A Cost Model for Heterogeneous Skeletons for CPU/GPU Systems

Khari A. Armih
College of Computer Technology, Zawia, Libya
khari.armih@gmail.com

Mustafa K. Aswad
Faculty of Engineering, Sabratha University, Libya
mustafaasawd@gmail.com

Abstract

Algorithmic skeletons are widely used to manage multi-processor computations but are most effective when deployed for regular problems on homogeneous systems, where tasks may be divided evenly without regard for processor characteristics. With the growth in heterogeneity, where a multicore is coupled with GPUs, skeletons become layered and simple task distribution becomes sub-optimal. We explore heterogeneous skeletons which use a simple cost model based on a small number of key architecture characteristics to find good task distributions on heterogeneous multicore architectures. We present a new extension to an existing skeleton library associated cost model that enable GPUs to be exploited as general purpose multi-processor devices in heterogeneous multicore/GPU systems. The extended cost model is used to automatically find a good distribution for both a single heterogeneous multicore/GPU node, and clusters of heterogeneous multicore/GPU nodes.

General Terms Algorithms, Design, Performance

Keywords Parallel, Skeleton, Heterogeneous, Cost model, multicore, GPU

1. Introduction

Graphical Processing Units (GPUs) were designed as specialized processors to accelerate graphics processing. Recently, however, the architectures that are comprised of multicores of GPUs have become ubiquitous and cost effective platform for both graphics and general-purpose parallel computing as they offer extensive resources such as high memory bandwidth and massive parallelism [4]. Compared to a conventional core, the performance of a GPU comes from creating a large number of lightweight GPU threads with negligible overheads, where in general purpose multicore the limited number of cores limits the number of data elements that can be processed simultaneously. Today's GPUs enable non-graphics programmers to exploit the parallel computing capabilities of a GPU using data parallelism. With the programmability available on the GPU, a new technique called General Purpose computation on GPU (GPGPU) has been developed [7]. Many parallel applications have achieved significant speedups with GPGPU implementations on a GPU over the CPU [3].

We have been exploring heterogeneous skeletons which use a simple cost model, based on a small number of key architecture characteristics, to find good task distributions on heterogeneous multicore architectures. We have constructed the *HWSkel* library [2] for heterogeneous architectures composed of distributed memory clusters of shared memory multi-core processors. The library, coupled with a simple cost model, offers good speed up for regular data parallel programs.

In this paper, we present the *GPU-HWSkel* extension to our skeletons, which enable GPUs to be exploited as general purpose multi-processor devices in heterogeneous multicore/GPU architectures. We also present an extension to our *HWSkel* cost model which may be used to automatically find a good distribution for both a single heterogeneous multicore/GPU node, and cluster of heterogeneous multicore/GPU nodes.

2. *GPU-HWSkel*: A CUDA-Based Skeleton Library

GPU-HWSkel is an extension of the *HWSkel* library [1, 2] which was designed for heterogeneous multicore cluster architectures. The *HWSkel* library provides data parallel heterogeneous skeletons such as *hMapReduce* and *hMapReduceAll*. They are novel in supporting execution on heterogeneous architectures, and facilitate performance portability, using an architectural cost model to automatically balance load across heterogeneous components of the architecture. *HWSkel* cost model characterise components of the architecture by the number of cores `C`, clock speed `S`, and crucially the size of the L2 cache `L2`, where the relative strength `Strength`[1] of node i of heterogeneous multicore cluster is given by:

$$Strength_i = C_i * S_i * L2_i \quad (1)$$

It is important to note that:

- our model excludes network communication costs;
- L2 is used as a shorthand for the top level shared cache which may be L3 on some processors

The new library is designed with the aim of providing a high-level parallel programming environment to program parallel heterogeneous systems including single- and multicore CPU, and GPU architectures. The *GPU-HWSkel* library is based on the CUDA programming model to make GPGPU accessible on NVIDIA GPUs. This makes our approach limited to NVIDIA architectures. The new library implements the same set of data-parallel skeletons that are provided by the base *HWSkel* library [2], *hMap* and *hReduce* parallel skeletons, *hMapReduce* skeleton, and *hMapReduceAll* skeleton. These skeletons provide a general interface for both GPUs and CPUs since the library is based on OpenMP and MPI to support CPU implementations, as well as CUDA for GPU implementations.

[1] In [2] we termed Strength "Power" but we now feel that this has inappropriate absolute connotations

3. A GPU Workload Distribution Cost Model

We next introduce an extension of our approach to account for the GPU as an independent processing element and to automatically find a good distribution in heterogeneous multicore/GPU systems. A new model is designed to provide a generic load-balancing strategy, so our skeletons will fully automate the distribution process on an integrated multicore/GPU architecture, which in turn makes the task of workload distribution much easier for the skeleton programmer.

3.1. Related Work

Several performance cost models have been developed to utilise the high performance of heterogeneous parallel systems. However, to the best of our knowledge little research has been done in considering the use of multiple cores and a GPU card simultaneously in heterogeneous architectures. This section briefly describes related models that consider using the heterogeneous multicore/GPU systems.

In [6] a performance cost model has been introduced in conjunction with a 2D-FFT library for finding the optimal distribution ratios between CPUs and GPUs. The model predicts the total execution time of a 2D-FFT of arbitrary data size. Firstly, the FFT computation is split into small steps, and then the model predicts the execution time for each execution step using profiling results on a heterogeneous multicore/GPU system, and finally the model determines the optimal load distribution ratio as the shortest predicted execution time.
Moreover, the model attempts to overcome the limitation in the memory sizes of GPUs by iterating GPU library calls.

An adaptive mapping technique is implemented in the heterogeneous programming system called Qilin [5] for computation placement on heterogeneous multiprocessors. It is a fully automatic approach to find the optimal computation mapping to processing elements of a heterogeneous system. Qilin has a capability to use any heterogeneous platform, since it does not require any hardware information for its implementations. This technique uses execution

time projections stored in a database to determine the execution times of both CPU and GPU for a given program, problem size and hardware configuration. Further, the determined execution times are used to statically partition the workload among the CPU and GPU. Thus, the first step in the Qilin programming system is to conduct a training run to add data to a database.

An optimisation framework is introduced in [8] to improve the load balance on heterogeneous multicore/GPU systems. Instead of using static partitioning, the model applies a new adaptive technique that dynamically balances the workload distribution between the CPU cores and the GPU in a single node. At the beginning of execution, the model measures the performance of both the CPU and the GPU, and then the measurement is used to guide the workload distribution in the next step. In addition, the model tries to hide the communication overhead of transferring the data between CPU and GPU by providing software pipelining to overlap data transfers and kernel execution.

3.2. Discussion

Load-balancing at the multi-node heterogeneous hardware level can either be done dynamically or statically before program execution is started. Static cost models incur less overhead than dynamic models due to their simplicity and lack of runtime overhead. Besides, heterogeneous integrated multicore/GPU systems are nonetheless highly distributed. Hence, we wish to develop an accurate cost model and prediction mechanism to balance the workload distribution across the CPU and the GPU in each node as well as between the nodes in a cluster. The new cost model inherits all the features of the *HWSkel* cost model presented in [2]. However, in contrast to the performance cost models described previously, our cost model provides the following new features:

Heterogeneous-mode. The performance cost models in [5, 6, 8] measure the performance of both the CPU and the GPU in a heterogeneous system by using profiling. Our performance cost model

is based on two different type of performance measurements for the CPU and GPU. Since the performance of the GPU is changed by changing the data size while the performance of the CPU can be more stable for different data sizes, we measure the performance of the GPU with a training run, while the CPU performance is calculated using the hardware parameters.

Hardware-auto-selection. Since our performance cost model can provide enough information about the CPU and GPU performance capability, our heterogeneous skeletons can choose to use either the multicore CPU or a GPU card to execute the ongoing program. This feature will be discussed in more detail in the future

3.3. Methodology

The new model is viewed as two-phase since the underlying target hardware consists of two levels of heterogeneous hardware architectures. The model is divided into two main components:

- *Single-Node*, to guide the workload distribution across the multiple cores and the GPU device inside each node in the integrated multicore/GPU system;
- *Multi-Node*, to balance the workload across the nodes in the cluster.

In general, we focus on predicting the runtime of the application code on the GPU device and use the *HWSkel* cost model [2] for measuring the processing strength of the CPU. In addition, since the workload is statically distributed across the multiple cores and the GPU and also between the nodes at the beginning of program execution, the model does not allow for any communication between the CPU cores and the GPU or between the nodes in the system other than via the skeleton.

3.3.1. Single-Node Cost Model

We base the workload distribution on the performance ratio between the core and GPU in the integrated multicore/GPU computing node. So the cost model aims to predict the execution time of a single core vs. the GPU device for arbitrary data sizes, and calculates the chunk size for a CPU core and the GPU by using this performance ratio. To facilitate our discussion, let us introduce the following notation:

`TC`: Program runtime on a single core.
`TG`: Program runtime on the GPU.
`Strength`: The relative strength of computational unit.
`C` : Number of cores in a single node.
`D` : Data Size.

We start by calculating Strength, the relative strengths of the GPU and a single core:

$$Strength = TC / TG$$

If the GPU is allocated DGPU units of data then the multicore will

$$D_{GPU} * Strength / (C - 1)$$

units. As the node comprises a multicore and a single GPU, the total data size is

$$D_{total} = D_{GPU} + D_{GPU} * Strength / (C - 1)$$

Factoring out DGPU, the data allocated to the multicore is

$$D_{multicore} = D_{total} / (1 + Strength / (C - 1))$$

and the each core is allocated

$$D_{core} = D_{multicore} / (C - 1)$$

3.3.2. Multi-Node Cost Model

The *Multi-Node* cost model is based on the *Single-Node* cost model to determine the chunk size for each node in the system. As a heterogeneous cluster might have different kinds of computing nodes, the key idea of the *Multi-Node* cost model is to measure the relative strength for each node in the cluster. Hence, the total available strength Strength for n nodes is given by:

$$Strength_{total} \sum_{i=1}^{i=n} Strength_i$$

So for data size D_{total}, the chunk size for node i is:

$$(Strength_i / Strength_{total}) * D_{total}$$

Nodes may have different architectures, and hence strengths. The relative strength of a node i that consists of a GPU and multiple cores

is the sum of the relative strengths of the cores, $Strength_{core}$, and the GPU $Strength_{GPU}$:

$$Strength_i = (Ci-1) * Strength_{corei} + Strength_{GPUi} \quad (2)$$

if there is only a single core, i.e. $C = 1$, it follows directly that

$$Strength_i = Strength_{GPUi} \quad (3)$$

To calculate $Strength_{core}$ and $Strength_{GPU}$, we first measure T_{Cbase}, the runtime of the program on core of the system, and use it follows.

$$Strength_{GPUi} = TC_{base} / TG_i \quad (4)$$

In practice we predict the relative strengths on the base core, $\mathtt{Strength}_{\mathtt{Cbase}}$, and on the cores of node $\mathtt{i}$, $\mathtt{Strength}_{\mathtt{Ci}}$ using the *HWSkel* cost model, i.e. Equation 1 in Section 2:

$$\mathtt{Strength}_{\mathtt{Ci}} = \mathtt{S}_{\mathtt{i}} * \mathtt{L2}_{\mathtt{i}} \quad (5)$$

$$\mathtt{Strength}_{\mathtt{Cbase}} = \mathtt{S}_{\mathtt{base}} _ \mathtt{L2}_{\mathtt{base}} \quad (6)$$

Hence the relative strength of a core on node $\mathtt{i}$ is:

$$\mathtt{Strength}_{\mathtt{corei}} = \mathtt{Strength}_{\mathtt{Ci}}/\mathtt{Strength}_{\mathtt{Cbase}} \quad (7)$$

Substituting equations (4) and (7) in (2) gives the cost equation used in the *GPU-HWSkel* library:

$$\mathtt{Strength}_{\mathtt{i}} = (\mathtt{C}_{\mathtt{i}}-1)*\mathtt{Strength}_{\mathtt{Ci}}/\mathtt{Strength}_{\mathtt{Cbase}} + \mathtt{Strength}_{\mathtt{GPUi}} \quad (8)$$

The key point is that we need only measure $\mathtt{TG}_{\mathtt{i}}$ and $\mathtt{TC}_{\mathtt{base}}$ to parametrise the model.

4. *GPU-HWSkel* Evaluation

4.1. Benchmarks

The performance of each *GPU-HWSkel* skeleton is evaluated using two applications: the first is the widely used matrix multiplication, and the second is an iterative Fibonacci program.

Matrix Multiplication. A well-known representative for a wide range of high-performance applications is the problem of multiplying two matrices. There are a number of different techniques to multiply matrices. Here, the number of multiplications performed is reduced by breaking down the input matrices into several submatrices.

Fibonacci Program. Fibonacci is a function that computes Fibonacci numbers. In our experiment, we use a simple program that calculate the Fibonacci value for an array of integer numbers with fixed constant by replicating the fib function in the original sequential program. In the parallel version, the array of the integers is split into chunks using a split function which employs the cost model for load distribution, and then the fib function is mapped in parallel across each chunk.

4.2. Platform

We conduct our experiments on a heterogeneous cluster with a number of different integrated multicore/GPU nodes located at Heriot-Watt University as described in Table 1. Each of the machines is connected to an NVIDIA GeForce GT 520 GPU device. The device has 1 GB of DRAM , one multiprocessor (MIMD unit) clocked at 810 MHz, and 48 processor cores (SIMD units) running at 1620 MHz with 16 KB of shared memory. CUDA version 4.0 was used for the experiments. The CUDA code was compiled using the NVIDIA CUDA Compiler (NVCC) to generate the device code that is launched from the host CPU.

Table 1: Experimental Architectures.

Name	**CPU**				**GPU**			
	archi	**Cores**	**MHz**	**L2**	**archi**	**SM**	**Cores**	**MHz**
lxpara	Xeon 5410	8	1998	6144KB	GT520	1	48	1620
lxphd	IntelE8400	2	1998	6144KB	GT520	1	48	1620
linuxlab	2 DuoCPU	2	1200	2048KB	G 520	1	48	1620
brahma	Xeon(TM)	4	3065	512KB	GT520	1	48	1620

4.3. Performance Evaluation

For all the measurements that are performed on the two-core (such as *linux* and *lxphd*) machines, we follow the common practice of increasing the input-data size to evaluate the behaviour consistency of the *hMap* skeleton with our performance cost model. While in the case of using machines with an eight-core processor (such as *lxpara*), all programs are measured with a fixed data size on 1,2,3,4,5,6, and 7 cores together with a single GPU device. We measure the runtimes for the *hMap* skeleton implementation, with a fixed data size of 1500 x 1500 for the input matrices, and 80,000 elements of Fibonacci (1,000,000).

4.3.1. Single multicore/GPU Node Results

The single-node experiments have been carried out our on *linux* lab, *lxphd*, and *lxpara* as single nodes.

Table 2 and 3 show the *hMap* runtime for matrix multiplication and Fibonacci on *linux* lab and *lxphd* respectively. The measurements report the runtime on 1 core, GPU, GPU plus 1 core, and show the percentage improvement of hMap using the CM2 cost model. The *hMap* Fibonacci has an improvement of 95% over the sequential time and improvement of 4% over the GPU time on *linux* lab and *lxphd* using CM2 , while the *hMap* matrix multiplication has an improvement of 68% over the sequential time on both *linux* lab and lxphd, and improvement of 32% on linux lab and 20% over the GPU time on *lxphd*.

Table 2: 1 Core *hMap* Runtimes (*linux lab*).

Data size	Run-Time (s)			1 Core+GPU Improvement%	
	1 Core	GPU	1 Core+GPU	1 Core	GPU
800x800	2.31	1.40	1.32	42%	5%
900x900	3.30	1.77	1.54	53%	12%
1000x1000	4.52	2.09	1.80	60%	13%
1100x1100	6.02	2.73	2.12	64%	22%
1200x1200	7.82	3.26	2.54	68%	22%
1300x1300	9.94	4.29	3.19	67%	25%
1400x1400	12.41	5.37	4.00	67%	25%
1500x1500	15.26	7.23	4.91	67%	32%

(a) matrix multiplication

Data size	Run-Time (s)			1 Core+GPU Improvement%	
	1 Core	GPU	1 Core+GPU	1 Core	GPU
1000	3.36	0.19	0.17	94%	10%
2000	6.77	0.34	0.32	95%	5%
5000	17.02	0.79	0.75	95%	5%
10000	34.17	1.53	1.47	95%	3%
20000	67.93	3.06	2.91	95%	4%
30000	103.30	4.55	4.39	95%	3%
40000	137.08	6.071	5.79	95%	4%
50000	170.80	7.55	7.24	95%	4%
60000	207.33	9.05	8.69	95%	4%
70000	243.79	10.51	10.12	95%	3%
80000	278.04	12.05	11.52	95%	4%

(b) Fibonacci

Table 3: 1 Core hMap Runtimes (lxphd).

Data size	Run-Time (s)			1 Core+GPU Improvement%	
	1 Core	GPU	1 Core+GPU	1 Core	GPU
800x800	4.28	1.47	1.41	67%	4%
900x900	6.09	1.84	1.66	72%	9%
1000x1000	8.37	2.25	1.98	76%	12%
1100x1100	11.12	2.88	2.36	78%	18%
1200x1200	14.43	3.49	3.07	78%	12%
1300x1300	18.34	4.46	3.90	78%	12%
1400x1400	22.91	5.79	4.88	78%	12%
1500x1500	28.25	7.61	6.02	78%	20%

(a) matrix multiplication

Data size	Run-Time (s)			1 Core+GPU Improvement%	
	1 Core	GPU	1 Core+GPU	1 Core	GPU
1000	3.27	0.20	0.19	94%	5%
2000	6.53	0.36	0.34	94%	5%
5000	16.36	0.79	0.77	95%	2%
10000	32.75	1.55	1.48	95%	4%
20000	65.47	3.07	2.93	95%	4%
30000	98.13	4.55	4.39	95%	3%
40000	130.97	6.07	5.77	95%	4%
50000	163.57	7.53	7.22	95%	4%
60000	196.44	9.06	8.67	95%	4%
70000	229.18	10.55	10.06	95%	4%
80000	261.77	12.00	11.52	95%	4%

(b) Fibonacci

Table 4 shows the runtime of matrix multiplication with data size of 1500 x 1500 and Fibonacci with data size 80,000 elements with a value of 1,000,000 using *hMap* on *lxpara*. The measurements show

that the *hMap* Fibonacci has improvement of 77% over 8 cores, while the *hMap* matrix multiplication shows that there is no improvement after 6 cores. The parallel performance is measured as the absolute speedup of using both the GPU device and the multiple cores within a single machine. Here the experiments have been carried out on *linux*, *lxphd*, and *lxpara* machines as single nodes.

Table 4: Multiple Core hMap Runtimes (lxpara).

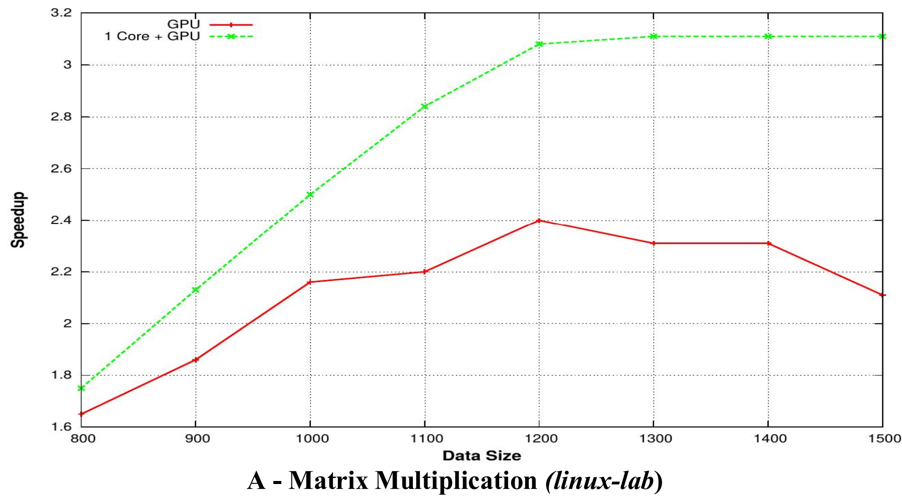
Data size	Run-Time (s)			1 Core+GPU Improvement%	
	Cores	GPU	(Core-1)+GPU	Cores	GPU
1	19.60	7.26	7.26	62%	0%
2	9.82	7.26	5.31	45%	26%
3	6.55	7.26	4.20	35%	42%
4	4.93	7.26	3.48	29%	52%
5	3.94	7.26	3.09	21%	57%
6	3.29	7.26	2.92	11%	59%
7	2.89	7.26	2.84	1%	60%
8	2.54	7.26	2.78	-9%	61%

(a) matrix multiplication

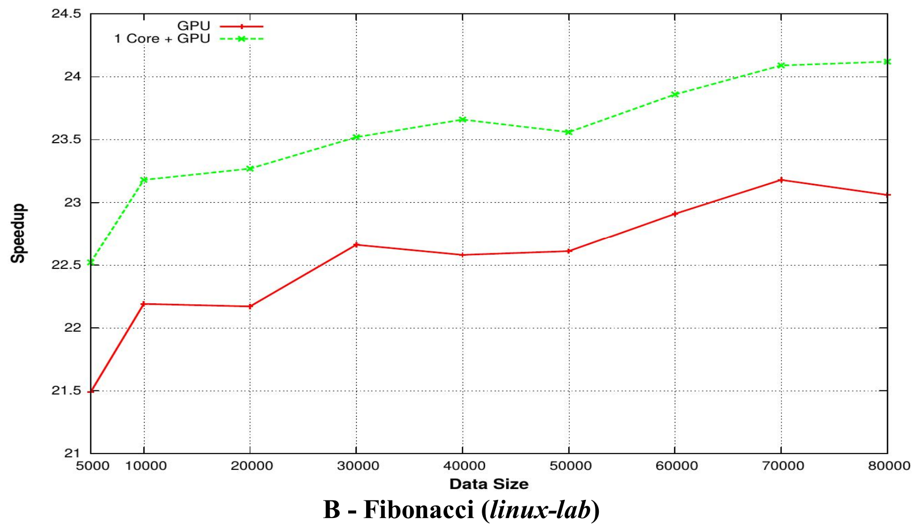
Data size	Run-Time (s)			1 Core+GPU Improvement%	
	Cores	GPU	(Core-1)+GPU	Cores	GPU
1	344.37	12.03	12.03	96%	0%
2	172.01	12.03	11.60	93%	3%
3	114.85	12.03	11.28	90%	6%
4	86.06	12.03	10.90	87%	9%
5	68.99	12.03	10.59	84%	11%
6	57.43	12.03	10.27	82%	14%
7	49.26	12.03	9.96	79%	17%
8	43.09	12.03	9.69	77%	19%

(b) Fibonacci

Figures 1 and 2 show the absolute speedup achieved for the Fibonacci and matrix multiplication programs with different input data sizes on the two-core *linux* and *lxphd* machines respectively. The graphs in Figures 1 and 2 compare the absolute speedup curve for one CPU-core plus single GPU implementation with the curve for GPU implementation.



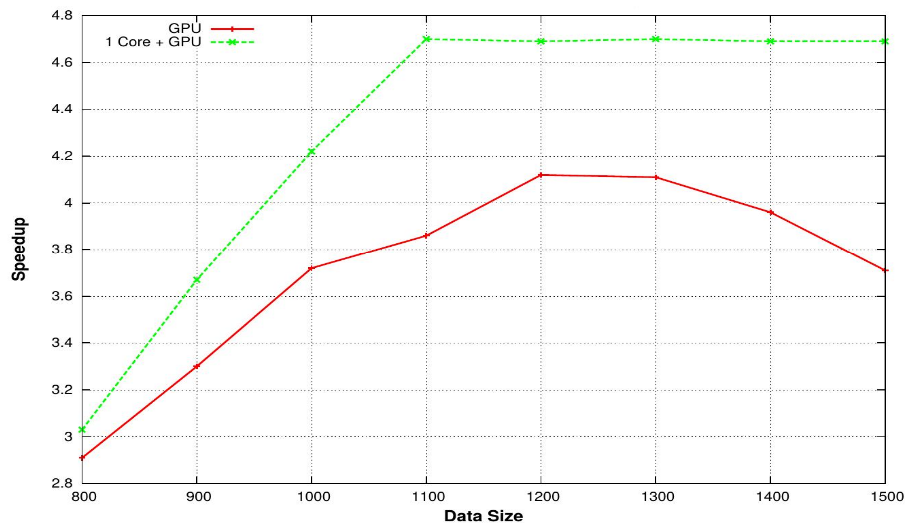
A - Matrix Multiplication *(linux-lab)*



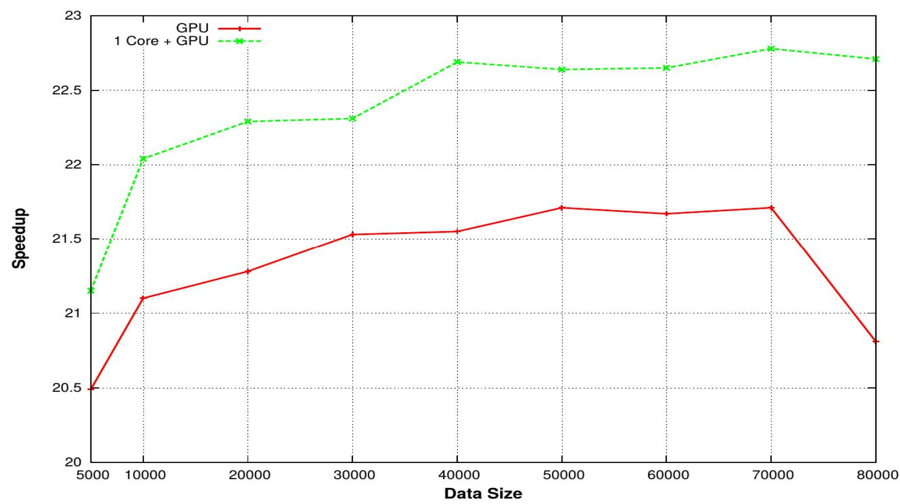
B - Fibonacci (*linux-lab*)

Figure 1: *hMap* Absolute Speedup on (*linux lab*)

Although the computing capability of GPU is relatively large compared with the computational strength of a single CPU-core, results show that using CPU-cores together with a GPU can deliver an expected and acceptable speedups on both machines.



A - Matrix Multiplication *(lxphd*)



B - Fibonacci (*lxphd*)

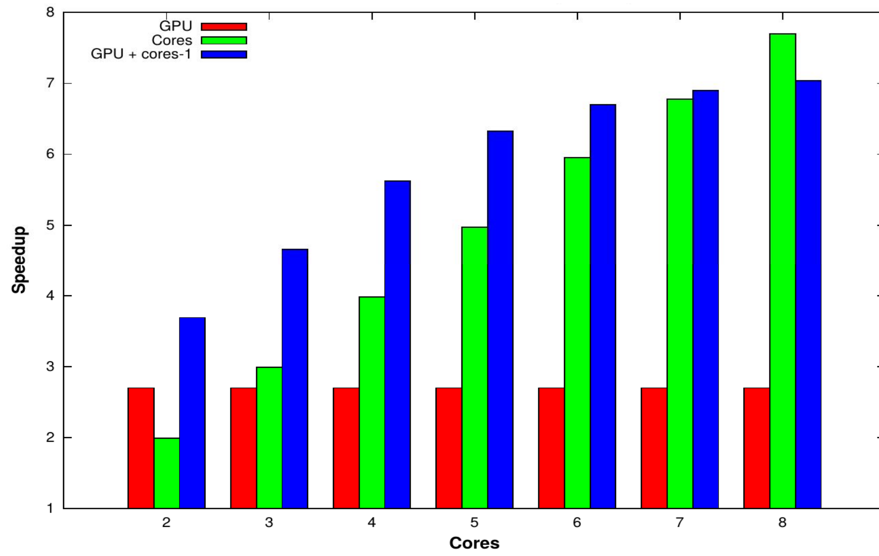
Figure 2: *hMap* Absolute Speedup on (*lxphd*)

Our results also suggest that using the performance cost model for determining granularity and data placement on different heterogeneous architectures can provide a good load balance for data distribution between CPU-cores and a GPU. This is reflected in the speedup graphs where the curves are broadly similar for both programs with different input data size on different parallel heterogeneous architectures. Next, to investigate the impact of the placement strategy on parallel performance of a varying number of CPU-cores with a single GPU, the experiments have been run with the Fibonacci and matrix multiplication programs on a machine with eight CPU-cores (*lxpara*). Figure 3 compares the absolute speedups of both Fibonacci and matrix multiplication programs on only CPU-cores and GPU, and CPU-core+GPU of the *lxpara* machine. This shows that in both programs a good performance has been obtained as anticipated.

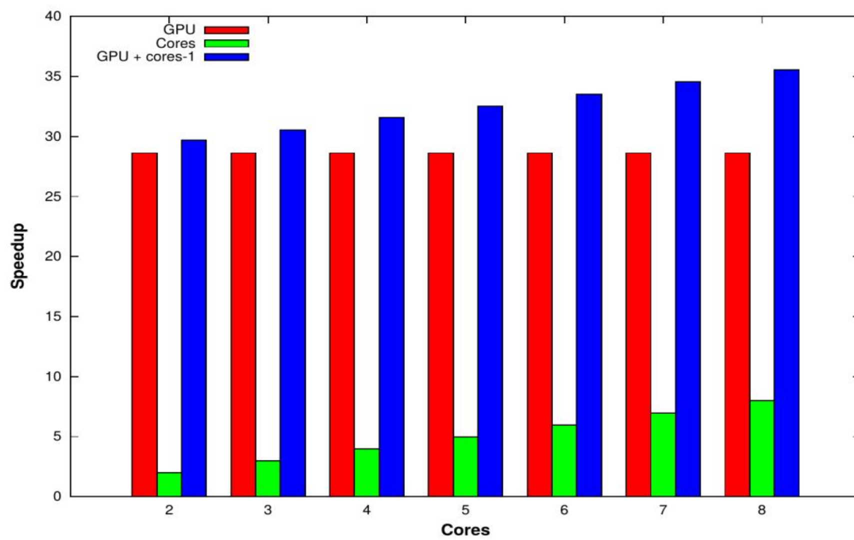
Firstly, the results presented in Figure 3 are consistent with others that obtained for both programs on the *linux* and *lxphd* machines, where the speedup is increased by using one CPU-core plus the GPU.

Secondly, we have obtained almost linear speedup with parallel efficiency of about 99% in both programs on CPU-cores. However, in the matrix multiplication program the speedup has a slight degradation to 95% parallel efficiency after six cores due to decreasing the chunk size. The results show that our skeleton delivers 28x from the GPU compared to a single CPU-core in the Fibonacci program, while we report nearly 2.8x speedup over a CPU-core by using a GPU in the matrix multiplication application. The variation in speedup between programs is due to the *GPU-HWSkel*-based parallel algorithm used for each program. Since the major problem with GPU implementations which affects the performance efficiency is the size of data being transferred between CPU and GPU, the algorithm requires too much data communication, which in turn increases the CPU/GPU communication overhead. Therefore, it is obvious that the algorithm for matrix multiplication is more suitable for multicore processors

than a GPU implementation, while the Fibonacci program makes a good GPU program.



A- Matrix Multiplication *(lxpara)*



B - Fibonacci (*lxpara*)

Figure 3: *hMap* Absolute Speedup on (*lxpara*)

4.3.2. Clusters of multicore/GPU Nodes Results

We evaluate the performance of an early version our cost model and its effect on our *hMap* heterogeneous skeletons on different combinations of the architectures outlined in Section 4.2. Figure 4 plots the speedups for different configurations with different processing elements calculating Fibonacci(1000000) 1500,000 times. The graph compares the speedups of three different kinds of computing units (i.e. CPU-cores, GPU-device, and GPU-device plus CPU-cores) on different numbers of given machines. Figure 4 shows that the results are consistent with those that were presented in Section 4.3.1. However, the performance of our *hMap* skeleton has been improved by exploiting the CPU-cores along with GPU in each host node. We suggest once again that our performance cost model has provided a good strategy of data placement for heterogeneous architectures. The graph shows that the implementation of our *hMap* skeleton can deliver good scalability, where the upper speedup curve shows improved performance results for using our cost model for data placement between the heterogeneous nodes as well as within each node between multiple cores and GPU.

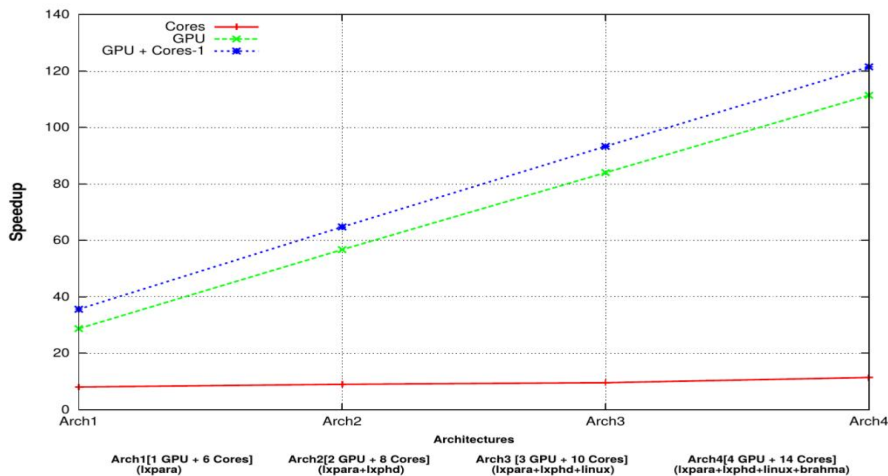


Figure 4: Speedups for the *hMap* on a Heterogeneous Cluster

5. Conclusions and Future Work

In this paper, a new performance cost model has been presented for heterogeneous integrated multicore/GPU systems. The purpose of the new cost model is to balance the workload distribution between the nodes on heterogeneous cluster as well as between multiple cores and GPU device inside each node in cluster. Our cost model is viewed as two-phase, the Single-Node phase guides workload distribution across a CPU core and a GPU using the performance ratio between the CPU and GPU in the integrated multicore/GPU computing node, and the Multi-Node phase balances the distribution of workload among the nodes on heterogeneous integrated multicore/GPU cluster. In general, we focus on predicting the runtime of the application code on the GPU and use an architectural performance cost model for measuring the processing strength of CPU to calculate the performance ratio. In summary, our experimental results show that using multiple cores together with a GPU in the same host with our skeleton and cost model can deliver good performance either on a single node or on multiple node architecture. Our work has a number of limitations, which we propose to address in future work:

- As noted above, our cost models do not take account of communication costs. We will explore how our simple notion of strength can be extended to account for communication characteristics.
- Our library, being based on CUDA, is NVIDIA specific. We will modify our library to use the OpenCL standard.

References

1. K. Armih. Toward Optimised Skeletons for Heterogeneous Parallel Architecture with Performance Cost Model. PhD thesis, Heriot Watt University, UK, 2013.
2. K. Armih, G. Michaelson, and P. Trinder. Cache size in a cost model for heterogeneous skeletons. In Proceedings of the fifth international workshop on High-level parallel programming

and applications, HLPP ’11, pages 3–10, New York, USA, 2011.

3. S. Gupta. Performance Analysis of GPU compared to Single-core and Multi-core CPU for Natural Language Applications. IJACSA-International Journal of Advanced Computer Science and applications, pages 50–53, 2011.
4. D. B. Kirk and W.-m. W. Hwu. Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.
5. C.-K. Luk, S. Hong, and H. Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, pages 45–55, New York, USA, 2009.
6. Y. Ogata, T. Endo, N. Maruyama, and S. Matsuoka. An efficient, model-based CPU-GPU heterogeneous FFT library. In Parallel and Distributed Processing. IEEE International Symposium on, 2008.
7. E. Wu and Y. Liu. Emerging technology about GPGPU. In IEEE Asia-Pacific Conference on Circuits and Systems, 2008.
8. C. Yang, F.Wang, Y. Du, J. Chen, J. Liu, H. Yi, and K. Lu. Adaptive optimization for petascale heterogeneous cpu/gpu computing. In Proceedings of the 2010 IEEE International Conference on Cluster Computing, CLUSTER ’10, pages 19–28, Washington, USA, 2010.