International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 12, Number 1, pages 131–163, January 2022

Performance Models for Heterogeneous Iterative Programs

Aparna Sasidharan
Ansys,Inc
Lebanon,NH, USA

Received: July 25, 2021
Revised: October 22, 2021
Accepted: November 29, 2021
Communicated by Susumu Matsumae

Abstract

This article describes techniques to model the performance of heterogeneous iterative programs, which can execute on multiple device types (CPUs and GPUs). We have classified iterative programs into two categories - static and dynamic, based on their workload distributions. Methods are described to model their performance on multi-device machines using linear regression from statistics. Experiments were designed to capture the behavioral response of different types of iterative programs to variations in regression variables. Experiments were divided into two sets - training and validation. Training sets were used to train and compare performance models, and validation sets were used to determine the accuracy of predictions. Performance models were developed for the execution time and the energy consumption of programs.

Keywords: Hybrid Programming Models, Multi-GPU architectures, Unified Memory, Performance Models, Statistics, Energy Models, Iterative Programs

1 Motivation

Several large applications have benefited from using GPUs to offload their compute-intensive program regions [20]. In this article we attempt to quantify the performance of heterogeneous programs in-terms of their resource usage. Performance models can expose benefits and flaws in programming models by modeling their effectiveness in translating a parallel computation to a program. Algorithms can be analyzed for scalability on different machines by comparing their performance models for each machine. Developing a performance model for computer programs is a non-trivial task considering the numerous hardware resources and programming methodologies available today. A large program is often composed of several libraries and has complex load distributions. Most of the work on performance models for software is focused on applications or particular algorithms [6]. Rather than build a model for an application, we have focused on a programming model and host-device architecture. Another application with similar algorithms, following the same programming model and executing on similar machines will have the same performance model (variables and exponents); with different coefficients. Complexity analysis of parallel algorithms usually cover their dependence on data sizes and number of processes [29]. Theoretical analysis can improve the implementation of an algorithm by providing growth rates for computation and communication costs, including costs for contention and data movement. But the implementations of an algorithm may show variation in performance across different machines which is beyond the scope of theoretical analysis. For example, the constants in the growth rates provided by complexity analysis may differ widely across

machines. Performance models are good tools for analyzing the effects of memory access patterns which are significant for parallel programs with large number of threads. The programming model defines interactions within and across devices and the sequence of operations that lead to a correct program. As far as statistical methods are concerned, we have relied on *regression* [21], [23]. The data collected from program execution on non-shared machines follow a normal distribution, with definite mean and variance. Measurements from computer programs satisfy requirements for statistical models, which makes statistics a useful tool for analyzing them. We have restricted the discussion to iterative programs, where the primary resource is memory.

Section 2 describes the programming model and section 3 the methods used to build performance models. Sections 4, 5 and 6 describe the programs and their performance models. Sections 7, 8 and 9 discuss conclusions, future work and related work.

2 Hybrid Programming Model

There are several programming models in literature that use mixed devices where a few devices are hosts and others are accelerators [3] [27] [31] [8] [38] [7]. Due to their flexibility in handling control flows, host devices are typically CPUs. Historically, hybrid (mixed device) programs have followed offload model [24], where host devices offload kernels to execute on GPUs. Recently, many-core CPUs have been used as offloading devices [24]. Most hybrid programming models are asynchronous, where host devices allocate memory for the data used in a program and schedule them for copying to devices. The kernels are scheduled in streams and there can be multiple streams scheduled to execute on a device. Although streams may execute in any order, the input to streams and output from streams are co-ordinated by hosts so that the total computation maintains program order. Hosts may also perform computations that are applied to the input to or output from accelerators. Therefore the same host can be used to co-ordinate multiple devices, or multiple hosts can be used for the same accelerator device. We have considered a simple subset of these programming models where the role of a host is reduced. The entire memory required for the program resides predominantly on GPUs and hosts access data by mapping pages to their physical memories. We intend to extend this model to accommodate more complex memory allocation schemes. The control flow has been simplified - host processes spawn a thread and initialize a stream for each device connected to them. Threads schedule kernels independently to streams. Scheduling overheads are low for this model since a single stream is assigned to a GPU and streams are handled in parallel on host processes by separate threads. Memory copies between hosts and devices use unified memory paging mechanism, which makes them implicit [30]. With currently available machines and problem sizes, a distributed memory architecture was necessary with groups of GPUs connected through host CPUs. The programming model is illustrated in figure 1 and the host-device architecture is provided in figure 2.

2.1 Memory Model and Device Architecture

A host device and the GPUs connected to it form a fully connected graph, where vertices are devices and edges are connections, shown in figure 2. An edge exists between any two devices A and B if they can access data residing in each other's memories. Depending on the networks used, the latencies of these connections are likely to be different. Edges between peer GPUs are shown in solid lines and edges between a host device and its GPUs are shown in dotted lines. Any two host-GPU groups are connected through their hosts. The GPUs in a group are configured to use RDMA to access the memories of peer GPUs (GPUDirect) [39]. Peer GPU memory accesses may transfer data using GPUDirect if possible or migrate pages to local memory. For a single host-device group, the entire program is a sequence of memory accesses of varying latencies and computations performed on them. From the perspective of the host process that spawned the threads, computations need to follow program order for correctness. Streams may be scheduled in any order on the devices. The host process must use *barriers*, *signal-wait* or *join* constructs to co-ordinate depenedencies between kernels. If the host process accesses data residing on GPUs, it will not have the correct version until the kernels attached to the data have completed execution. A *join* or *barrier* ensures that all

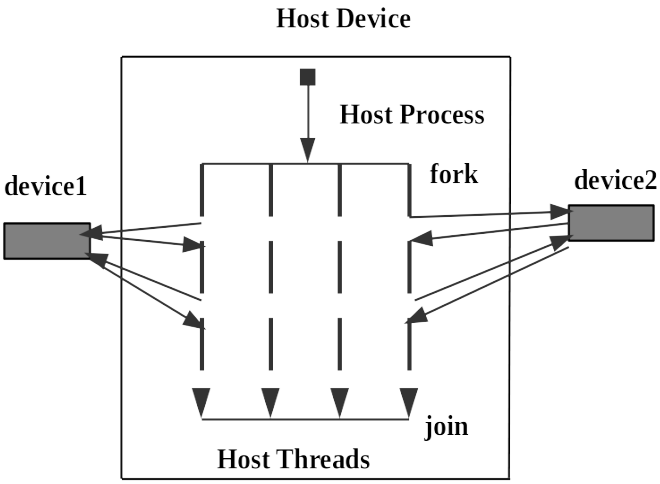

Figure 1: Hybrid Programming Model

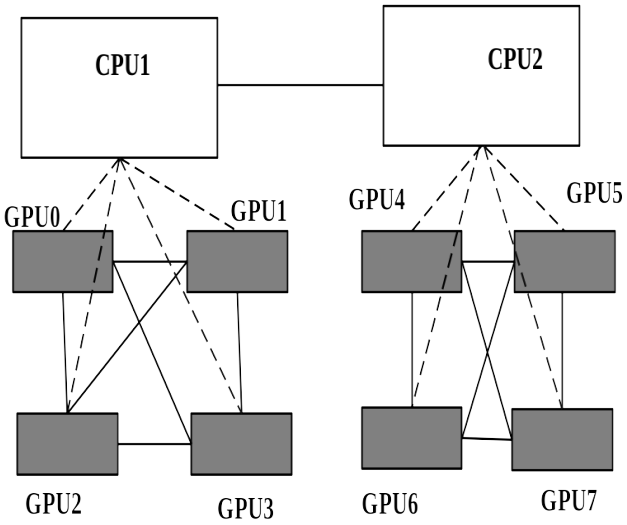

Figure 2: Device Architecture

memory accesses issued during prior *fork-join* sections have flushed values from caches to memory and that the address locations have final values consistent with program order. The coherency of shared pages is handled by the unified mechanism [30] and the operating system. This programming model did not add any non-determinism to its programs. However, a parallel program that performs floating point operations may produce values different from its sequential execution due to lack of associativity in floating point computations and rounding errors [29].

3 Performance Model

We have used regression to build the performance models. Given a response y and n input variables X, where $X^T = \{X_1, X_2, ..., X_n\}$, y can be modeled as a function of X :

$$\hat{y} = f(X) \tag{1}$$

where $\hat{y}$ is the value of the response computed by some regression function f. Linear regression uses a linear approximation for the function f. Replacing $f(X)$ in equation 1 by a linear approximation, we get equation 2, where the R.H.S is a hyperplane in n-dimensional space with coefficients B_i and intercept B_0. The coefficients are the slopes of the hyperplane in each dimension and B_0 is the point where the hyperplane intersects the axes:

$$\hat{y} = B_0 + \sum_{i=1}^{i=n} B_i X_i \tag{2}$$

Equation 2 is a general form that includes higher degree polynomials (degree of > 1). Higher degree terms are included by replacing $X_i = X_i^{a_i}$, where a_i are integers. Most of the models considered in this article have included interactions between variables. Interaction terms are product terms added to equation 2. For example, the product term X_1X_2 models the combined effects of X_1 and X_2. The addition of interaction terms would modify equation 2 to the following :

$$\hat{y} = B_0 + \sum_{i=1}^{i=n} B_i X_i + \sum_{k=1,j=1}^{k=n,j=n} B_{jk} X_k X_j + \sum_{i=1,j=1,k=1}^{i=n,j=n,k=n} B_{ijk} X_i X_j X_k + ... \tag{3}$$

The metrics used to evaluate models are defined using residuals or differences between computed and measured values of the response variable (y), i.e $|\hat{y} - y|$. The most commonly used metric is the Residual Sum of Squares (RSS). The coefficients computed by linear regression are the set of values that minimize this metric for a given training data. Therefore, the coefficients are considered as generated from a multivariate normal distribution with quantities such as mean, variance and error defined on them (column *Stand.Error* in the tables) [21]. Besides linear regression, Generalized Additive Models (GAM) can be used to model the programs discussed in this article. GAMs follow a different approach, where the response is modeled as a sum of functions [23], minimizing for each function seperately. It is useful if dependent variables have high-order exponents ($>= 3$). For most hybrid programs with offloading and typical responses measured from them, high-order terms are not common. Most of the relationships between responses and variables are linear, quadratic or logarithmic, with high chances of interactions.

We used the method of least squares to solve the linear regression. Experiments were divided into two groups - training and validation. The data gathered from training experiments were used to train and validation data were used to verify the models. We used a large number of observations relative to the number of variables to train the models so that predicted values in the population are likely to be close to ideal predictions. Residual Standard Error (RSE), which is a derivative of RSS, was used to quantify the fitness of models. The lower the RSE of a model, the smaller its difference between predictions and measurements of a response variable, leading to a better fit. The significance of a variable (X_i) to a model is reported as the result of a null-hypothesis that a particular coefficient is zero ($t - value$ and $Pr(> |t|)$ in the tables). If a variable has high $Pr(> |t|)$, then its contribution may not be significant to the response and it may be possible to build a

model by dropping it. In such scenarios, we have used tests such as Anova and cross-validation for comparing models. Anova uses a metric called F-statistic (columns F and $Pr(> F)$ in the tables) to quantify the significance of a group of variables. We have reported these metrics wherever Anova was used along with the degrees of freedom for each model (column *Res.df* in the tables) [21]. We used raw data in the models without applying any transformations because the input data were not correlated. Standardization of input data was not required for least squares. A model with the best fit was considered acceptable. This decision was made based on its RSE and predictions for test data (not training data).

4 Heterogenous Iterative Programs

We have only considered iterative programs in this article, i.e programs which perform a repetitive sequence of computations until a criterion is met. The primary resource for these programs is memory since they access data repeatedly with scope for reuse. They are interesting test cases for performance modeling on heterogeneous machines because they capture the effects of memory hierarchies and memory types. For convenience, we have classified heterogeneous iterative programs into two groups based on their *load* and data reuse. We use the term *load* to quantify the work done by a program in any iteration, e.g for matrix multiplication, *load* would mean the sizes of input matrices.

1. Static : Programs belonging to this group have constant loads. Once its data is partitioned, a program can execute to completion by reusing partition information and data. Such programs do not have overheads from load balancing. Most iterative programs that use fixed size matrices (meshes) fall into this category [25].

2. Dynamic : Programs in this group have workloads that evolve during their execution. They may need load balancing for scalable performance. Examples include n-body simulations [5] and mesh refinement [15].

Both categories of programs can be analyzed using the same techniques. But the experiment design needed for building performance models are different for each. The performance of static programs can be analyzed by measuring their load and execution time per iteration. For dynamic programs, there can be a range of values for load in a single execution of the program. The load and execution time per iteration take the form of distributions with mean and variance. Load balancing will add overheads to the execution time per iteration whenever it is invoked.

5 Static Iterative Programs

The program considered in this section is an implementation of Lagrange force computation on a structured mesh based on a DOE benchmark [25].

The base program was re-written entirely in the programming model described here. It takes mesh dimensions as input and generates a structured mesh of hexahedrons in 3 dimensions. Each hexahedron (element) has 8 corner nodes, with nodes shared between neighboring elements. Each element maintains the ids of nodes that constitute it. Two primary data structures were used by the kernels : element and node arrays. Elements and nodes were assigned unique global ids. These data structures were partitioned equally across GPUs with a single partition assigned to a GPU. Elements and nodes had different decompositions across GPUs, which created peer memory accesses in the kernels. The program is iterative and runs for a fixed number of iterations. Kernels were written entirely using Nvidia CUDA10.0 and driver functions used by hosts were implemented in c++ [42]. The independent threads on the hosts were spawned using pthreads [1]. Let n be the number of mesh elements per GPU and T the number of threads per GPU. All algorithms implemented as CUDA kernels in this program had computational complexity $O(\frac{n}{T})$, linear in the number of mesh elements n. We implemented two versions of this program :

Algorithm 1 Sedov Blast Simulation

```
1: procedure SEDOV(K)
2:     n ← getNumGPUs
3:     for do i ← 1, K
4:         SPAWN(n, InitializeNodesKernel)
5:         JOIN(n)
6:         SPAWN(n, ComputeVolumeForcesKernel)
7:         JOIN(n)
8:         SPAWN(n, UpdateNodeQuantitiesKernel)
9:         JOIN(n)
10:        SPAWN(n, UpdateElementPropertiesKernel)
11:        JOIN(n)
12:        SPAWN(n, CalculateConstraintsKernel)
13:        JOIN(n)
14:    end for
15: end procedure
```

1. Shared Memory : single host-device group
2. Distributed Memory : multiple host-device groups

The pseudo-code for the shared memory version is provided in algorithm 1. The *spawn* function in the pseudo-code spawns pthreads on the host process, one for each GPU. Each *spawn* function has a corresponding *join* function where host threads synchronize. The pseudo-code for the distributed memory version of the program is provided in algorithm 2. It uses the same CUDA kernels with minor changes for identifying local and remote nodes. Exchange functions are invoked on host processes in every iteration to send/recv remote node data. Both programs have used the unified memory [30] implementation provided by CUDA. A single address space is used for a host and devices connected to it. Memory allocation follows first-touch policy and page faults and remote atomics may lead to page migration between devices. The distributed memory version was implemented by spawning an MPI process on each host and pthreads within each MPI process. Packing and unpacking of MPI communication buffers on host processes access data residing on device memories. This could lead to page migration from GPU memories to host memories and vice versa. The program synchronized at the end of every iteration where host processes computed global reductions of constraints [25].

5.1 Shared Memory

This section describes methods used to build the performance model for a shared memory implementation which runs on a single host-device group. The response variable modeled is execution time/iteration. The variables considered are the following :

1. Total Memory Usage ($X1$)
2. Number of GPU Threads ($X2$)
3. Number of Peer GPU Memory Accesses ($X3$)

The variable $X3$ was measured using an auxiliary program that computed the edge-cut of partitions. We used the maximum edge-cut as $X3$ since GPUs execute independently and synchronize at *join* primitives. $X1$ refers to the total persistent memory allocated across all GPUs in a group. Thread scheduling policy has been previously found to be an important factor in determining the execution time of kernels on GPUs [16]. We have ignored other possible variables such as cache sizes, page sizes and page table sizes. These factors get covered by the variable for memory usage and its interactions. R [34] was the software used to build the statistical model. The errors for each coefficient as well as the RSE for the model [21] are reported.

Algorithm 2 Distributed Sedov Blast Simulation

```
1: procedure SEDOV(K)
2:     n ← getNumGPUs
3:     for  doi ← 1, K
4:         spawn(n, InitializeNodesKernel)
5:         join(n)
6:         spawn(n, ComputeVolumeForcesKernel)
7:         join(n)
8:         spawn(n, UpdateNodeQuantitiesKernel)
9:         join(n)
10:        ExchangeNodePositions()
11:        ExchangeNodeVelocities()
12:        ExchangeNodeAccelerations()
13:        spawn(n, UpdateElementPropertiesKernel)
14:        join(n)
15:        spawn(n, CalculateConstraintsKernel)
16:        join(n)
17:        ReduceConstraints()
18:    end for
19: end procedure
```

5.1.1 Experiments

All evaluations were performed on Bridges GPU nodes provided by XSEDE [44], which consist of two types of compute nodes : two CPUs (Intel Broadwell E5-2683 v3) and two Tesla P100 GPUs [46] or two CPUs (Intel Harwell E5-2695 v3) and four Tesla K80 GPUs [45]. We used both types of compute nodes in our experiments. All experiments were performed for a fixed number of iterations (> 100) and the execution time per iteration was measured. Each experiment was repeated 5 times and all repetitions were included in the training set. A total of 275 measurements were used for training the model, with total memory usage [1-20GB], number of threads [512-3584] (Tesla P100) and [512-4992] (Tesla K80). Mesh dimensions were varied from $160X160X160$ (4 million elements and nodes) to $640X320X320$ (65 million elements and nodes) on 2 GPUs (Tesla P100)/4 GPUs (Tesla K80). The GPUs and CPU in a group were connected using high-bandwidth NVlink1 [44]. Memory usage is reported in gigabytes(GB) and execution time/iteration is reported in milliseconds(ms).

5.1.2 Variable Selection

We built the following three models and chose the most suitable one.

1. Model A : $X1$,$X2$ and $X3$ without any interaction terms.
2. Model B : $X1$,$X2$ and the interaction between $X1$ and $X2$
3. Model C : $X1$,$X2$,$X3$ and all interaction terms.

We used a quadratic dependence for total memory usage based on the scatter plot in figure 3 and linear dependence for the other variables for all models. Model A had an RSE of 181.4 and p-value of $< 2.2e-16$ for training data. It was chosen after considering other single variable models and comparing them using Anova [23];it confirmed the significance of all three variables in the final model. Model B dropped $X3$ and included $X1X2$ and $X1^2X2$. The RSE for training data for Model B was 138.3 with p-value $< 2.2e-16$ which showed the significance of the interaction terms. It modeled the effects of threads accessing caches and memory in different ways and the contention between them for resources. Model C included all interaction terms. This lowered the RSE for training data to 136.5 with p-value $2.2e-16$. The coefficients of Model C for Tesla P100 are provided in table 1.

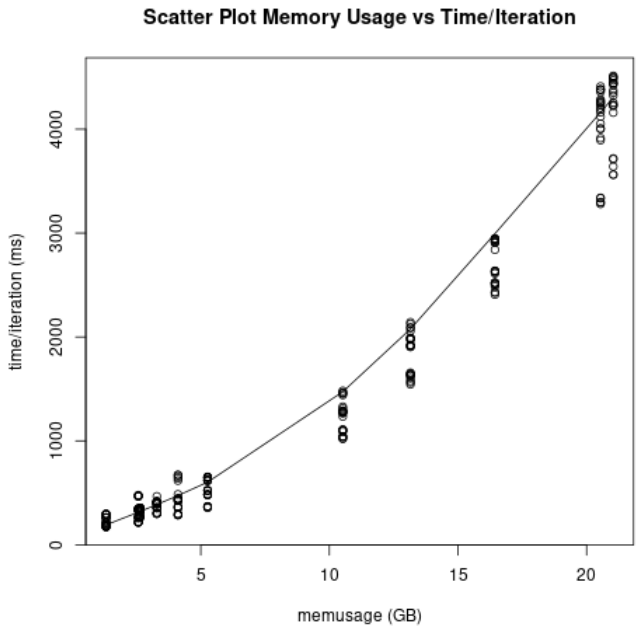

Figure 3: Memory Usage vs Time/iteration

	Estimate	**Stand.Error**	**t-value**	**Pr(>|t|)**
$Intercept$	1.402e+03	1.374e+02	10.206	$< 2e-16$
$X1$	1.636e+04	2.215e+03	7.386	$2.00e-12$
$X1^2$	-7.938e+02	1.374e+03	-0.578	0.563863
$X2$	3.155e-03	5.745e-02	0.055	0.956247
$X3$	-7.471e-04	7.006e-04	-1.066	0.287265
$X1X2$	3.297e+00	9.264e-01	3.559	0.000442
$X1^2X2$	2.643e+00	5.745e-01	4.601	6.54e-06
$X1X3$	2.056e-02	8.807e-03	2.334	0.020335
$X1^2X3$	7.894e-03	4.085e-03	1.932	0.054410
$X2X3$	5.460e-07	2.930e-07	1.863	0.063533
$X1X2X3$	-1.063e-05	3.683e-06	-2.887	0.004207
$X1^2X2X3$	-3.345e-06	1.709e-06	-1.958	0.051284

Table 1: Coefficients for Tesla P100 for Shared Memory Program

From the table, the terms $X1^2$ and $X2$ may not be significant in isolation, but significant in combination with each other and $X3$ ($Pr(>|t|)$ values). The terms $X1X2$ and $X1^2X2$ include overheads such as page faults, cache misses and limits in device memory bandwidth caused by threads sharing resources while satisfying their memory requests. The terms containing $X3$ model the cost of accessing peer device memories through NVLINK (GPUDirect), including overheads such as page faults, page migrations (if any), cache misses and resource contention. For example, $X2X3$ covers the cost of $X2$ threads issuing a maximum of $X3$ peer memory requests. Since local and peer memory accesses belong to different pages, it could lead to increased page faults and cache misses, depending on the total memory usage (number of pages) of the program. We have modeled these contributions to the response by including interactions between number of threads and number of peer memory accesses ($X2X3$), total memory usage and number of peer memory accesses ($X1X3$ and $X1^2X3$) and all three factors ($X1X2X3$ and $X1^2X2X3$). The results from Anova are tabulated in table 2. When comparing two models, F-statistic reports the result of the null-hypothesis that the

Model	**Res.Df**	**RSS**	**F**	**Pr(> F)**
A	270	8881181		
B	269	5145468	200.5555	$< 2e-16$
C	263	4898857	2.2066	0.0428

Table 2: Anova Table for Shared Memory Models

new variables in a model are insignificant. If the probability of this null-hypothesis being true is low, then the new variables are considered significant. The columns F and $Pr(>F)$ in table 2 report the results of comparing two consecutive models. Between models A and B, B is clearly better than A, because its p-value is $2e-16$. Between models B and C, it is not clear from Anova whether C has an advantage over B, although its p-value is low. We validated our findings from Anova using cross-validation. The set of 275 observations were divided into 232 training data and 43 validation data. The models were built using training data and RSE was measured for the validation set. The model with the lowest error in the validation set was Model C. Once Model C was selected, we built the full model using the entire observation set. The graph in figure 4 shows predicted fit values and training data as a function of $X1$ for Model C (Tesla P100). The coefficients for Model C for Tesla K80 are tabulated in table 3 for the same training data.

	Estimate	**Stand.Error**	**t-value**	**Pr(>|t|)**
Intercept	1.987e+03	5.144e+02	3.864	0.000141
$X1$	1.410e+04	8.274e+03	1.704	0.089577
$X1^2$	-9.797e+03	5.136e+03	-1.908	0.057549
$X2$	3.479e-01	1.804e-01	1.929	0.054843
$X3$	1.416e-03	1.753e-03	0.808	0.420098
$X1X2$	1.454e+01	2.900e+00	5.013	9.87e-07
$X1^2X2$	6.652e+00	1.798e+00	3.699	0.000263
$X1X3$	6.421e-02	2.203e-02	2.914	0.003873
$X1^2X3$	2.349e-02	1.027e-02	2.287	0.023020
$X2X3$	-4.353e-08	6.147e-07	-0.071	0.943597
$X1X2X3$	-2.152e-05	7.719e-06	-2.788	0.005700
$X1^2X2X3$	-1.817e-06	3.589e-06	-0.506	0.613219

Table 3: Coefficients for Tesla K80 for Shared Memory Program

Tables 1 and 3 cannot be used to deduce hardware differences between Tesla K80 and Tesla P100 since the number of devices are not the same for the experiments. Instead, the two device groups (4 Tesla K80 and 2 Tesla P100) [45], [46] can be compared by comparing their models. The

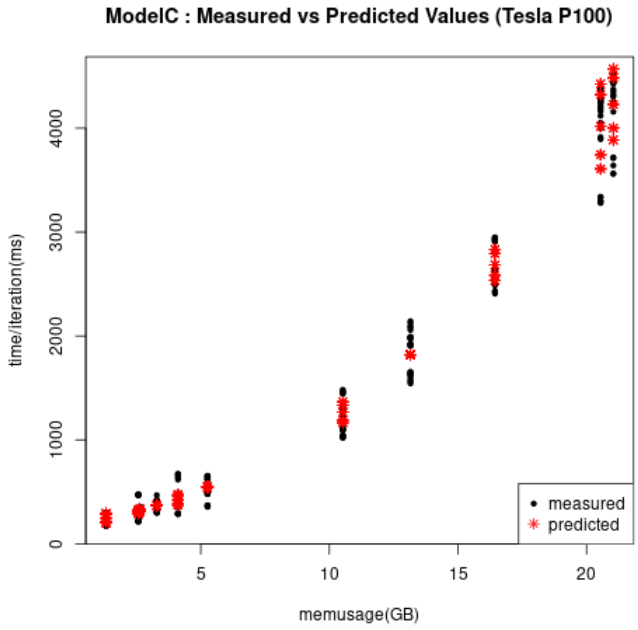

Figure 4: Model C:Predicted and Measured Responses (Tesla P100)

coefficients and their p-values for Model C are quite different for these device groups. For example, the coefficients of the interaction terms $X1X2$, $X1^2X2$, $X1X3$ and $X1^2X3$ are higher for K80 compared to P100, which shows higher costs for these components for this device group. From experience, we found the large test cases (high memory usage) to be considerably slower on K80, despite using 4 K80 devices compared to 2 P100 devices. Therefore, P100 may be a better choice of device for large test cases which access several giga bytes of memory (local and remote).

5.1.3 Testing the Model

Model C described in the previous section was put to test by comparing its predicted values against measured test data from 15 experiments. All tests were performed on Tesla P100. The graph in figure 5 shows the comparison. The predicted values used in the graph are the *fit values*. The model predicts values within a confidence interval of 95 percent (lower,upper). Measured values and those predicted by the model are tabulated in table 4. From the observations in table 4 it can be seen that Model C generates good predictions of time/iteration for the shared memory program. The scaled residual error for test data was 136.4802 which was close to the RSE for training data (136.5).

To validate the model built using linear regression, it was compared against a similar model built using GAM. The models predicted close values for test data. It validated the strength of the linear regression model and also helped to understand relationships between variables. What we also encountered is a shortcoming of linear regression - few test cases can cause large errors. Model A (without any interactions) produced good predictions for most inputs. Its predictions were not good enough for test cases with high/low memory usage. Interactions improved predictions for the entire input range, which lead to overall reduction in the errors for these models (B and C).

5.2 Distributed Memory

Unlike the shared memory experiments, this performance model was built by measuring three responses from every experiment - computation time, communication time and total time. Suppose there are N host-GPU groups, with D GPUs per group. Let T_{iter} be the total number of iterations,

time/iteration(ms)	fit	lower	upper
99.35773	101.9065	-23.55771	227.3708
162.59791	135.0566	32.15234	237.9609
221.56805	145.5342	63.80132	227.2671
1118.74100	1082.8179	1003.0477	1162.5882
1618.76647	1524.496	1470.17118	1578.8209
99.89656	136.9941	35.58081	238.4074
130.49190	163.3681	80.19015	246.5461
185.71135	177.3275	111.26249	243.3926
865.64931	1038.4937	974.0151	1102.9724
1375.52520	1507.2771	1463.36609	1551.1881
153.12473	207.1693	131.07085	283.2677
200.82306	219.9911	157.57611	282.4062
317.62413	240.9142	191.34036	290.488
882.75143	949.8453	901.46189	998.2288
1623.64608	1472.8393	1439.88933	1505.7892

Table 4: Predictions for Shared Memory Program

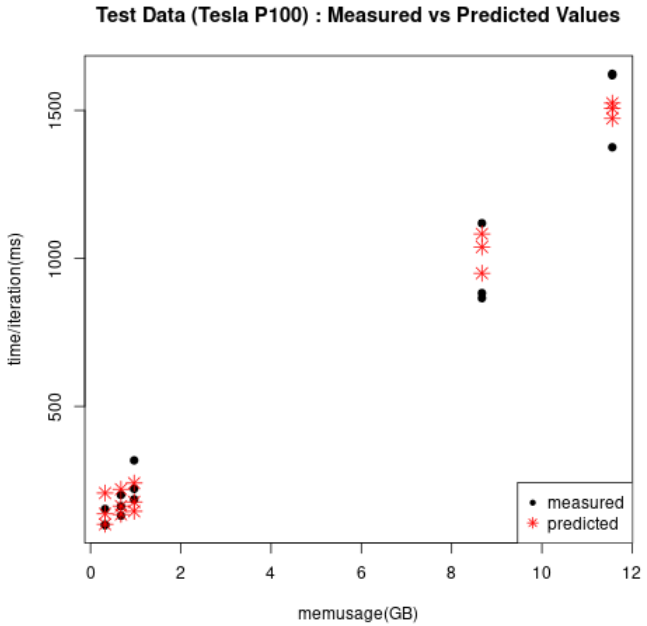

Figure 5: Model Test: Predicted and Measured Responses (Tesla P100)

$T_{comp_{ijd}}$ be the computation time measured in the j^{th} iteration on the d^{th} GPU belonging to the i^{th} host. Then the total computation time T_{comp} is :

$$T_{comp} = \max_{i=1,N} \sum_{j=1,T_{iter}} \max_{d=1,D} T_{comp_{ijd}} \tag{4}$$

Communication time is the time taken for the following operations :

1. Copy required data from GPUs to communication buffers allocated on CPUs.
2. Post non-blocking MPI send/receive messages and wait for completion of data exchange.
3. Update GPU data structures with received data.

Let $T_{comm_{ij}}$ be the communication time measured in the j^{th} iteration on the i^{th} host process. Then, the total communication time T_{comm} is :

$$T_{comm} = \max_{i=1,N} \sum_{j=1,T_{iter}} T_{comm_{ij}} \tag{5}$$

The third response was the total time, which was measured on CPUs as the total time taken to execute all iterations. Let T_i be the total time measured on the i^{th} host. The total execution time T_{tot} of the program is :

$$T_{tot} = \max_{i=1,N} T_i \tag{6}$$

We considered the three responses as separate and used them to build three models - computation, communication and program model. We used Model C from the previous section for the computation model. To better understand factors affecting host-host communication and host-GPU page transfers, communication time/iteration was used as a separate response. We built a simple communication model since we used a small cluster for our experiments. Factors such as network topology and congestion have not been considered [29]. We built a separate program model using the third response (total time/iteration). Since least squares tries to find the hyperplane that best fits the data, the values of coefficients can be quite different when separate models are added vs constructing a full model with all variables and their interactions.

5.2.1 Experiments

All experiments in this section were performed on bridges GPU cluster provided by XSEDE [44]. An MPI process was placed on each host CPU. A host-GPU group consisted of a CPU (Intel Broadwell E5-2683 v3) and two Tesla P100 GPUs. NVLink1 was used for communication within a group and PCIe across groups. Maximum of 4 such groups were used for these experiments. 500 observations were used to build the performance model. All values for memory usage are reported in gigabytes(GB) and time/iteration is reported in milliseconds(ms). The total number of GPUs were varied from 2 to 8. The number of threads per GPU was varied in the range [512-2048]. Mesh sizes were varied to cover a range of memory sizes [1-20GB] per group of GPUs. The largest mesh size was 131 million elements and nodes, which required a total memory of 40GB (allocated on GPUs).

5.2.2 Variable Selection

The primary variable considered for the communication model is communication buffer size. This variable factors in the network bandwidth and the time taken to transfer a certain number of bytes between two host processes. Besides buffer sizes, we have also included the total memory usage on GPUs to cover the costs of filling buffers and updating GPU data structures. Since GPUs execute asynchronously, the total number of GPUs used in the experiments was added as a variable. This would include costs such as synchronization between hosts and devices and the cost of initiating

MPI messages. Therefore, communication follows a logP [29] model with number of GPUs (message setup) and maximum number of inter-process edges (bandwidth) as parameters. We used maximum instead of sum of inter-process edges in this model, assuming messages are sent or received in parallel and the time taken for completion of data exchange depends on the largest message. The coefficients for this model are provided in the table 5. $X1$ is the memory usage per host-GPU group, $X4$ is the maximum number of inter-process edges in any group and $X5$ is the number of GPUs.

	Estimate	**Stand.Error**	**t-value**	**Pr(>|t|)**
Intercept	-2.082e+01	4.112e+00	-5.064	$5.79e-07$
$X1$	-2.569e+02	6.414e+01	-4.005	$7.14e-05$
$X1^2$	-4.261e+01	3.358e+01	-1.269	0.205
$X4$	1.702e-04	1.074e-05	15.847	$< 2e-16$
$X5$	7.918e+00	1.062e+00	7.453	4.03e-13

Table 5: Coefficients for Communication Model

The graph in figure 6 shows training data and predicted fit values as a function of communication buffer sizes ($X4$). The number of inter-process edges was computed using the auxiliary program. We chose the model in table 5 after considering models with $X4$ only and $X4$ and $X5$. The models were compared using Anova and cross-validation. Anova test showed $X4$ and $X5$ as significant variables, but it was not clear whether to include $X1$. The model with 3-variables had the lowest RSE during cross-validation tests. We divided the set of 500 observations into 400 training data and 100 validation data for cross-validation.

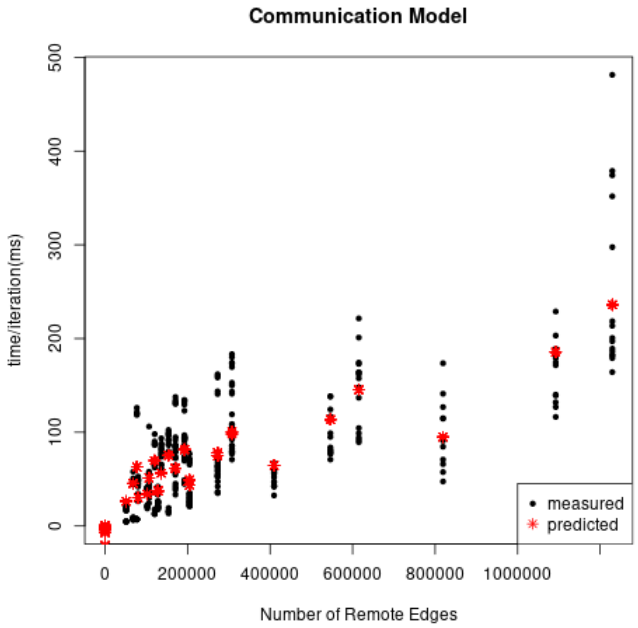

Figure 6: Communication Time/Iteration(ms) vs Maximum Message Size

The graph in figure 7 shows training data and predicted fit values from Model C (section 5.1) as function of $X1$. The variables it considered are total memory usage per group, number of GPU threads and maximum number of peer memory edges across all node partitions. Since partitions are load balanced, the slowest GPU is likely to have higher peer GPU memory accesses.

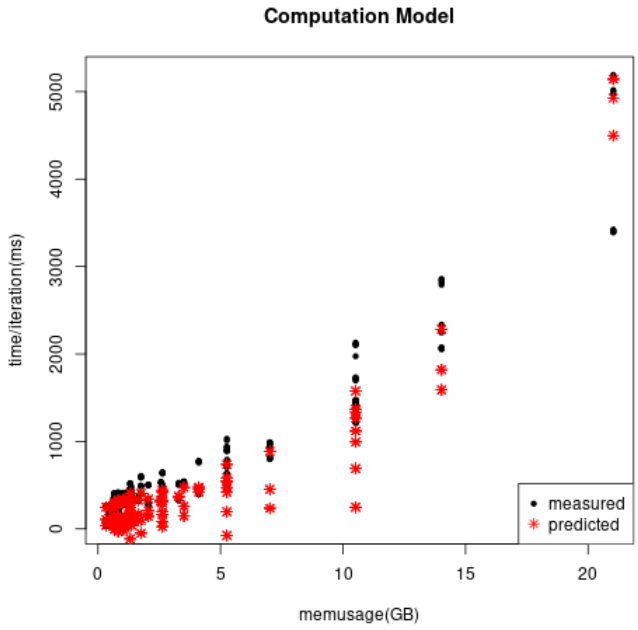

Figure 7: Computation Time/Iteration(ms) vs Maximum GPU memory usage per host-GPU group(GB)

The full model considered five variables. The end-end data flow in the program is covered by these variables. Consider a critical path in the program where a GPU in group i requires data from a GPU in group j. The host process in group j copies data from its GPUs to communication buffers by mapping the required pages to its physical memory and posts MPI messages to communicate them to remote hosts. When messages are received, the host process in group i updates its GPU data structures with received data. The variables that directly affect this critical path are maximum number of inter-process edges, number of GPUs and memory usage on GPUs. Other variables affect this data flow indirectly. The availability of a GPU to the host depends on when it completes its computation kernels, which in turn depends on the number of GPU threads and maximum number of peer memory edges. The variables are enumerated below:

1. Memory usage on GPUs : maximum memory usage per host-GPU group $(X1)$.
2. Number of GPU threads : $(X2)$.
3. Peer memory edges : maximum peer GPU memory accesses within any host-GPU group (RDMA) $(X3)$.
4. Inter-process memory edges : maximum host-host edges across all inter group edge-cuts (network) $(X4)$.
5. Total number of GPUs : $(X5)$.

The coefficients for the full model are provided in the table 6. The graph in figure 8 shows measured total time/iteration and predicted fit values from the full model for training data. The full model was chosen after considering models without $X4$ and/or $X5$ (ignoring inter-group communication). We compared these models with the full model using Anova and cross-validation tests. Since the communication cost was a small fraction of the total cost, Anova tests did not suggest $X4$ or $X5$ as significant variables. However, during cross-validation, a model with $X4$ and $X5$ had the

	Estimate	**Stand.Error**	**t-value**	**Pr(> |t|)**
Intercept	8.371e+02	6.392e+01	13.096	$< 2e-16$
$X1$	2.851e+04	5.279e+03	5.401	1.03e-07
$X1^2$	1.077e+04	2.590e+03	4.158	3.80e-05
$X2$	-1.376e-01	2.005e-02	-6.865	2.01e-11
$X3$	4.253e-04	4.472e-04	0.951	0.342100
$X4$	3.697e-04	5.183e-04	0.713	0.475939
$X5$	-1.299e+01	9.075e+00	-1.432	0.152879
$X1X2$	9.392e-02	9.739e-01	0.096	0.923214
$X1^2X2$	-7.273e-02	8.172e-01	-0.089	0.929113
$X1X3$	-6.958e-02	2.197e-02	-3.167	0.001635
$X1^2X3$	-7.574e-03	2.612e-03	-2.899	0.003911
$X2X3$	2.941e-07	7.284e-08	4.037	6.28e-05
$X1X4$	6.403e-02	2.254e-02	2.841	0.004688
$X1^2X4$	1.194e-02	3.566e-03	3.349	0.000874
$X1X2X3$	1.665e-06	1.121e-06	1.485	0.138153
$X1^2X2X3$	2.183e-06	7.332e-07	2.977	0.003050

Table 6: Coefficients for Full Model

lowest RSE. We retained both variables in the full model and included interaction terms between $X1$ and $X4$. These terms include cache misses, page faults and page migrations between CPU and GPU. We intend to improve this model by training on a larger cluster with a higher range for communication data.

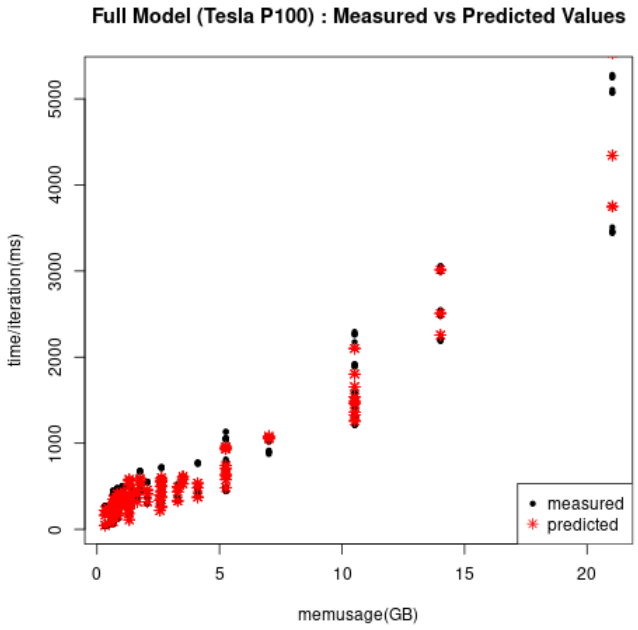

Figure 8: Time/Iteration (ms) vs Memory Usage per group (GB) for Full Model

5.2.3 Validating the Full Model

The full model with 5 variables was validated using test data. The graph in figure 9 and the observations in table 7 compare measured execution time per iteration for test data and their fit values. The fit values were predicted within a confidence interval of 95 percent. The RSE for test data was 116.5921 which was close to the RSE for training data (116.6).

time/iteration(ms)	fit	lower	upper
46.057262	173.3884	124.82432	221.9524
286.45206	373.596	351.65627	395.5357
620.197656	837.9877	796.66996	879.3053
1680.902432	1696.7846	1482.69508	1910.8741
83.24181	225.3706	178.24071	272.5004
314.151184	434.7377	410.98099	458.4943
243.856856	311.1511	276.63363	345.6687
442.949606	484.1703	467.44523	500.8953
51.570538	138.686	100.76018	176.6119
59.516424	198.9536	158.64701	239.2601
369.134176	396.0422	377.01229	415.0721
50.438426	117.5954	84.08812	151.1026
367.567462	462.943	445.59438	480.2915
487.82655	765.6987	726.07988	805.3175
401.660898	320.0431	288.86461	351.2217
603.316294	614.5687	586.39621	642.7411
742.213366	798.5749	714.69392	882.4558
760.788466	1035.4734	984.11449	1086.8323
696.21977	605.0783	561.3897	648.7669

Table 7: Predictions from Full Model

We compared the full model built using linear regression against a similar model built with GAMs. The two models predicted close values for the training and test data considered here. Another option was to use multivariate regression [21] to model the computation and communication times, since both responses were measured from the same experiments. But we did not follow that option because the dependent variables for computation and communication are different.

6 Dynamic Iterative Programs

Algorithm 3 Monte Carlo Simulation

```
1: procedure MonteCarlo(K,m,f)
2:     n ← getNumGPUs
3:     spawn(n, MCSimulation, K, m, f)
4:     join(n)
5: end procedure
```

As a representative program in this category, we used a Monte Carlo simulation that tracks particles dispersed in a mesh. The mesh is static and partitioned during initialization. The number and location of particles is dynamic and changes as the simulation evolves. We implemented a CUDA11.0/C++ version of this program based on Quicksilver, which is a DOE benchmark [36]. We used a structured mesh with hexahedrons, where each element had 6 faces, 24 facets and 14 nodes. The mesh was partitioned into d equal sized blocks, where d was the number of GPUs. Each

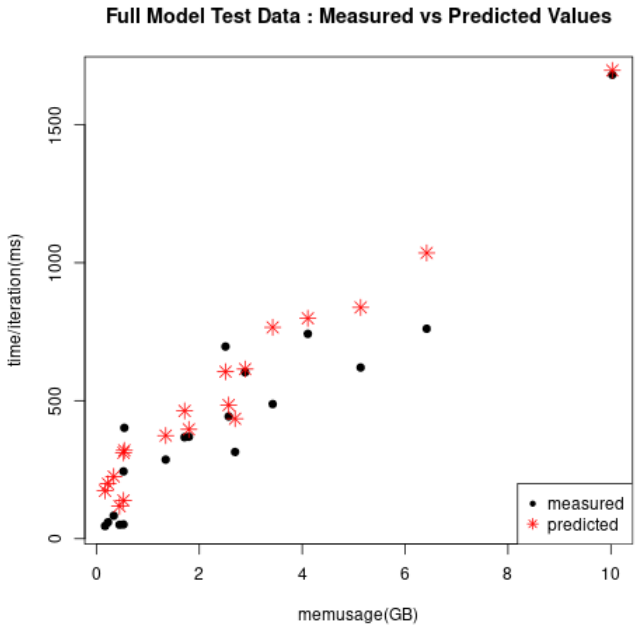

Figure 9: Time/Iteration (ms) vs Memory Usage per group (GB) for Test Data

Algorithm 4 MC Simulation Kernels

```
1: procedure MCSimulation(K,m,f)
2:     n ← getNumGPUs
3:     for do i ← 1, K
4:         for do j ← 1, m
5:             ParticleTrackingKernel
6:         end for
7:         Barrier(n)
8:         if then i%f == 0
9:             PopulationControl
10:            Barrier(n)
11:        end if
12:    end for
13: end procedure
```

GPU generated its partition of the mesh and the particles distributed in it. The particles were initialized with random co-ordinates and velocity values and assigned to mesh elements containing them. A single iteration of the program performed m tracking iterations. The number of particles was modified by adding or deleting particles based on a desired particle population (population control) every f iterations. The pseudo-codes in algorithms 3, 4 and 5 describe the simulation for K iterations. This article discusses a shared memory version of this program. This program was used to create performance models for execution time, power and energy consumption. Let T_{ijk} be the time taken to track a maximum of n_{ik} particles during the jth tracking iteration of iteration i on GPU k ($1 <= k <= d$). Let L_{ik} be the time taken by the ith population control iteration on GPU k. The total execution time T_{tot} of the program is given by equation 7 :

$$T_{tot} = \sum_{i=1}^{i=K} \max_{k=1}^{k=d} (\sum_{j=1}^{j=m} T_{ijk}) + \sum_{i=1}^{i=\frac{K}{f}} \max_{k=1}^{k=d} L_{ik} \tag{7}$$

Algorithm 5 Population Control

```
1:  procedure PopulationControl
2:      n ← getNumGPUs
3:      MigrateParticlesKernel
4:      Barrier(n)
5:      m ← getNumParticles
6:      r ← getRandomNumber
7:      if then r ≤ m
8:          DeleteParticlesKernel(r, m)
9:      else
10:         AddParticlesKernel(r, m)
11:     end if
12: end procedure
```

6.1 Algorithm Analysis

This section describes the algorithms implemented in the program. Let M be the number of mesh elements and N the number of particles at any instant on any GPU. Let T be the number of threads used by a GPU. Particles are assigned to threads in a load balanced manner, with load imbalance of at most one entity. The computational complexity of the particle tracking algorithm is $O(\frac{N}{T})$, linear in the number of particles. The computational complexity of the algorithms used for population control (addition and deletion) is also $O(\frac{N}{T})$.

The computation that determines particle-element intersections depends on mesh size and number of particles. We implemented an algorithm that uses kd-trees to compute intersections between two entity types distributed in the same 3-dimensional space. Empty kd-trees were built recursively on the CPU for a fixed number of leaf nodes. Tree nodes were split based on the geometry of the domain and neither the mesh nor the particle data structures were accessed during their construction. Empty leaves were copied to GPUs, which determined membership of particles and mesh elements in tree leaves and computed intersections. We used kd-trees with leaves of granularity D. A particle kd-tree built using the geometry of the domain and leaf size D may not be balanced [9]. The granularities of leaves would take a range of values $[0 - D]$. The lower bounds discussed here are for uniform distributions of particles and mesh elements. For other distributions, the computational complexity will be a constant factor of these lower bounds, since the algorithms depend on the number of leaves and not the intermediate nodes.

The mesh kd-tree has $\lceil \frac{M}{D} \rceil$ leaves and the particle kd-tree has at least $\lceil \frac{N}{D} \rceil$ leaves. The algorithms which determine membership of entities in leaves have to search for enclosing leaf nodes. The complexity of these algorithms depends on the search method. The computational complexity is at

least $O(\frac{M}{T} * \frac{M}{D})$ and $O(\frac{N}{T} * \frac{N}{D})$ if the algorithm uses linear search. The complexity can be improved to $O(\frac{M}{T} * log(\frac{M}{D}))$ and $O(\frac{N}{T} * log(\frac{N}{D}))$ if binary search is used after sorting the leaves.

The algorithm for computing intersections between particles and mesh elements depends on M and N. The $\lceil \frac{N}{D} \rceil$ leaves were assigned to threads in a load balanced manner. Each thread first determined intersections at the granularity of tree leaves. Intersections between leaves were determined using their bounding box extents without accessing the particle or mesh data structures. Intersections at the granularity of leaves has computational complexity of at least $O(\frac{N}{D*T} * log(\frac{M}{D}))$ if binary search is used. The computation that determines intersections between entities depends on the number of intersecting mesh leaves per particle leaf and D. The maximum number of intersecting mesh leaves per particle leaf can be bounded by the ratio of the lengths of the coarsest particle leaf and the finest mesh leaf bounding boxes. This ratio is determined by the depths of the two trees and can be approximated by $\lceil \frac{N}{M} \rceil$ for uniform distributions. The computation of intersections between entities has complexity $O(\frac{N}{D*T} * \frac{N}{M} * D^2)$. The total computational complexity of the intersection algorithm is at least $O(\frac{N}{D*T} * log(\frac{M}{D})) + O(\frac{N}{D*T} * \frac{N}{M} * D^2)$. Computing intersections at two different granularities and using GPUs for all computations reduced the computational complexity of an algorithm that would have taken $O(M * N)$ work in the worst case. We haven't formally included the computational complexity of the algorithms in any of the models discussed in this article. The quadratic term for memory accesses in the models works only if the algorithms have linear complexity.

6.2 Experiment Design

The simulation was initialized by creating particles at random locations. During population control the GPUs generated different random values for particle counts independently. The execution times of the particle algorithms described in the previous section follow normal distributions. For each experiment, it can be shown that the execution time per iteration will not differ greatly from the execution time per iteration for the mean particle count for that experiment [35]. Therefore, it was sufficient to measure the sample means of particle count and execution time per iteration per experiment. All experiments were performed on a single compute node with 8 NVIDIA Volta V100 GPUs [47] and one Intel Xeon Gold 6248 CPU, provided by XSEDE [44]. The GPUs were connected using NVLINK. Mesh sizes were varied in the range [1000000-64000000] elements and the total number of particles were varied in the range [100000-51200000]. The program was executed for a maximum of 200 iterations. Each iteration (time/iteration in the tables) consisted of 100 tracking iterations. Population control was performed every 20 iterations. The number of GPUs was varied in the range [1-8] and each experiment was repeated 3 times. The number of kd-tree leaves was varied in the range [512-16384], which kept the value of d quite low (maximum 3900). A total of 450 experiments were used in the training set and 40 experiments were used in validation set. All values for memory usage are reported in megabytes (MB) and values for mean execution time per iteration are reported in seconds (s).

6.3 Variables

In our experiments, the fraction of time spent in particle tracking was almost always $> 50\%$ of the total time. We built a performance model for the particle tracking kernel and skipped population control. Population control is likely to become significant in distributed memory versions of the program. Both particles and mesh elements were stored in arrays. We chose the following variables to model the execution time of this kernel :

1. Mesh Memory Usage (X1)
2. Particle Memory Usage (X2)
3. Number of threads (X3)

The response variable used was the mean time per tracking iteration. The total memory allocated in any iteration is the sum of the memory allocated for the mesh and the particles in that iteration.

Since some particles escape the mesh partition, the active particle memory is less than that allocated. Particle tracking performed the following memory operations :

1. Particle Memory : read and write particle co-ordinate values.
2. Mesh Memory : read the mesh element to which the particle belongs, along with its facets and nodes.

Particle memory allocation also follows a normal distribution with definite mean and variance. For each experiment, it was sufficient to measure the sample mean of this distribution as values of $X2$.

6.4 Models

Performance models for execution time, maximum power and energy consumption are discussed in this section.

6.4.1 Execution Time

We first built a model (Model 1) for execution time by dropping $X1$. The coefficients for this model are tabulated in table 8 and measured and predicted values are plotted in figure 10. The RSE for this model was 0.035 and p-value was $< 2.2e-16$ for training data.

	Estimate	**Stand.Error**	**t-value**	**Pr(>|t|)**
Intercept	2.503e-01	3.749e-03	66.764	2e-16
$X2$	5.165e+00	1.628e-01	31.728	$< 2e-16$
$X2^2$	-5.143e-01	1.946e-01	-2.643	0.00851
$X3$	-9.757e-05	2.634e-06	-37.050	$< 2e-16$
$X2X3$	-1.912e-03	8.744e-05	-21.863	$< 2e-16$
$X2^2X3$	2.941e-04	9.985e-05	2.946	0.00339

Table 8: Coefficients for 2-variable Performance Model

Model	**Res.Df**	**RSS**	**F**	**Pr(> F)**
1	456	0.56192		
2	444	0.53633	1.7655	0.05145

Table 9: Anova Table for Tracking Time Models

Although the tracking kernel does not directly depend on the mesh data structure, it affects cache usage in the kernel. The cache misses caused in $X2$ accesses by $X1$ will be captured by the coefficients of terms containing $X2$. We built a second model by adding $X1$ and its interactions (Model 2). The RSE for this model was 0.035 and its p-value was $< 2.2e-16$. The addition of $X1$ modeled the cost of accessing the mesh data structure and the overheads caused in its accesses by $X2$ and $X3$, but it did not improve predictions. We compared models 1 and 2 using Anova. The results of comparison using Anova are tabulated in table 9. From table 9, $X1$ can be considered as a significant variable. However, for all discussions in this article, we have used Model 1 because the dominant data structure in the tracking kernel is the particle array and the association between particles and mesh elements is random. Lack of reuse in the mesh data structure can cause capacity misses in caches, but it does not affect the spatial locality in particle array accesses. The graph in figure 10 shows a plot of measured and predicted values for the training set for Model 1. The coefficients for interactions between $X2$,$X1$ and $X3$, can be improved by using data layouts that increase cache reuse in mesh array accesses. Threads sharing the same cache are likely to benefit from accessing adjacent particles and mesh elements.

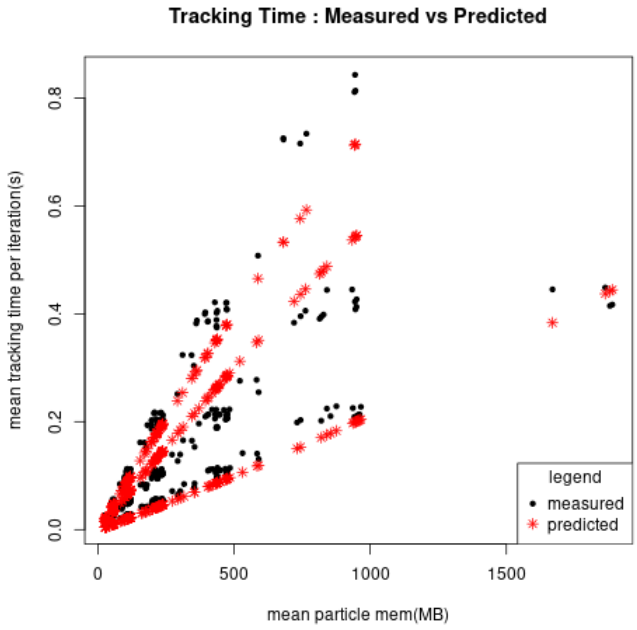

Figure 10: Mean Time per iteration(s) vs Mean Particle Memory(MB)

6.4.2 Energy Consumption

The energy consumption of a computational device can be divided into energy consumed by arithmetic circuits, static random access memories (SRAM) and dynamic random access memories (DRAM) [33]. The energy used by these components is further divided into idle and dynamic energies. Idle energy depends on the constant idle power drawn by the device as long as it is switched on. Dynamic energy is the additional energy consumed by the device to perform arithmetic and memory operations. Total energy can be computed as the product of the mean power drawn by a device and the duration for which it is used.

We used the following approach to model the energy consumed by a program written using the programming model discussed in section 3. Let E_s be the energy consumed by a program that runs for T_s seconds using d devices. Let P_{idle_i} and $P_{dynamic_i}$ be the idle and dynamic power drawn by the i^{th} device. E_{idle_i} and $E_{dynamic_i}$ are defined as the idle and dynamic energies consumed by device i. Let T_{s_i} be the duration for which the i^{th} device is used by the program. The energy models are described by equations 9, 10 and 11. Assuming devices run asynchronously, the execution time T_s of the program is the maximum execution time measured across all devices (CPUs and GPUs), shown in equation 8. For programs discussed in this article, T_s and E_s are time and energy values per iteration.

$$T_s = \max_{i=1}^{i=d} T_{s_i} \tag{8}$$

$$E_s = \sum_{i=1}^{i=d} (E_{idle_i} + E_{dynamic_i}) \tag{9}$$

$$E_{idle_i} = T_{s_i} * P_{idle_i} \tag{10}$$

$$E_{dynamic_i} = T_{s_i} * P_{dynamic_i} \tag{11}$$

We have assumed P_{idle_i} of a device as a constant. Dynamic power depends on the computational and communication complexities of parallel algorithms, their implementations and device parameters. We have ignored the energy consumed by the host CPU in this model since it did not perform any significant computation. We first built a model for the maximum power ($P_{idle_i} + P_{dynamic_i}$) drawn by the GPUs. Nvprof [16] was used to sample maximum wall power. All CUDA kernels were considered to determine variables for this model. The number of arithmetic operations and memory accesses in any kernel depends on the number of particles and mesh elements. The number of active hardware components can be approximated by the number of GPU threads per device. Therefore, the variables $X1$, $X2$ and $X3$ and interactions between them can be used to model dynamic power consumption. We have used linear dependence between memory sizes (accesses) and maximum power. The RSE for this three-variable model was 5788 and p-value was $< 2.2e-16$ for training data. It provided a better fit compared to a two-variable model with $X2$ and $X3$. From the coefficients in table 10, the terms $X1$ and $X1X2$ don't seem to be significant and can be dropped. The term $X1X2X3$ has higher significance and models the cost of both data structures being accessed together in a kernel by $X3$ threads. Across all kernels, maximum power is drawn at the instant when the maximum hardware componets are active. This is likely to happen when the maximum number of memory accesses are issued by all threads in a device. Maximum power consumption is reported in milliwatts(mW). We have ignored the kd-trees and used only mesh and particles data structures to determine maximum power. The graph in figure 11 shows the plot of measured and predicted values of maximum power drawn by each GPU for training data. The X-axis in graph 11 is the mean particle memory size. Trend lines have been added for different values of GPU threads.

	Estimate	**Stand.Error**	**t-value**	**Pr(>|t|)**
Intercept	7.111e+04	1.179e+03	60.320	$< 2e-16$
$X1$	3.414e-01	5.005e-01	0.682	0.495431
$X2$	2.143e+01	4.626e+00	4.634	4.73e-06
$X3$	9.956e+00	7.960e-01	12.508	$< 2e-16$
$X1X2$	1.511e-03	2.028e-03	0.745	0.456629
$X1X3$	1.128e-03	3.304e-04	3.414	0.000698
$X2X3$	-2.322e-03	2.689e-03	-0.864	0.388171
$X1X2X3$	-1.573e-06	1.197e-06	-1.314	0.189488

Table 10: Coefficients for 3-variable Power Model

For the energy model, we used the mean power values sampled by nvprof. A product of mean power and mean time per tracking iteration will be a measure of mean energy per tracking iteration per GPU with high probability. The mean power per device was added as variable $X4$. The coefficients for the energy model are provided in table 11. The measured and predicted values of mean energy per tracking iteration per GPU for training data are plotted in the graph in figure 12. The RSE for this model was 1316 and p-value was $< 2.2e-16$ for training data.

The mean energy per tracking iteration is reported in millijoules(mJ). The total mean energy per tracking iteration can be computed by multiplying the predicted energy values by the number of GPUs.

6.5 Model Validation

A set of 40 experiments were used to validate the models. The values for mesh size and particle count used in these experiments were different from those used in the training set.

6.5.1 Execution Time

Table 12 has the measured and predicted fit values (95 percent confidence interval) for mean tracking time per iteration. Some of the predicted values for lower fit values are negative. This can be avoided

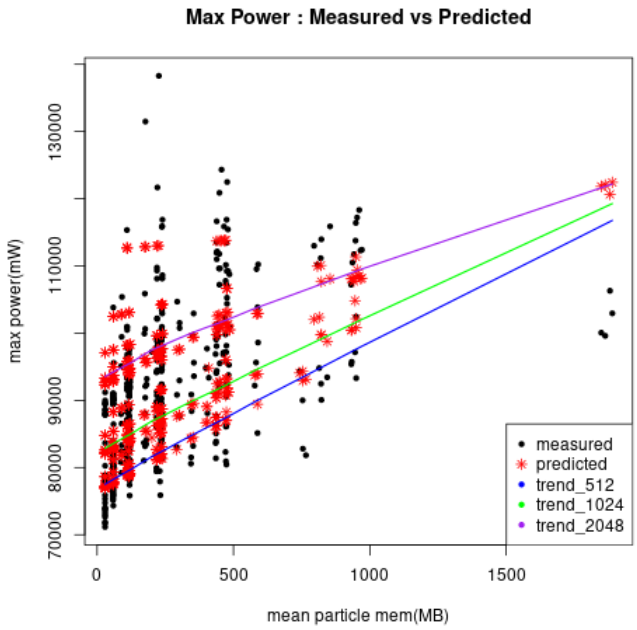

Figure 11: Maximum Power Consumption(mW) vs Mean Particle Mem(MB)

	Estimate	**Stand.Error**	**t-value**	**Pr(>|t|)**
$Intercept$	1.062e+05	5.187e+03	20.476	$< 2e-16$
$X2$	4.277e+06	3.277e+05	13.051	$< 2e-16$
$X2^2$	2.159e+06	3.214e+05	6.719	5.72e-11
$X3$	-5.787e+01	2.651e+00	-21.831	$< 2e-16$
$X4$	-1.246e+00	7.317e-02	-17.034	$< 2e-16$
$X2X3$	-2.196e+03	1.602e+02	-13.705	$< 2e-16$
$X2^2X3$	-1.004e+03	1.596e+02	-6.291	7.67e-10
$X2X4$	-5.486e+01	4.578e+00	-11.985	$< 2e-16$
$X2^2X4$	-3.063e+01	4.495e+00	-6.815	3.13e-11
$X3X4$	7.092e-04	3.663e-05	19.360	$< 2e-16$
$X2X3X4$	2.874e-02	2.231e-03	12.886	$< 2e-16$
$X2^2X3X4$	1.437e-02	2.222e-03	6.468	2.66e-10

Table 11: Coefficients for 3-variable Energy Model per iteration per GPU

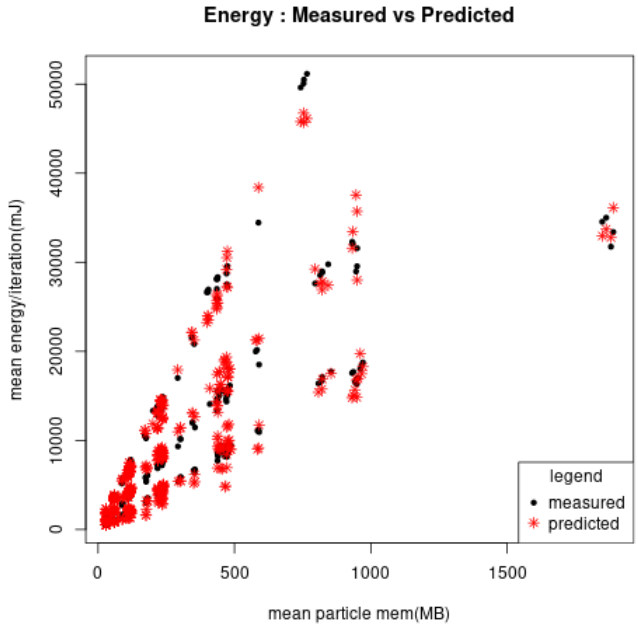

Figure 12: Mean Energy per Iteration(mJ) vs Mean Particle Mem(MB)

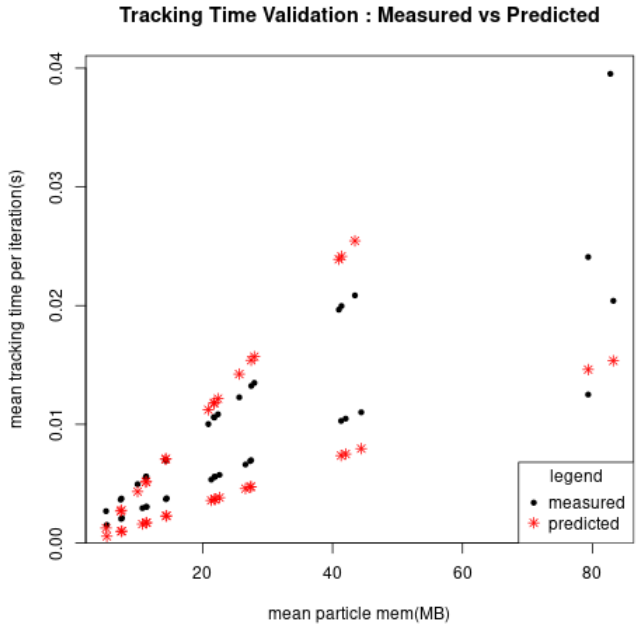

Figure 13: Validation Tests for Mean Tracking Time per Iteration

mean time/iteration(s)	fit	lower	upper
0.0015180	0.0005636855	-0.0083176986	0.009445070
0.00293745	0.0015938210	-0.0071198664	0.010307508
0.00534056	0.0035872691	-0.0048113722	0.011985910
0.00661307	0.0045811382	-0.0036652068	0.012827483
0.0125028	0.0146092272	0.0077065583	0.021511896
0.00268073	0.0012551619	-0.0055000981	0.008010422
0.00495556	0.0043324235	-0.0022368314	0.010901678
0.010019	0.0112227041	0.0050544954	0.017390913
0.0122708	0.0142292807	0.0082290804	0.020229481
0.0240905	0.0479666805	0.0435089576	0.052424403
0.00689969	0.0047158626	-0.0035100869	0.012941812
0.00697341	0.0047453477	-0.0034761462	0.012966842
0.0102853	0.0073644974	-0.0004728519	0.015201847
0.0104714	0.0074973320	-0.0003211594	0.015315823
0.0204006	0.0153503870	0.0085320038	0.022168770
0.0132328	0.0153907370	0.0094542578	0.021327216
0.0134883	0.0156974431	0.0097776791	0.021617207
0.0199637	0.0241506441	0.0186725259	0.029628762
0.0196578	0.0238995052	0.0184088126	0.029390198
0.0395256	0.0501024540	0.0457179332	0.054486975
0.00369186	0.0022737125	-0.0063311074	0.010878532
0.00376819	0.0022940840	-0.0063074964	0.010895664
0.00557681	0.0036958279	-0.0046860217	0.012077678
0.00558542	0.0036803723	-0.0047038656	0.012064610
0.0110117	0.0079523051	0.0001979377	0.015706672
0.0069363	0.0070945963	0.0006887255	0.013500467
0.00699061	0.0070513170	0.0006429128	0.013459721
0.0105739	0.0118109752	0.0056759821	0.017945968
0.0106016	0.0117629571	0.0056252589	0.017900655
0.020857	0.0254538591	0.0200404339	0.030867284
0.00203546	0.0009772452	-0.0078364205	0.009790911
0.00209728	0.0010003949	-0.0078094958	0.009810286
0.00304786	0.0017169165	-0.0069769527	0.010410786
0.00304273	0.0016969942	-0.0070000793	0.010394068
0.00573539	0.0038266488	-0.0045350166	0.012188314
0.00365392	0.0026818858	-0.0039866212	0.009350393
0.00374118	0.0027572193	-0.0039067316	0.009421170
0.00556004	0.0051340477	-0.0013874379	0.011655533
0.00560097	0.0051739338	-0.0013451824	0.011693050
0.01084	0.0121578217	0.0060423350	0.018273308

Table 12: Predictions from Performance Model

by taking logarithms. The predicted values were found to be quite close to measured values as shown in figure 13. The RSE for validation data was 0.035, close to the RSE for training data.

6.5.2 Maximum Power

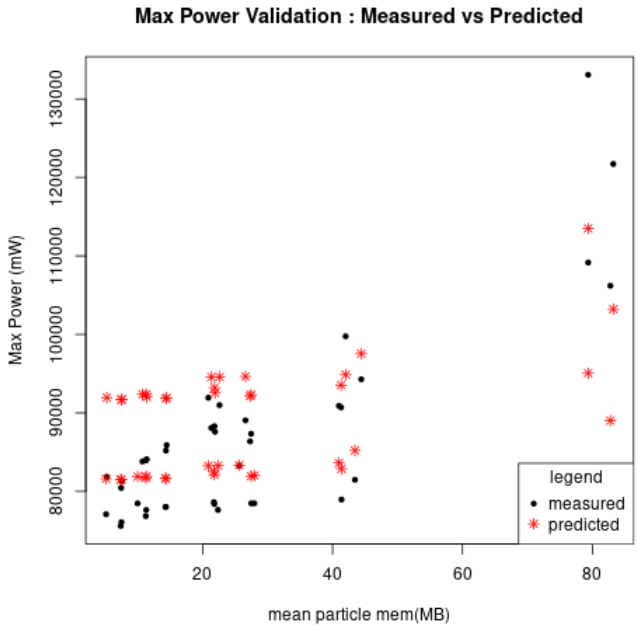

Figure 14: Validation Tests for Maximum Power

Table 13 has measured and predicted fit values for maximum power drawn by any GPU during the execution of the program. All predicted values are reported within a confidence interval of 95 percent. The measured and predicted fit values are also plotted in figure 14. The RSE for power model for validation data was 5788.197, close to the RSE for training data. The predictions for maximum power can be improved with more training data.

6.5.3 Energy Consumption

The measured and predicted values for energy consumed per tracking iteration per GPU are tabulated in table 14 and plotted in figure 15. All values are predicted within a confidence interval of 95 percent. The predicted values for energy are quite close to measured values. The RSE for power model for validation data was 1316.058, close to the RSE for training data.

7 Conclusions

Performance and energy models were built for hybrid programs by following a programmer's approach of viewing host-device programs as composed of different types of memory accesses. The models helped to understand relationships between total execution time and different types of memory accesses and energy consumption of programs. They were also useful as tools to compare different devices and evaluate the programming model and machine architecture. For example, there are certainly gains from keeping GPUs independent and exposing their memories to peer GPUs. It removed the explicit cost of copying data between host and devices by caching pages. But what we observed in the experiments and models is that allocating the entire memory on devices can cause overheads

maxpower(mW)	**fit**	**lower**	**upper**
81829	91927.30	90461.15	93393.45
83807	92357.50	90951.79	93763.20
88077	94547.97	93374.76	95721.18
89039	94626.56	93462.15	95790.98
133080	113497.95	109944.73	117051.17
77074	81593.43	80514.81	82672.05
78464	81878.97	80851.98	82905.95
91922	83239.70	82400.15	84079.24
83271	83329.91	82500.59	84159.23
109153	95048.04	92508.54	97587.54
86367	92122.24	90683.84	93560.64
87328	92292.94	90878.58	93707.30
90691.5	93518.94	92258.98	94778.90
99754	94856.54	93717.17	95995.91
121719	103207.36	101585.68	104829.04
78464.5	81922.61	80892.54	82952.67
78464.5	82028.27	81017.54	83039.00
78945	82860.82	81978.83	83742.81
90900.5	83620.64	82823.19	84418.09
106200	89001.13	87849.29	90152.96
85192.75	91822.40	90340.85	93303.96
85885.75	91909.33	90440.54	93378.11
87581.75	92538.82	91157.87	93919.78
88287.75	93211.03	91913.49	94508.58
94274.25	97541.15	96435.98	98646.32
77996	81624.07	80544.86	82703.28
78009.5	81671.22	80601.46	82740.97
78370.25	82103.96	81110.55	83097.37
78610.5	82487.50	81557.28	83417.73
81466	85201.99	84414.15	85989.83
80426.625	91664.80	90159.87	93169.72
81314.625	91709.78	90211.60	93207.95
84083.5	92028.36	90576.80	93479.92
83942.125	92366.41	90961.94	93770.88
90980	94566.91	93395.83	95737.99
75580.125	81466.95	80360.88	82573.02
76060.75	81493.55	80392.89	82594.22
76841.625	81710.03	80650.05	82770.01
77616.375	81904.26	80881.10	82927.41
77622.875	83267.75	82431.40	84104.10

Table 13: Predictions from Power Model

mean energy/iteration(mJ)	fit	lower	upper
108.936903362	368.6066	-432.703829	1169.9170
218.943481989	339.8362	-217.547933	897.2203
410.2418989112	413.7265	20.508367	806.9447
514.2982555079	471.7241	123.802601	819.6456
929.78885106	978.8382	534.721617	1422.9549
192.3244970309	143.6491	-179.317111	466.6153
361.862176762	114.6940	-249.437475	478.8255
737.59787829	463.2939	80.050729	846.5371
905.341341912	634.6731	250.093133	1019.2531
1718.835252645	3144.8458	2923.612331	3366.0793
535.7821105483	481.3755	130.688834	832.0621
546.08156563215	481.0291	147.394458	814.6637
813.3291767315	677.6897	367.336971	988.0424
818.700963872	676.5948	357.869026	995.3206
1575.255483681	1179.5721	876.025678	1483.1186
987.329841932	578.4775	127.452160	1029.5028
1004.699495142	618.8817	180.548728	1057.2147
1482.8975470715	1210.2213	824.196339	1596.2463
1446.418171908	1298.1803	962.359319	1634.0014
2867.598287868	3124.5946	2901.418580	3347.7706
272.4124736745	386.7212	-220.084338	993.5267
280.2449628556	377.0485	-184.636723	938.7337
426.708292657625	423.4788	15.783272	831.1744
422.7347189409	430.7239	-25.392385	886.8401
835.40534589575	673.6071	264.451497	1082.7627
501.62255143875	413.8621	94.588258	733.1360
508.3510074632	342.8000	5.560508	680.0395
771.01444925975	617.1062	283.829533	950.3829
769.996831896	660.9652	342.427035	979.5034
1505.2565206675	1607.5060	1340.761559	1874.2504
140.6791590001	450.6951	-581.914695	1483.3049
146.18697	435.8396	-542.413859	1414.0930
221.11889797365	388.6878	-326.527064	1103.9027
216.850460697875	415.7200	-409.670639	1241.1106
419.394665529238	461.9734	-176.998191	1100.9450
251.0084642568	772.9187	271.591730	1274.2457
261.59687673045	560.1559	171.054974	949.2569
389.5473695789	679.5373	308.130099	1050.9445
395.984315261587	571.5765	237.923025	905.2301
761.73927955	1071.8061	734.740429	1408.8717

Table 14: Predictions for Energy Model/device

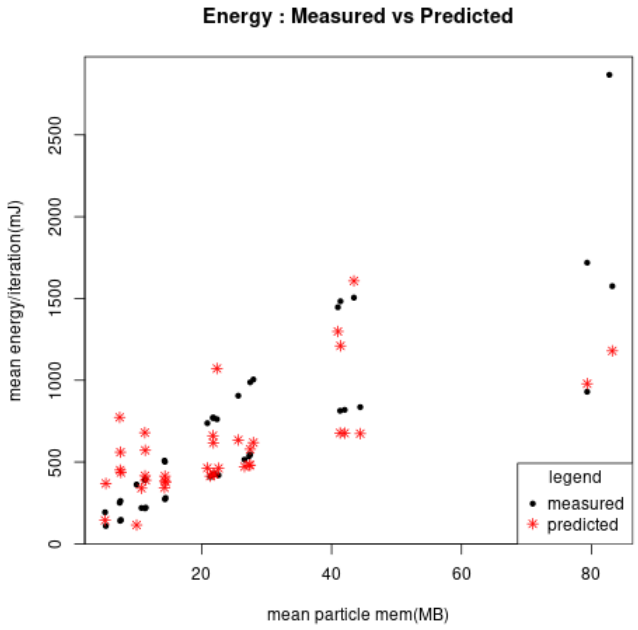

Figure 15: Validation tests for Mean Energy per Iteration

for large test cases through page faults and cache misses. Increasing the number of peer GPUs per group also caused overheads for programs by increasing page faults on the shared CPU. In our energy models we observed the dependence on memory accesses and overheads such as cache misses and page faults. For both types of iterative programs, the models seem to predict execution time quite well, for the entire memory range. Although the dynamic iterative program had two data structures, the tracking kernel did not involve close interaction between them, which made experiment design and the model quite simple. Modeling the interplay between multiple dynamic data structures will require careful experiment design. The communication and power models need improvement.

8 Future Work

One of the future directions is to explore other programming models for heterogenuous programs along with different memory allocation schemes. One of the dominant factors found in the performance models is the relationship between memory usage and time per iteration and energy per iteration. It is possible to extend the programming model considered in this article by allocating some percentage of total memory on the host and staging memory allocation on devices, so that devices are always operating in their best range, in terms of memory. Since cache misses and page faults were found to be significant factors in determining the total execution time, the effort should be to design memory allocation and data layout schemes that minimize these overheads. Space-filling curves [37] are an option for partitioning and data layout that improves spatial locality in kernels. Better spatial locality affects both execution time and dynamic power by increasing cache hits. Future work also includes extending performance models for dynamic iterative programs to include load balancing costs. The power (energy) models have to be extended to include host CPUs. Trade-offs between maximum power and execution time per iteration can be determined to obtain a set of desirable operating ranges for the resources used by a program that runs for several iterations. The performance models will be used to study scalability of applications. It will also be used to decide resource allocation (type and number of devices) for applications by considering both execution time and maximum power consumption. We have not included strong scaling predictions in

this article because the cluster size was limited. Future work includes training experiments on larger number of host-GPU groups, improving the models and using them for strong scaling predictions.

9 Related Work

The most common method to build performance models is by expressing a program as a sequence of sub-programs with known analytical models. These models are then composed to build a full analytical model for the program, as a function of its parameters [6]. This technique has been used quite extensively for scalability studies of parallel algorithms on several large machines. They have been used to model communication performance for different network hardware at scale [48]. There are several issues with this technique :

1. It is not always possible to express large programs as sequences of known sub-programs. Some programs may have complex interactions between sub-programs.
2. Analytical models depend heavily on the algorithms used and it is sometimes a non-trivial task to extract dependent parameters for all algorithms in a large application. They may also give undue importance to some parameters, while ignoring others. It may be difficult to isolate and identify significant dependencies from analytical models.

Several papers have discussed the use of statistical methods to model the performance of programs [11] [4] [26] [13] [22] [28] [10]. Most models have used linear regression as their method of choice. Some of these statistical models did not differ much from analytical models in the way they modeled the problem. It almost seemed like they were searching for the analytical model of the algorithm using methods from statistics. Some approaches have used a combination of known analytical and statistical models to build performance models. Most of these techniques were used to identify scalability bugs in algorithms by using a training set at lower number of processes and predicting performance at large scale [12] [40]. Some methods depended on profiling tools to collect detailed runtime data and used them to populate training data [41]. A related topic is the use of simulators to mimic and model the performance of processors or accelerators. This method is often used to model the performance of new hardware and they focus less on applications. The programs used to build these models are usually very simple and they take several hours or even days to generate simulated data [43] [2]. There has recently been a lot of work in connecting multiple GPUs into a peer group that can access each other's memories via RDMA [39] or communicate directly using MPI [32]. Much of the current work in the domain of GPU computing is around the possibility of an autonomous multi-GPU machine. The overheads of such a machine have not been studied much. It is also not clear how a well-written scalable program would compare on a CPU-GPU architecture verses an autonomous architecture. [18] has developed theoretical lower-bounds for communication and energy for different representative programs on CPUs and GPUs. Several papers have discussed models for power using linear regression [33]. Roofline power models for GPUs are described in [14], [19]. However, they have used relatively simple micro benchmarks as test cases for their models. Complex programming models and modern machine architectures have not been considered by these models. Powerpack [17] discusses a set of tools for measuring runtime power consumed by different hardware components.

References

[1] Standard for information technology–portable operating system interface (posix(r)) base specifications, issue 7, Sept 2016.

[2] Y. Arafa, A. A. Badawy, G. Chennupati, N. Santhi, and S. Eidenbenz. Ppt-gpu: Scalable gpu performance modeling. *IEEE Computer Architecture Letters*, 18(1):55–58, 2019.

[3] Ben Ashbaugh, Alexey Bader, James Brodman, Jeff Hammond, Michael Kinsner, John Pennycook, Roland Schulz, and Jason Sewall. Data parallel c++: Enhancing sycl through extensions

for productivity and performance. In *Proceedings of the International Workshop on OpenCL*, IWOCL '20, New York, NY, USA, 2020. Association for Computing Machinery.

[4] Bradley J. Barnes, Barry Rountree, David K. Lowenthal, Jaxk Reeves, Bronis de Supinski, and Martin Schulz. A regression-based approach to scalability prediction. In *Proceedings of the 22nd Annual International Conference on Supercomputing*, ICS '08, page 368–377, New York, NY, USA, 2008. Association for Computing Machinery.

[5] J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324:446–449, December 1986.

[6] G. Bauer, S. Gottlieb, and T. Hoefler. Performance modeling and comparative analysis of the milc lattice qcd application su3rmd. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 652–659, 2012.

[7] Michael Edward Bauer. *Legion: Programming distributed heterogeneous architectures with logical regions*. PhD thesis, Stanford University, 2014.

[8] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. Groute: An asynchronous multi-gpu programming model for irregular computations. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, page 235–248, New York, NY, USA, 2017. Association for Computing Machinery.

[9] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[10] Arnamoy Bhattacharyya and Torsten Hoefler. Pemogen: Automatic adaptive performance modeling during program runtime. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, page 393–404, New York, NY, USA, 2014. Association for Computing Machinery.

[11] A. Calotoiu, D. Beckinsale, C. W. Earl, T. Hoefler, I. Karlin, M. Schulz, and F. Wolf. Fast multi-parameter performance modeling. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 172–181, 2016.

[12] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf. Using automated performance modeling to find scalability bugs in complex codes. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2013.

[13] Jaemin Choi, David F. Richards, Laxmikant V. Kale, and Abhinav Bhatele. End-to-end performance modeling of distributed gpu applications. In *Proceedings of the 34th ACM International Conference on Supercomputing*, ICS '20, New York, NY, USA, 2020. Association for Computing Machinery.

[14] JeeWhan Choi, Marat Dukhan, X. Liu, and R. Vuduc. Algorithmic time, energy, and power on candidate hpc compute building blocks. *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 447–457, 2014.

[15] P. Colella, D. T. Graves, J. N. Johnson, H. S. Johansen, N. D. Keen, T. J. Ligocki, D. F. Martin, P. W. Mccorquodale, D. Modiano, P. O. Schwartz, T. D. Sternberg, and B. Van Straalen. Chombo software package for amr applications design document. Technical report, LBNL, 2003.

[16] Shane Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[17] R. Ge, Xizhou Feng, Shuaiwen Song, Hung-Ching Chang, Dong Li, and K. Cameron. Powerpack: Energy profiling and analysis of high-performance systems and applications. *IEEE Transactions on Parallel and Distributed Systems*, 21:658–671, 2010.

[18] Andrew S. Gearhart. Bounds on the energy consumption of computational kernels. 2014.

[19] M. Ghane, J. Larkin, Larry Shi, S. Chandrasekaran, and M. Cheung. Power and energy-efficiency roofline model for gpus. *ArXiv*, abs/1809.09206, 2018.

[20] A. Gray. Accelerating weather prediction with nvidia gpus. 09 2018.

[21] T. Hastie, R. Tibshirani, and J.H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer series in statistics. Springer, 2009.

[22] Engin Ipek, Bronis R. de Supinski, Martin Schulz, and Sally A. McKee. An approach to performance prediction for parallel applications. In *Proceedings of the 11th International Euro-Par Conference on Parallel Processing*, Euro-Par'05, page 196–205, Berlin, Heidelberg, 2005. Springer-Verlag.

[23] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning: With Applications in R.* Springer Publishing Company, Incorporated, 2014.

[24] James Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[25] I Karlin. Lulesh programming model and performance ports overview. 12 2012.

[26] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *SC '01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, pages 39–39, 2001.

[27] Ronan Keryell, Ruyman Reyes, and Lee Howes. Khronos sycl for opencl: A tutorial. In *Proceedings of the 3rd International Workshop on OpenCL*, IWOCL '15, New York, NY, USA, 2015. Association for Computing Machinery.

[28] Benjamin C. Lee, David M. Brooks, Bronis R. de Supinski, Martin Schulz, Karan Singh, and Sally A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, page 249–258, New York, NY, USA, 2007. Association for Computing Machinery.

[29] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Array, Trees, Hypercubes.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1991.

[30] Unified Memory. http://on-demand.gputechconf.com/gtc/2017/presentation/s7285-nikolay-sakharnykh-unified-memory-on-pascal-and-volta.pdf, 2017.

[31] Aaftab Munshi, Benedict Gaster, Timothy G. Mattson, James Fung, and Dan Ginsburg. *OpenCL Programming Guide.* Addison-Wesley Professional, 1st edition, 2011.

[32] MVAPICH-CUDA. https://developer.nvidia.com/mvapich.

[33] Kenneth O'brien, Ilia Pietri, Ravi Reddy, Alexey Lastovetsky, and Rizos Sakellariou. A survey of power and energy predictive models in hpc systems and applications. 50(3), June 2017.

[34] R Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2014.

[35] Prabhakar Raghavan Rajeev Motwani. *Randomized Algorithms.* Cambridge University Press, 2007.

[36] David Richards, Patrick Brantley, Shawn Dawson, Scott Mckenley, and Matthew O'Brien. Quicksilver, version 00. 3 2016.

[37] Aparna Sasidharan. A distributed multi-threaded data partitioner with space-filling curve orders. 2018.

[38] Alina Sbîrlea, Yi Zou, Zoran Budimlíc, Jason Cong, and Vivek Sarkar. Mapping a data-flow programming model onto heterogeneous platforms. *ACM SIGPLAN Notices*, 47(5):61–70, 2012.

[39] Gilad Shainer, Ali Ayoub, Pak Lui, Tong Liu, Michael Kagan, Christian R. Trott, Greg Scantlen, and Paul S. Crozier. The development of mellanox/nvidia gpudirect over infiniband–a new model for gpu to gpu communications. *Comput. Sci.*, 26(3–4):267–273, June 2011.

[40] Sergei Shudler, Alexandru Calotoiu, Torsten Hoefler, Alexandre Strube, and Felix Wolf. Exascaling your library: Will your implementation meet your expectations? In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, page 165–175, New York, NY, USA, 2015. Association for Computing Machinery.

[41] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for performance modeling and prediction. In *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 21–21, 2002.

[42] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 4th edition, 2013.

[43] Yifan Sun, Trinayan Baruah, Saiful A. Mojumder, Shi Dong, Xiang Gong, Shane Treadway, Yuhui Bao, Spencer Hance, Carter McCardwell, Vincent Zhao, Harrison Barclay, Amir Kavyan Ziabari, Zhongliang Chen, Rafael Ubal, José L. Abellán, John Kim, Ajay Joshi, and David Kaeli. Mgpusim: Enabling multi-gpu performance modeling and optimization. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 197–209, New York, NY, USA, 2019. Association for Computing Machinery.

[44] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson, R. Roskies, J. R. Scott, and N. Wilkins-Diehr. Xsede: Accelerating scientific discovery. *Computing in Science & Engineering*, 16(5):62–74, Sept.-Oct. 2014.

[45] Tesla K80 Whitepaper. www.nvidia.com, 2015.

[46] Tesla P100 Whitepaper. www.nvidia.com, 2016.

[47] Volta V100 Whitepaper. www.nvidia.com, 2017.

[48] Jidong Zhai, Wenguang Chen, and Weimin Zheng. Phantom: Predicting performance of parallel applications on large-scale parallel machines using a single node. *SIGPLAN Not.*, 45(5):305–314, January 2010.