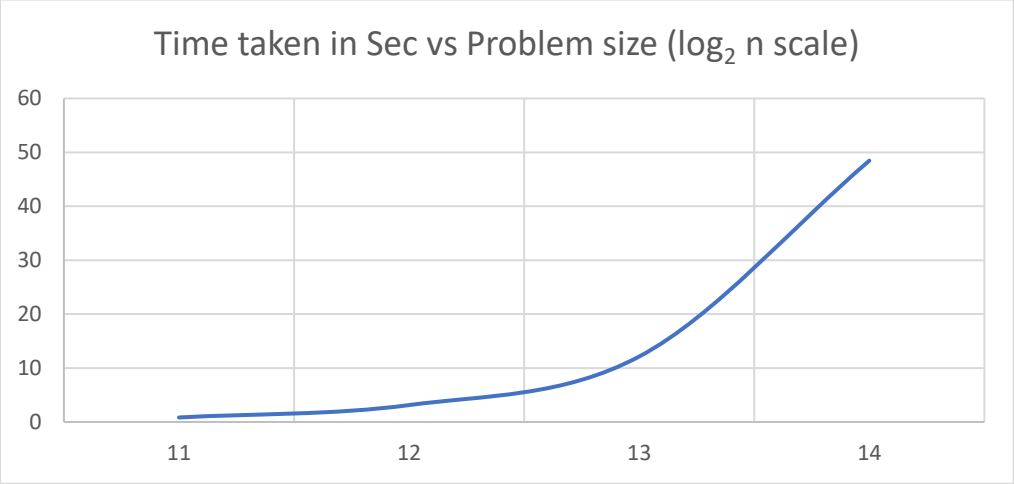
#	Implementation detail	Architecture	Obtained speed up
1	Sequential implementation of the algorithm for baseline comparison	1-CPU	0.05 billion operations per sec
2	Omp based implementation of the algorithm	Multi-CPU (shared memory)	0.18 billion operations per sec (for 8 CPU)
3	MPI based implementation of the algorithm	Multi-CPU (distributed memory)	0.33 billion operations per sec (for 16 CPU)
4	CUDA based implementation	1 CPU- multi-GPU	~0.525 billion operations per sec
5	CUDA based implementation	1 CPU-1 GPU	~33 billion operations per sec
6	MPI- CUDA based implementation	Multi-CPU (distributed memory) and multi-GPU	~0.21 billion operations per sec

Although each of the implementation had starting up issues, the MPI-CUDA based implementation was essentially the hardest to implement. This was due to several data transfers between nodes and then, from host processor to device (Following usual conventions, host refers to CPU and device refers to GPU).

I. Sequential case

Problem size (2^n)	Time	Billions of Operations per Second

2^12	3.13	0.054534
2^13	12.138727	0.055285
2^14	48.458520	0.055395
2^16	0.831188	0.050462



II. MPI based implementation:

Here, the n-bodies are split into p-processors and each of the p processor calculates the values for the n-bodies belonging to it. Since the calculation requires the position of rest of the bodies, a collective communication routine like MPI_AllGather is used.

Problem size (2^n)	No. of CPU cores	Time taken in Sec	Billions of Ops per second	Speed up	Efficiency	Karp Flatt metric
2^12	2	2.26795	0.073975	1.380101	0.69005	2.449169
2^12	4	1.298984	0.129156	2.409575	0.602394	0.886681
2^12	16	0.498589	0.336494	6.277716	0.392357	0.23658
2^12	32	0.261366	0.641905	11.97554	0.374236	0.118455

Iso-efficiency and karp-flatt analysis:

(Consider the machine parameters as unit)

Algorithm complexity in sequential (asymptotic analysis): $T_s = t_c\, n^2$

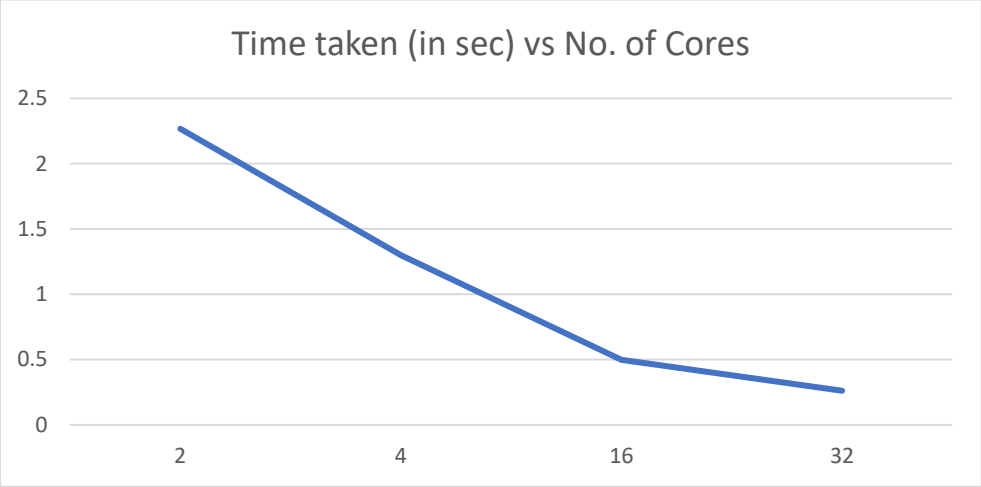
MPI version: $T_p = t_c \frac{n^2}{p} + (t_s + t_w \frac{n}{p})\ log\ p$

Total overhead: $T_o = T_s - pT_p = p\ log\ p + n \log p = W = n^2$

Hence, $W = n^2 \sim p \log p$ and $W = n^2 \sim n \log p$

For larger n, the second term starts to dominate as the amount of buffer transferred starts to dominate. This is evident from above results.

The serial fraction starts to reduce as number of processors are increased which shows scalability of this algorithm for multi-cpu case. It is obvious as the number of processors are increased, the work per processor essentially reduces due to the division of serial work.



III. 1 CPU-1GPU case:

In this implementation, we have a base function which runs on the host and creates random initialization as creating random numbers in GPU is not possible currently. The generated data is transferred to GPU device where simulations runs calling two CUDA kernels. We use synchronize to create barriers between kernels. Finally, the results are copied back to host and memory is freed.

Optimizing the right number of threads and blocks is essential at this step for achieving maximum performance. Based on literature, it is understood that the blocks are simultaneously scheduled by the Streaming multiprocessors. From the device specification, it was found that the Tesla P100-PCIE-16GB had 56 SMs. Two experiments were conducted:

1. To create number of blocks in the multiples of number of Streaming processors.
2. To create number of blocks by fixing the number of threads and covering the entire iteration space using the formula.

$$nblocks = \frac{iteration\ space + nthreads - 1}{nthreads}$$

Problem size	Threads per block	Blocks	Billion Ops per second	Time in sec	Speed up
2^12	128	32	33.548	0.005049	619.9247
	512	8	31.377	0.005347	585.375
	74	56	32.295	0.005195	602.5024
	32	128	32.376	0.005182	604.0139
	1024	4	17.683	0.009488	329.8904
	512	112	32.602	0.005146	608.2394
	1024	112	32.64	0.00514	608.9494

	256	112	33.846	0.004957	631.4303
	128	112	33.229	0.005049	619.9247
	64	64	33.078	0.005072	617.1136
	64	112	32.552	0.005154	607.2953
	256	32	33.548	0.005001	625.8748

Although it is not possible to conclude based on small number of experiments, we can see the case where block size very small compared to number of SMs gave drastically low performance. Further, when the number of blocks were multiples of number of SMs, the performance was consistent. It should be noted that, in this problem, we have not employed special strategy to exploit data reuse. It would be of next immediate step for optimizing further. One way to do would be to create three dimensional blocks and using shared memory for reducing access from GPU global memory. Nevertheless, by localizing the data to GPU in the first hand, we have already reduced the host to device transfer per iteration which is the primary cause of current speed up over sequential.

IV. Comparison using 1-CPU-1GPU vs 1-CPU-2GPU:

We used nvprof – NVidia profiling tool for understanding the problems in scaling up and diagnosing the load balance issues etc. Below is an example of profiling done for two cases having same number of threads and blocks. In this problem, using two GPU devices did not help as much as one would have thought. We can see the average operations at 0.525 billion ops per second against 31.99 billion ops per second in one device case. Moving further in the report, we can find the kernel – bodyForce taking almost same time in both case. There is no difference in CPU page faults as the device almost holds the data after first initialization. The reason can be identified from the API calls information where we can see the cudaMemPrefetchAsync is called 22 times against 2 time in 1 device case. It takes 45% of time in the API calls which explains the sluggish. Since data transfer dominates over computation, we find this effect.

Comparisons of two outputs from running nvprof (nvidia profiling tool)

```
==20042== NVPROF is profiling process 20042, command: ./nbodysol.o
SMs 56 2
Threads 256 Blocks 16
Simulator is calculating positions correctly.
4096 Bodies: average 0.525 Billion Interactions / second
0.319435, 0, -1
==20042== Profiling application: ./nbodysol.o
==20042== Profiling result:
Time(%)     Time    Calls      Avg      Min      Max  Name
 99.35%  3.9578ms       10  395.78us  395.13us  397.47us  bodyForce(Body*, float, int)
  0.65%  26.048us       10  2.6040us  2.4320us  3.9040us  position(Body*, float, int)

==20042== Unified Memory profiling result:
Device "Tesla P100-PCIE-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
       1  96.000KB  96.000KB  96.000KB  96.00000KB  11.07200us  Host To Device
Device "Tesla P100-PCIE-16GB (1)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
       2  32.000KB  4.0000KB  60.000KB  64.00000KB  6.624000us  Device To Host
Total CPU Page faults: 1

==20042== API calls:
Time(%)     Time    Calls      Avg      Min      Max  Name
 53.32%  370.88ms        1  370.88ms  370.88ms  370.88ms  cudaMallocManaged
```

```
==20185== NVPROF is profiling process 20185, command: ./nbodysol.o
SMs 56 2
Threads 256 Blocks 16
Simulator is calculating positions correctly.
4096 Bodies: average 31.999 Billion Interactions / second
0.005243, 0, -1
==20185== Profiling application: ./nbodysol.o
==20185== Profiling result:
Time(%)     Time    Calls      Avg      Min      Max  Name
 99.51%  3.9548ms       10  395.48us  394.72us  396.86us  bodyForce(Body*, float, int)
  0.49%  19.296us       10  1.9290us  1.8560us  2.4640us  position(Body*, float, int)

==20185== Unified Memory profiling result:
Device "Tesla P100-PCIE-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
       1  96.000KB  96.000KB  96.000KB  96.00000KB  12.22400us  Host To Device
       2  32.000KB  4.0000KB  60.000KB  64.00000KB  6.912000us  Device To Host
Total CPU Page faults: 1

==20185== API calls:
Time(%)     Time    Calls      Avg      Min      Max  Name
```

45.34%	315.37ms	22	14.335ms	19.266us	313.64ms	cudaMemPrefetchAsync
0.62%	4.3130ms	20	215.65us	9.1410us	567.77us	cudaDeviceSynchronize
0.33%	2.2861ms	2	1.1431ms	1.1421ms	1.1440ms	cuDeviceTotalMem
0.15%	1.0648ms	182	5.8500us	357ns	205.89us	cuDeviceGetAttribute
0.13%	927.01us	20	46.350us	25.492us	138.04us	cudaLaunch
0.08%	560.63us	1	560.63us	560.63us	560.63us	cudaFree
0.01%	96.345us	2	48.172us	44.166us	52.179us	cuDeviceGetName
0.00%	23.584us	20	1.1790us	777ns	4.2760us	cudaSetDevice
0.00%	18.716us	60	311ns	140ns	4.4270us	cudaSetupArgument
0.00%	16.225us	1	16.225us	16.225us	16.225us	cudaGetDevice
0.00%	11.507us	20	575ns	278ns	3.8060us	cudaConfigureCall
0.00%	5.9810us	3	1.9930us	461ns	4.1970us	cuDeviceGetCount
0.00%	4.6220us	6	770ns	489ns	1.2190us	cuDeviceGet
0.00%	4.1600us	1	4.1600us	4.1600us	4.1600us	cudaGetDeviceCount
0.00%	1.5670us	1	1.5670us	1.5670us	1.5670us	cudaDeviceGetAttribute

97.71%	433.85ms	1	433.85ms	433.85ms	433.85ms	cudaMallocManaged
0.93%	4.1338ms	20	206.69us	9.9930us	403.89us	cudaDeviceSynchronize
0.52%	2.2953ms	2	1.1477ms	1.1426ms	1.1527ms	cuDeviceTotalMem
0.29%	1.2715ms	2	635.75us	162.18us	1.1093ms	cudaMemPrefetchAsync
0.25%	1.1147ms	182	6.1240us	360ns	219.41us	cuDeviceGetAttribute
0.22%	995.01us	20	49.750us	35.730us	153.16us	cudaLaunch
0.04%	164.16us	1	164.16us	164.16us	164.16us	cudaFree
0.02%	101.14us	2	50.571us	46.789us	54.354us	cuDeviceGetName
0.01%	33.001us	60	550ns	388ns	4.6040us	cudaSetupArgument
0.00%	16.221us	1	16.221us	16.221us	16.221us	cudaGetDevice
0.00%	16.162us	20	808ns	541ns	4.4810us	cudaConfigureCall
0.00%	7.3470us	3	2.4490us	699ns	5.3250us	cuDeviceGetCount
0.00%	4.7580us	6	793ns	454ns	1.2800us	cuDeviceGet
0.00%	4.1200us	1	4.1200us	4.1200us	4.1200us	cudaGetDeviceCount
0.00%	1.3730us	1	1.3730us	1.3730us	1.3730us	cudaDeviceGetAttribute

V. MPI-CUDA implementation:

Here, we have two programs separately written – one using MPI and another using CUDA with base in C programming. The two programs are linked using MPI-aware impi complier. The MPI part of the program dealt with initializing the communication world for CPUs and serving as a gateway to transfer data between nodes to host and then host to GPU.

From experiments, we can see that all processors in a node has access to both GPUs, but we restrict usage to one per processor. This makes one GPU shared by 5 CPUs, but the memory and operations are performed independently without sharing. This would be a hypothetical case where data is stored decentralized and only required information shared between nodes. Here, we find there is lot of data transfer happening with barriers (in host) and synchronizers (in device) making the process less efficient.

Problem size (2^n)	No. of CPU cores	No. of GPUs used per core (although 2 available)	Time taken in Sec	Billions of Ops per second	Speed up
2^12	8	1	0.789548	0.212491	14.73004
2^12	16	1	5.061948	0.033144	94.4364
2^12	4	1	1.266794	0.132438	23.6337
2^12	4	1	1.165648	0.14393	21.74668

Comparing to the pure MPI case, we see the speed up is decreasing due to the increasing number of processors and additional communication cost. We can certainly conclude that this is not an efficient way for at least this problem. The caveat here is that we have not used GPU-GPU communication and in future, if that is possible, it could reduce fairly the communication cost.

Lessons from Unified memory: One of the reasons for achieving the high speed up in the single CPU-GPU implementation is through Unified memory. Unified memory is a single memory address space accessible from any processor in the system. By using cudaMallocManaged(), there is a Just-in-time approach to memory management. The data is migrated once a page fault occurs (at device or host side) depending on the data location and then, the data is copied to requesting device. Also, by using asynchronous copy, we can plan to pull the required data before the start of computation, thereby saving time. The NVidia profiler

option also allows to identify the page faults which are useful to assess and optimize the performance of kernels/ data transfers. There are also options for pinned memory at host which allows for localizing a data based on frequency of usage. The option cudaMemPrefetchAsync() helps to prefetch asynchronously and thus overlapping with previous computation. In all, the bottleneck identified in the most optimized (or fastest) run was the cudaDeviceSynchronize() API call which is an inherent requirement for the correctness of the problem and design part of the algorithm. If we remove it, there may be overlaps and the obtained answer may not be correct every time.

Conclusion:

In this work, we explored the possibility of heterogeneous combinations of CPU-GPU architecture for accelerating n-body simulation problem. The lessons learnt on compiling with MPI, setting up CUDA environment etc. are invaluable for building up from here. Some of the cuda constructs employed – cudaDeviceSynchronize, cudaDeviceProperties, cudaMemCpyAsync, cudaMallocManaged helped to understand the functionality of CUDA programs and to work on complex problems in the future. We hope to explore collective communication routines for multiple GPUs to address the drawbacks of multi CPU-GPU architecture which would be essential when single GPU cannot hold the problem or when resources are distributed physically. We found one CPU-GPU with unified memory access model helps to achieve phenomenal performance compared to the sequential version and distributed version.

Reference:

1. https://en.wikipedia.org/wiki/N-body_problem
2. Fundamentals of Accelerated Computing C/C++
3. Class notes: GPU materials from the NVidia Teaching Toolkit

Note:

The problem was inspired from Nvidia challenge in the course Fundamentals of Accelerated Computing C/C++. The test scripts used for checking the accuracy of the answer is from the challenge. The rest of all programs are written from scratch by self.

Metric for measuring speed used in calculations:

Billions of Operations per second = $10^{-9} * \frac{problem\ size^{\,2}}{avg\ time\ taken\ per\ iteration}$