The Boat Hull Model: Enabling Performance Prediction for Parallel Computing Prior to Code Development

Cedric Nugteren Henk Corporaal
Eindhoven University of Technology, The Netherlands
http://parse.ele.tue.nl/
{c.nugteren, h.corporaal}@tue.nl

ABSTRACT

Multi-core and many-core were already major trends for the past six years and are expected to continue for the next decade. With these trends of parallel computing, it becomes increasingly difficult to decide on which processor to run a given application, mainly because the programming of these processors has become increasingly challenging.

In this work, we present a model to predict the performance of a given application on a multi-core or many-core processor. Since programming these processors can be challenging and time consuming, our model does not require source code to be available for the target processor. This is in contrast to existing performance prediction techniques such as mathematical models and simulators, which require code to be available and optimized for the target architecture.

To enable performance prediction prior to algorithm implementation, we classify algorithms using an existing *algorithm classification*. For each class, we create a specific instance of the *roofline model*, resulting in a new class-specific model. This new model, named the *boat hull model*, enables performance prediction and processor selection prior to the development of architecture specific code.

We demonstrate the boat hull model using GPUs and CPUs as target architectures. We show that performance is accurately predicted for an example real-life application.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures; C.4 [**Performance of Systems**]: Modeling Techniques

General Terms

Performance

Keywords

Parallel Computing, Performance Prediction, The Roofline Model, GPU, CPU

1. INTRODUCTION

For the past five decades, single-processor performance has shown an exponential growth, enabling technology to become pervasive and ubiquitous in our society. This exponential growth ended in 2004, limited by two aspects: 1) it became unfeasible to increase clock frequencies because of power dissipation problems, and 2), processor architecture improvements have seen a diminishing impact [9]. To re-enable performance growth, parallelism is exploited. Enabled by Moore's law, more processors per chip (i.e. multi-core) was already a major trend for the past six years and is expected to continue for the next decades [7]. While multi-core is expected to enable 100-core processors by 2020 [7], another trend (many-core) already yields more than 2000 cores per chip, enabled by using much simpler processing elements. An example of such a many-core processor is the Graphics Processing Unit (GPU).

These trends (multi-core and many-core) make processor selection increasingly challenging. Which processor to use for a given application set is far from trivial. It is simply not feasible anymore to port an application to all target candidate processors. This is caused by two factors. Firstly, the search space has expanded, as both multi-core and many-core processors co-exist in one system or even on a single chip [12]. Secondly, it has become increasingly challenging and time consuming to program such processors [12].

To solve this problem of processor selection, we argue that a performance prediction method which does not require code to be available is desirable. This is in contrast to existing performance prediction techniques such as mathematical models or simulators, which do require code to be available and optimized for a target processor.

In this work we present the *boat hull model.* We use an existing *algorithm classification* to classify algorithms. Then, for each class, we adapt the *roofline model* [21] to include class information. In this way, we generate multiple rooflines, each specific for an algorithm class. We present an example application to demonstrate the new model. Although we focus in this work on the domains of image processing and computer vision, we believe that the concepts of the boat hull model can be extended to other domains.

The remainder of this paper is organized as follows. First, in sections 2 and 3, we present related work and background information respectively. Following, in section 4, we introduce the boat hull model. In section 5, we evaluate the work by predicting the performance of an example real-life application with the new model. Finally, we discuss future work in section 6 and conclude in section 7.

2. RELATED WORK

In this work, we present a new method to perform performance prediction. In this section we therefore discuss related work on performance prediction techniques. Additionally, since we base our work on an algorithm classification, we discuss existing algorithm classification methods.

2.1 Performance prediction methods

Traditionally, performance prediction is achieved using analytical performance models or detailed hardware simulators. For many-core architectures such as the GPU, multiple detailed performance models (e.g. [2], [11] and [19]) and hardware simulators (e.g. [3]) exist. These performance prediction techniques are both based on detailed knowledge of the hardware architecture and on the presence of optimized code for the target architecture, both of which we assume not to be present in our work.

In recent work, performance was predicted for a Convey HC-1 processor using an idiom recognizer tool [6]. This tool analyzes reference source code to find *idioms* and predicts performance based on the presence of these idioms. Similar to the goals of our work, their tool does not require code to be available for the target architecture. However, in contrast to our work, their work does not target multi-core CPUs and GPUs, is based on limited algorithm classes (or: idioms), does not provide an insightful visual model, and does require the presence of reference code.

2.2 Algorithm classification methods

Many variations of algorithm classifications have been introduced as part of work on *algorithmic skeletons* [8], for example [4]. In [5], a survey of 10 different classifications is presented. They include on average 4 classes, with *divide and conquer*, *pipeline*, and *farm* being the most common among all 10 classifications. These types of classes distinguish algorithms at a coarse-grained level, while we use a much finer-grained classification in this work.

Classifications for other purposes exist, such as *dwarfs*, *computational patterns* [1], and *design patterns* [13]. These even less detailed classifications introduce classes as a scheme to capture solutions for recurring design problems in systematic ways. They are intended to be used in natural language rather than with automated tools.

Another existing algorithm classification intended to be used for algorithmic skeletons is the classification presented in [15]. This classification does distinguish algorithms at a fine granularity, and is therefore used in this work. An overview of this classification is presented in section 3.2.

3. BACKGROUND AND MOTIVATION

We base our work on the *roofline model* an on an existing algorithm classification, which we both introduce in this section as background information. Additionally, we motivate the approach taken in this work.

3.1 The roofline model

Performance prediction and bottleneck analysis are two topics that have become increasingly important for heterogeneous and parallel computing. The roofline model was introduced as an easy to understand performance model capable of identifying performance bottlenecks [21]. This model gives a rough performance estimate based on the assumption

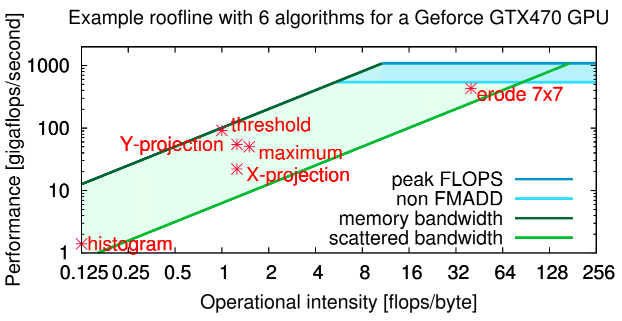


Figure 1: Applying the roofline model to an example GPU. The red stars represent the measured performance of example algorithms.

that performance is limited either by peak memory bandwidth or by peak ALU throughput. The roofline model is processor specific: for each processor there is a specific instance of the model.

In the roofline model, performance is measured in operations per second, which will either be memory bound or compute bound, dependent on an application's operational intensity (given in operations per byte). Because not every application will reach the peak performance (and thus the *roof* of the model), multiple *ceilings* can be added, denoted by properties such as limited instruction level parallelism or scattered memory accesses. These properties make the model suitable for bottleneck analysis and guidance during application development.

To illustrate the use of the roofline model, we map 6 example algorithms onto the roofline model for a GeForce GTX470 GPU. The results are shown in figure 1. The location of each algorithm is based on two aspects: 1) the performance of a CUDA implementation executed on the GPU, and 2), the operational intensity in ALU operations per off-chip load/store.

We observe two obstacles if we want to use this model to predict performance: 1) the execution time is not directly visible, and 2), the range of the predicted performance is very wide. For example, as shown in figure 1, the performance of the `X-projection` algorithm is a factor 7 beneath the memory bandwidth roof.

3.2 Algorithm classification

In [15], a well-defined fine-grained algorithm classification is introduced. The granularity of such a classification is of high importance for the applicability. When using the classification for performance prediction, a finer-grained classification might yield a more accurate prediction. On the other hand, if the classification is coarser-grained, it can be easier to use and to understand. The classification presented in [15] finds a solution to this trade-off by introducing a modular and parameterisable classification. This enables a fine-grained classification, while using a limited vocabulary.

We briefly illustrate the classification by giving four example code snippets and their corresponding classes. These examples, shown in listing 1, are classified as follows:

- In lines 1-2 a vector of size K is element-wise multiplied, incremented, and stored as another vector. Since every *element* of the input corresponds to an *element* of the output and the vector size is K, we classify this code snippet as 'K|element $\rightarrow$ K|element'.

- The for-loop in lines 4-5 performs a similar operation, but now also requires two *neighbours* to compute one output *element*. The classification becomes 'K|neighbourhood(3) → K|element', since the neighbourhood is of size 3 (including the element itself).
- Similar to the code snippet in lines 1-2, the code in lines 7-12 performs an *element* to *element* computation. However, in this case, we process two dimensional matrices of size 10 by 10. The code is therefore classified as '10x10|element → 10x10|element'.
- The final snippet (lines 14-15) processes the input per *element*, but stores the result in a *shared* output. It is therefore classified as 'K|element → 1|shared', with 1 being the size of the output.

The algorithm classification captures both the parallelism as well as the data access dependencies. Further details and more code examples can be found in [15].

```
 1  for(i=0; i<K; i=i+1)
 2    B[i] = 2 * A[i] + 5;
 3
 4  for(i=0; i<K; i=i+1)
 5    B[i] = 0.3*A[i-1] + 0.4*A[i] + 0.3*A[i+1];
 6
 7  for(a=0; a<10; a=a+1)
 8    for(b=0; b<10; b=b+1)
 9      value = A[a][b];
10      if (value > 255)
11        value = 255;
12      B[a][b] = value;
13
14  for(i=0; i<K; i=i+1)
15    B = B + A[i];
```

Listing 1: Four example code snippets of different algorithm classes.

4. THE BOAT HULL MODEL

Selecting which processor architecture is best suited for a given application can be done using architecture models or hardware simulators. These methods do however require the presence of optimized target architecture code, which is often not available before selecting a processor. Although not designed for this purpose, the roofline model does give an indication of the expected performance without requiring code, but falls short when an application's compute or data-access patterns are non-ideal. To enable performance prediction prior to algorithm implementation, we introduce a modified version of the roofline model based on the algorithm classification presented in [15].

The modified model, referred to as the boat hull model, makes the following changes to the roofline model in order to enable performance prediction:

- With use of the classification, the roofs and the ceilings of the roofline model can be fine-tuned to match the properties of a specific class. Because the amount of off-chip data accesses is inherent to a class-architecture combination, the metric on the horizontal axis of the model can be changed from 'operations per byte' into 'complexity': the number of operations given for a class' operator $f()$ (see [15]).
- Since application developers are primarily concerned about execution time, the metric on the vertical axis of the model is changed from 'flops per second' into 'execution time', as is also briefly mentioned in [20]. In combination with the change of the metric on the horizontal axis, we create an inverse view of the roofline model, resembling the cross section of a boat's hull.

Furthermore, for accelerators such as the GPU, data transfer time between the accelerator and a host processor can influence whether or not to run an application on such an accelerator. Therefore, the boat hull model is extended to include data transfer cost between different processors. Moreover, we can now enable performance prediction of a complete application by combining multiple computational parts of an application with host-accelerator data transfer.

In this section, we discuss the boat hull model. We first give two toy examples to illustrate the model. Following, we introduce and validate the boat hull model for both NVIDIA GPUs and Intel CPUs. We shortly discuss our corresponding tool, and finally evaluate the new model. In this section, we use the notation *primitive* to refer to a computational intensive part of an algorithm (i.e. *kernel*).

4.1 Code examples

To get an intuitive feel for the boat hull model, we discuss two toy examples. We take two out of the four examples from listing 1 and use an NVIDIA GPU as a target processor architecture. We set up two bounds for these code snippets: the memory bound and the compute bound. The largest of these will set the performance bound in terms of execution time. In these examples, we use the notation $P_{compute}$ to denote the theoretical peak architecture limitation for ALU performance in operations per second, and $P_{coalesced}$ and $P_{uncoalesced}$ to denote the practical peak memory performance in bytes per second, for respectively sequential and scattered memory accesses. We assume individual data elements to be 32-bit (4 byte) large in these examples.

- In lines 1-2, K elements are read and as many are written to background memory in a coalesced fashion. If we divide the amount of bytes accessed by the peak bandwidth, we obtain the execution time in the case that the snippet is memory bound: $\frac{2 \cdot K \cdot 4}{P_{coalesced}}$. To obtain the compute bound, we divide the amount of operations by the peak compute rate. Implementing every iteration of the loop as a thread, we perform 2 operations per thread, plus an offset to calculate among others the array index. With K iterations, this results in: $\frac{K \cdot (2+offset)}{P_{compute}}$.
- The example in lines 14-15 reads K elements coalesced, but writes the result uncoalesced. The memory bound becomes in this case: $\frac{K \cdot 4}{P_{coalesced}} + \frac{1 \cdot 4}{P_{uncoalesced}}$. The compute bound is similar to the snippet from lines 1-2, but now performs only 1 operation. On a GPU, a parallel reduction tree is used, which causes a certain overhead. This is taken into account using the *offset* variable: $\frac{K \cdot (1+offset)}{P_{compute}}$.

In order to determine whether the code snippet is compute or memory bound and to find the predicted execution time, we evaluate both equations and identify the one that gives us the highest execution time.

Table 1: Example classes and their corresponding parameters for a GPU boat hull model. The table lists for each class an example from the domain of image processing. The expressions for α_i and β_i for neighbourhood-based classes are left out for readability.

class	example primitive	w	m	o	d	c	u	floors
AxB|element $\rightarrow$ AxB|element	binarization	A·B	1	16	2·A·B	d	0	c_1
unordered AxB|element $\rightarrow$ AxB|element	xy-mirroring	A·B	1	16	2·A·B	d	0	c_1 and m_1
AxB|tile(1xB) $\rightarrow$ A|element	x-projection	A	B	$4 \cdot m$	A·B+A	d	0	c_1 and m_1
AxB|tile(UxV) $\rightarrow$ $\frac{A}{U}$x$\frac{B}{V}$|element	scale down	$\frac{A}{U} \cdot \frac{B}{V}$	U·V	$4 \cdot m$	2·A·B	d	0	c_1
AxB|tile(UxV) $\rightarrow$ AxB|tile(UxV)	2D-DCT	$\frac{A}{U} \cdot \frac{B}{V}$	U·V	$4 \cdot m$	2·A·B	A·B	A·B	c_1
AxB|element $\rightarrow$ A·UxB·V|tile(UxV)	enlarge	$\frac{A}{U} \cdot \frac{B}{V}$	U·V	$4 \cdot m$	2·A·B	d	0	c_1
AxB|neighbourhood(NxM) $\rightarrow$ AxB|element	2D-convolution	A·B	N·M	64	2·A·B	$d+\alpha_1$	β_1	c_1
AxB|neighbourhood(N) $\rightarrow$ AxB|element	1D-convolution	A·B	N	64	2·A·B	$d+\alpha_2$	β_2	c_1
AxB|element $\rightarrow$ 1|shared	sum	A·B	1	16	A·B+1	A·B	1	c_1
AxB|element $\rightarrow$ C|shared	histogram	A·B	1	64	A·B+N	N	A·B	c_1
AxB|element $\wedge$ AxB|element $\rightarrow$ AxB|element	differencing	A·B	1	32	3·A·B	d	0	c_1

4.2 The boat hull model for example classes

In this section, we introduce the boat hull model for 11 example classes, which we present in table 1. These classes are taken from [15] and are defined in more detail in [15]. To introduce the boat hull model, we take NVIDIA GPUs as example target processors. A similar approach can be taken for other processors.

Analogous to the roofline model, we define a compute equation (c_0) and a memory equation (m_0). While these equations represented theoretical roofs in the roofline model, they represent performance predictions in terms of execution time in the boat hull model. These equations additionally contain class-dependent variables (w, m, o, c, and u) in order to create a distinguished model per class. The equations further depend on the amount of operations performed on input data, defined as the complexity of the operator $f()$, which in turn is defined as part of the classification. For an NVIDIA GPU, we define these equations as:

$$c_0 = \frac{w \cdot (f_{complexity} \cdot m + o)}{P_{compute}} \qquad \text{(Compute equation)}$$

$$m_0 = \frac{c}{P_{coalesced}} + \frac{u}{P_{uncoalesced}} \qquad \text{(Memory equation)}$$

Similar to the roofline model, performance is either compute bound (by c_0) or memory bound (by m_0). Therefore, to find the execution time, we take the maximum value of these equations. The complexity of the operator ($f_{complexity}$) remains variable in these equations, while $P_{compute}$, $P_{coalesced}$ and $P_{uncoalesced}$ represent peak limits of the processor and are constant for a given GPU model. The class-dependent variables (w, m, o, c, and u) are set as shown in table 1 for a number of example classes and are defined as:

w: This represents the amount of parallel **work-units**. For a large number of classes, this is equal to the input data size. In the case of a GPU, w reflects the number of worker-threads which can be spawned and possibly be executed in parallel.

m: A **modifier** for the amount of computations per work-unit. This is used for example for tile and neighbourhood based computations, for which the operator $f()$ is applied multiple times per work-unit. For other classes, m equals 1.

o: The **offset** per work-unit. In the context of GPUs, this can be seen as the class-specific initialization per thread, such as index computation and pre-loading data into on-chip memory. The values for o are estimates obtained experimentally and are rounded to powers of two. The prediction is not very sensitive for this variable: for large values of $f_{complexity}$ o is insignificant (equation c_0) and for small values it is likely that equation c_0 is not used at all (memory bound). The offset is responsible for the curve in the compute equations, as seen later on in figures 2, 3 and 7.

d: The total amount of input plus output **data** to be accessed.

c: Amount of compulsory off-chip memory accesses in a **coalesced** manner (i.e. sequential).

u: The amount of compulsory off-chip memory accesses in an **uncoalesced** manner (i.e. scattered).

For the variables d, c and u, we consider only compulsory memory accesses. Any accesses to local memories in case of data re-use are not added to these variables.

Not all relevant characteristics can be captured by classes, some are still dependent on the operation $f()$ performed. Therefore, we add floors to the boat hull model (similar to ceilings in the roofline model). For a GPU, we define a compute floor (c_1) and a memory floor (m_1). These floors can limit performance, but only for a limited amount of classes. For the example classes, the floors applicable are shown in the last column of table 1. The floors are given as:

$$c_1 = \frac{c_0}{2} \qquad \text{(Non fused multiply add)}$$

$$m_1 = \frac{d}{P_{uncoalesced}} \qquad \text{(Scattered memory accesses)}$$

To obtain the execution time including host-accelerator data transfers, we have to accumulate the presented equations (c_x and m_x) with CPU-GPU data transfer cost (t_0). We assume a copy-in and a copy-out of all the input and output data over a data bus (e.g. PCI Express). The total data transfer time is dependent on the peak practical bandwidth of the bus (B_{bus}) and the amount of data transferred (the class variable d). The equation for t_0 is as follows:

$$t_0 = \frac{d}{B_{bus}} \qquad \text{(Host-accelerator data transfer)}$$

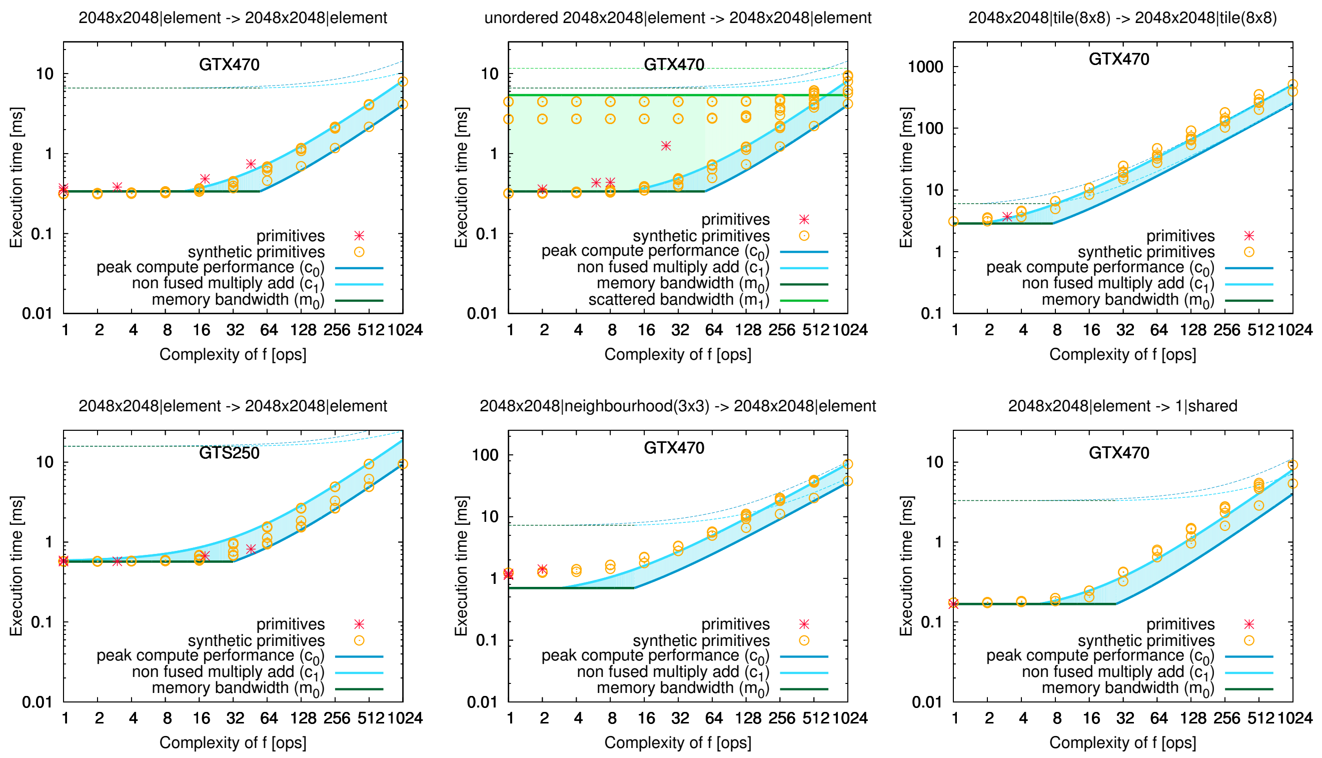


Figure 2: Six examples of the boat hull model applied to GPUs. We show five different classes taken from table 1 for the GTX470 GPU and show one example class for the GTS250 (bottom left). The model is validated against synthetic primitives (orange circles) and real-life examples (red stars). The dashed lines furthermore show the total predicted execution time, including host-accelerator data transfers.

4.3 Validating the boat hull model

To validate the boat hull model, we take primitives from the image processing domain as examples. We target two different GPUs from NVIDIA: a Geforce GTS250 and a GeForce GTX470. The characteristics of these two example GPU models are given in table 2. $P_{compute}$ is obtained from the product specification of the GPUs, while the bandwidth values are obtained using the `bandwidthTest` program supplied by the NVIDIA CUDA SDK [17].

Table 2: Characteristics of two example NVIDIA GPUs, further referred to as GTX470 and GTS250.

	GeForce GTX470	GeForce GTS250
compute capability	sm_20	sm_11
$P_{compute}$	1089 GFLOPS	470 GFLOPS
$P_{coalesced}$	95 GB/s	56 GB/s
$P_{uncoalesced}$	5.9 GB/s	3.5 GB/s
B_{bus}	5.1 GB/s	2.1 GB/s

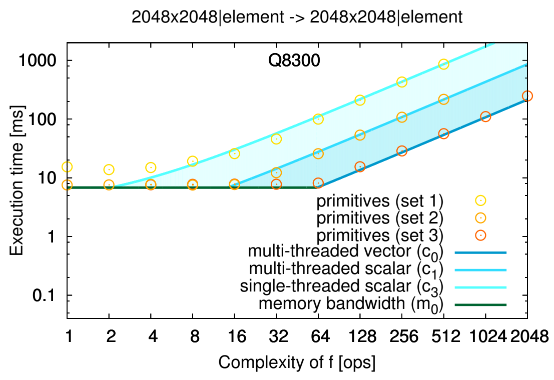
We show examples of the boat hull model in figure 2 for five different classes[1] taken from table 1. For the class '2048x2048|element → 2048x2048|element', we show a graph for both the GTX470 and the GTS250 GPU. For the other five classes, only the graph for the GTX470 GPU is shown. In figure 2, we show the predicted execution time based on the equations c_0, c_1, m_0 and m_1. To verify correctness of the model, we add a number of synthetic primitives (orange circles in figure 2) and real-life primitives (red stars in figure 2). The synthetic primitives are artificially constructed and vary in complexity, instruction mix (`add`, `mul`, `fma`) and instruction type (`int`, `float`). For classes with the 'unordered' prefix, they also vary in memory access pattern. The real-life primitives are taken from the image processing domain and contain primitives such as binarization, gamma correction, rotation, xy-mirroring, 2D-DCT, 2D-convolution, adaptive binarization, sum and dilation.

Furthermore, by accumulating c_0, c_1, m_0 and m_1 with the equation for the host-accelerator data transfers (t_0), we achieve a predicted total execution time, including CPU-GPU data transfer. This is given in figure 2 as dashed lines. The colour of the dashed lines is based on the same colour scheme as for the solid lines. Because data transfer is trivial to model and to improve the clarity of the graphs, we show no measurements for the total execution time in figure 2.

We briefly evaluate the results of figure 2 and draw the following conclusions:

- The predicted performance matches the measured performance closely: the measured execution time is either equal or slightly higher compared to the predicted value. The higher execution time can be attributed to the model's simplified view of the architecture.
- The boat hull model's quality is consistent between two different GPU models, even though they have different specifications and architectural properties (e.g.

[1]We assume elements to be 32-bit integers in these examples.



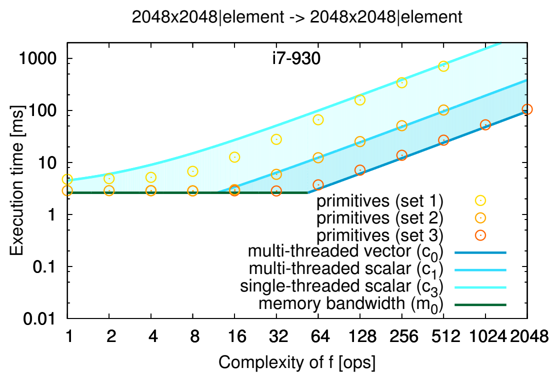


Figure 3: The boat hull model for an example class for both the Q8300 and the i7-930 CPUs. The model is validated against synthetic primitives, shown as circles in the graphs.

the GTX470 includes two levels of caches, which is not present in the GTS250 GPU).

- The additional memory ceiling causes the 'unordered 2048x2048|element → 2048x2048|element' class to give a wide prediction range. The synthetic primitives show that different memory access patterns indeed fit within this range, yielding a better performance when the fraction of coalesced memory accesses is larger.
- A large number of the tested non-synthetic primitives have a low complexity ($f_{complexity} < 10$) making them memory bound rather than compute bound.

4.4 The boat hull model for a multi-core CPU

So far, we have only demonstrated the use of the boat hull model for NVIDIA GPUs. To demonstrate the suitability of the model for a different type of processor, we apply the boat hull model in this section to Intel multi-core CPUs. Similar as for the GPUs, we take two CPU models with different specifications and a different micro-architecture. In this section, we use notations for the theoretical peak ALU performance ($P_{compute}$), the practical peak memory bandwidth (P_{memory}) measured using STREAM [14], the supported number of simultaneous threads ($N_{threads}$) and the width of the vector lane (W_{vector}). The characteristics of the two CPUs are given in table 3.

Table 3: Characteristics of two Intel multi-core CPUs. They are further referred to as Q8300 and i7-930.

	Core 2 Quad Q8300	Core i7-930
micro-architecture	Core	Nehalem
$P_{compute}$	40 GFLOPS	90 GFLOPS
P_{memory}	4.7 GB/s	12.2 GB/s
$N_{threads}$	4 threads	8 threads
W_{vector}	128 bits	128 bits

For the multi-core CPUs we re-use the compute and memory equations as given for the NVIDIA GPUs, but with a slight modification to the memory equation, since memory coalescing is not an issue. We also re-use the class dependent variables w, m, o and c, which have the same meaning as those given for GPUs. The equations are given as follows:

$$c_0 = \frac{w \cdot (f_{complexity} \cdot m + o)}{P_{compute}} \qquad \text{(Compute equation)}$$

$$m_0 = \frac{c}{P_{memory}} \qquad \text{(Memory equation)}$$

Compute performance on the CPU might be limited by single-threaded execution, scalar execution, or both. We therefore introduce three compute floors. Without using the vector extensions of the architecture, the peak compute performance decreases by the vector lane width (W_{vector}). Furthermore, single-threaded performance causes the peak computer performance to drop by a factor $N_{threads}$. The compute floors are therefore defined as follows:

$$c_1 = \frac{c_0}{W_{vector}} \qquad \text{(Multi-threaded scalar)}$$

$$c_2 = \frac{c_0}{N_{threads}} \qquad \text{(Single-threaded vector)}$$

$$c_3 = \frac{c_0}{W_{vector} \cdot N_{threads}} \qquad \text{(Single-threaded scalar)}$$

We furthermore identify instruction level parallelism (ILP) as another compute floor. To simplify results, we assume in this case that any loops are unrolled and ILP is maximized.

To illustrate the boat hull model for multi-core CPUs, we select the '2048x2048|element → 2048x2048|element' class as an example. We enable all floors for this class and set the class-specific variables as follows:

$$w = 2048 \cdot 2048 \qquad m = 1$$

$$o = 4 \qquad c = d = 2 \cdot 2048 \cdot 2048$$

For the example class, we run three sets of synthetic primitives on both architectures, while varying the complexity of $f()$ in terms of number of operations. Each set enables either: 1) single-threaded scalar execution, 2) multi-threaded scalar execution, or 3), multi threaded vector execution. The boat hull models along with the performance of the synthetic primitives are shown in figure 3 for the Q8300 and the i7-930 processors. We make the following observations with respect to the results:

- For both CPU models, we see a good match between the predicted and measured performance for the compute bound synthetic primitives. The multi-threaded vector operations match the peak compute rate of the architecture, while the performance for multi-threaded scalar operations is a factor 4 lower (assuming 32-bit data elements). Single-threaded operations perform a factor 4 or 8 lower, as predicted by the model.

- The multi-threaded synthetic primitives are able to achieve the predicted memory performance. The prediction is based on the OpenMP version of the benchmark STREAM [14], and does therefore not represent the theoretical peak memory bandwidth. Reasons for not reaching the theoretical peak bandwidth due to the complex memory sub-system of a CPU are discussed in more detail in [21].

4.5 The boat hull model tool

The boat hull model creates a different graph for each processor-class combination. Since classes contain parameters, a tool to automate the creation of such graphs can be very helpful. We present in this section a small tool for this purpose, which is available through our website[2].

Similar to the boat hull model, the tool takes as an input processor parameters and a class description. The processor parameters are as shown in tables 2 and 3. The class description is the full class name, as given in table 1. The class dependent variables w, m, o, c and u and the equations are set by the tool itself. Currently, the tool supports GPUs and CPUs for the 11 classes as shown in table 1.

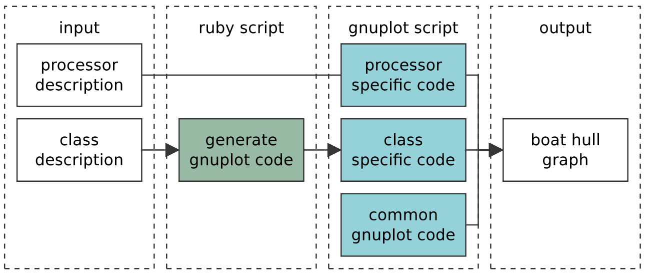


Figure 4: An overview of the tool corresponding to the boat hull model.

We give an overview of the tool in figure 4. Its main component is a script using the graphing utility *gnuplot*, which, after execution, generates a graph as shown in figures 2, 3 and 7. The script consists of three parts:

- A processor specific part, which is based on the processor description given as input to the tool.

- A class specific part, which is generated from a user supplied class description. The generator is written in the Ruby scripting language.

- A common part, which is processor and class independent. It contains the equations and the plot markup settings.

[2]http://parse.ele.tue.nl/

4.6 Evaluating the boat hull model

In this section, we evaluate the boat hull model. Since the model is solely based on class information and the primitive's operational intensity, we cannot expect a performance prediction comparable to detailed architectural models or simulators. With this in mind, we reflect on the boat hull model in this section.

We have seen in this work that the boat hull model's predicted performance is only roughly equivalent to the measured performance. This is due to the fact that many minor limitations are not taken into account. For example, the model does not take load balancing, cache behaviour and register pressure into account. Furthermore, the operator complexity ($f_{complexity}$) will at best yield an estimate due to aspects such as special instructions or compilation optimizations, both of which could alter the prediction. Because of these limitations, we presented the boat hull model as a technique to enable a rough performance prediction which can be used in an early design stage to determine whether or not to select a certain processor and whether or not to develop code for that processor. Once a processor is selected, a more precise performance prediction could be made with a detailed architectural model or simulator if necessary.

Although the presented technique does not rely on the availability of optimized code for a target processor, we do base the values for the class dependent variables w, m, o, c and u on the best available code implementations. The class dependent variables can simply be updated whenever faster code implementations are available. If the programmer eventually decides to develop code for the target architecture, the predicted performance will only be reached if the code is fully optimized for the target architecture.

Because the roofline model is intended to be used to help a programmer improve performance rather than to predict performance, it is of no surprise that the boat hull model gives a much tighter bound on execution time. This is due to the fact that the boat hull model creates multiple instances of the roofline model, each specific to a given class. Since algorithm classes embed information on parallelism as well as on data access dependencies, much more information is available to the boat hull model. This includes for example data re-use information and synchronization requirements.

If we compare the boat hull model to detailed mathematical models and simulators, we find that the boat hull model has the following advantages: 1) it is straightforward to extend to other or future processor architectures, 2) it requires very little architectural information (only four parameters for a GPU or CPU), and 3), most importantly, the boat hull model does not require code implementation nor code optimizations for the target architecture to be available. This allows for rapid performance estimation early in product design trajectories.

5. CASE-STUDY APPLICATION

To illustrate the boat hull model we evaluate a real-life computer vision application targeted at GPU acceleration. This particular application is selected because of its wide variety of (image processing) primitives. In the production process of organic LEDs, the centers of individual LEDs have to be identified under challenging throughput and latency requirements. As explained in [10], this can be achieved using the 3-stage *fast focus on structures* flow.

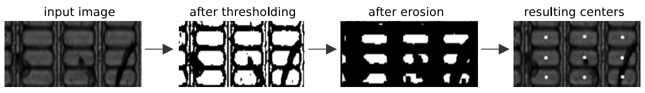


Figure 5: An example of the fast focus on structures application, which finds the centers of 9 LED structures.

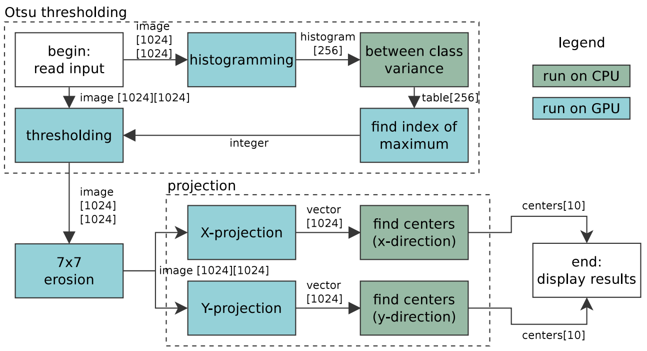


Figure 6: The case-study fast focus on structures flow, in which a 1024x1024 pixel input image is used and 10x10 LED structures are assumed to be present.

An example of the fast focus on structures application is shown in figure 5, in which the 3-stages are applied subsequently: Otsu thresholding, erosion, and projection. A flow chart is given in figure 6, in which the individual image processing primitives are shown. As shown in figure 6, six image processing primitives are targeted for acceleration, in this case using a GPU. More information on the fast focus on structures application is given in [10].

Table 4: Classification of the application's image processing primitives according to the algorithm classification.

primitive	classification
histogram	1024x1024|element → 256|shared
maximum	262144|element → 1|shared
threshold	1024x1024|element → 1024x1024|element
erode 7x7	1024x1024|neighb(7x7) → 1024x1024|element
X-projection	1024x1024|tile(1x1024) → 1024|element
Y-projection	1024x1024|tile(1024x1) → 1024|element

We classify the six primitives according to the algorithm classification presented in [15]. The results[3] are shown in table 4. The application is executed using both of the GPUs introduced in table 2: the GTX470 and the GTS250. For each primitive we generate a graph using the boat hull model tool. As input to the tool we supply the class names as shown in table 4 and the GPU specifications as given in table 2. The resulting graphs are shown in figure 7 for both the GTX470 (top) and the GTS250 (bottom). In these figures, we mark the measured performance with a red star symbol. We make the following observations with respect to the results as shown in figure 7:

[3]The problem size of the primitive `maximum` is artificially increased from 256 to 262144 elements, because the original problem size is too small to be measured accurately.

- The performance of the primitives `histogram`, `threshold` and `erode` is accurately predicted for both GPUs.
- The `maximum` primitive shows a higher execution time for both GPUs compared to the predicted time. Even though the problem size has been artificially increased, it is still too small to yield a high occupancy on the GPU and to reach its peak memory bandwidth. Although the prediction is a factor 3 lower compared to the measured performance, it hardly affects the absolute performance of the complete application due to the low execution time.
- Both the `X-projection` and `Y-projection` primitives suffer from similar inefficiencies because of their relatively small problem sizes. A wide prediction range is given for primitives such as `X-projection`, because memory accesses for this class might be uncoalesced.
- The results are consistent over both GPU models, except for the `X-projection` algorithm. In that case we see that the tighter constraints on memory coalescing for the GTS250 results in a significantly higher execution time. Nevertheless, the measured performance is still within the predicted range for both GPU models.

Furthermore, we evaluate the total execution time of the fast focus on structures application and compare it to the predicted performance. To do so, we assume that fused multiply add instructions do not occur, but make no assumptions on memory access patterns. We show the results in table 5, in which we separately show kernel execution time and CPU-GPU data transfer time.

Table 5: Predicted and measured execution time of the case-study application. The difference is calculated using the average of the predicted execution time.

GTX470	predicted	measured	difference
GPU kernels	1.66-2.32 ms	1.96 ms	2%
data transfers	0.95 ms	1.07 ms	11%
total	2.61-3.27 ms	3.03 ms	3%
GTS250	**predicted**	**measured**	**difference**
GPU kernels	3.21-4.24 ms	3.99 ms	7%
data transfers	1.90 ms	2.13 ms	11%
total	5.11-6.14 ms	6.12 ms	8%

From the results, we conclude that the measured performance of the kernels falls well within the predicted performance range for both GPUs. The range is due to the fact that we assume the fraction of coalesced memory accesses for the `X-projection` primitive to be unknown. Concerning the CPU-GPU data transfer time, we observe that the predicted values are too optimistic. This is attributed to the fact that transferring smaller amounts of data over the PCI-Express bus yields an increasingly higher overhead.

The roofline model is not designed to predict performance, but can still be used for performance prediction. We therefore compare it briefly with the boat hull model. To do so, we map the 6 primitives of the fast focus on structures application onto the roofline model. We compare the results (figure 1) for the GTX470 architecture with the boat hull model (top half of figure 7). We observe three major

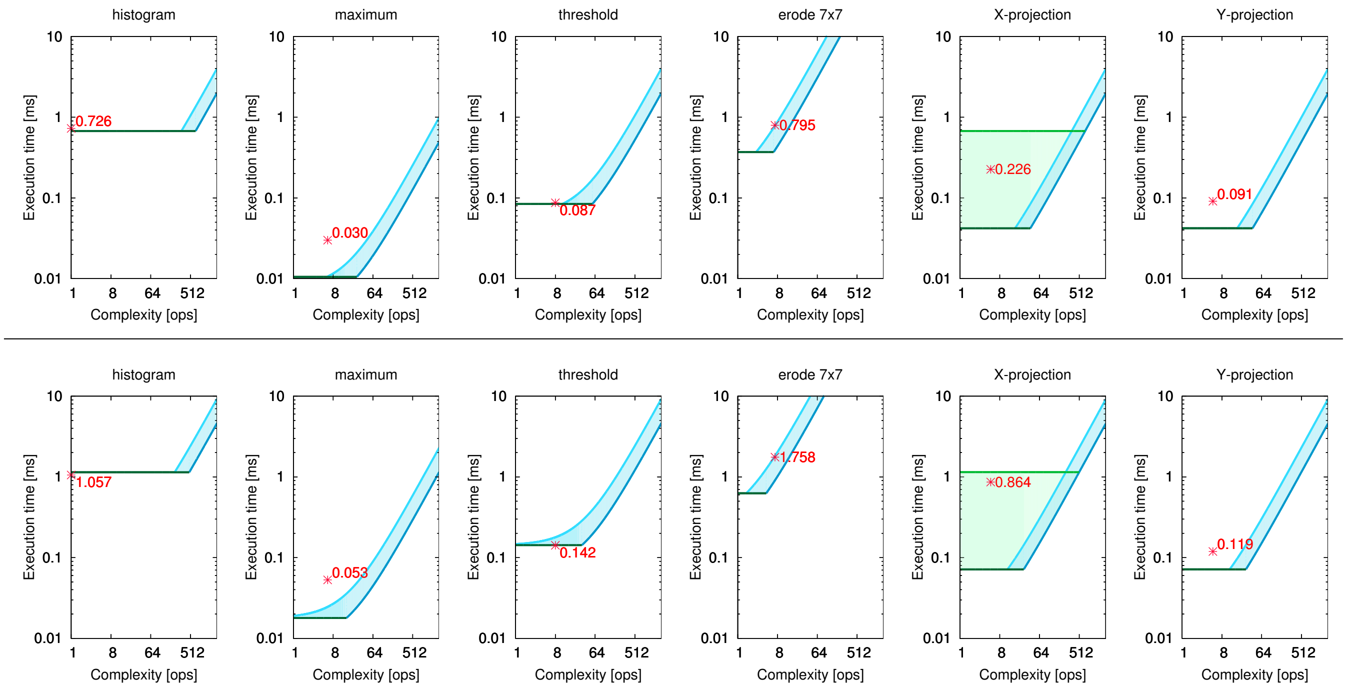


Figure 7: Applying the boat hull model to the fast focus on structures application for the GTX470 (top) and the GTS250 (bottom) GPUs. Red star symbols show the measured performance of a CUDA implementation running on these GPUs, while the curves show the predicted performance. The legend is as shown in figure 2.

differences. Firstly, the roofline model does not show the execution time directly. This can cause a problem when comparing two primitives, as one might have both a higher 'performance' and a larger execution time (e.g. `erode`). Not showing the execution time makes it also less intuitive to accumulate multiple primitives and/or data transfers for a total performance prediction. Secondly, we observe that the boat hull model gives a much tighter prediction. This is due to the creation of class-specific models, as explained in the paper. Lastly, we observe that the roofline model's x-axis requires the amount of loads and stores to off-chip memory, while the boat hull model embeds this information in the algorithm class. Determining which accesses go to off-chip or on-chip memory might not be a trivial task.

6. FUTURE WORK

The applicability of a technique such as the introduced boat hull model depends on the level of automation. In this work, we presented a tool to automatically generate a boat hull model graph. The generation of such a graph depends on the availability of class parameters (as for example given in table 1) and on architectural parameters. Class parameters are given for a number of example classes for GPUs and CPUs. Other, currently not supported classes, will be added in future work. Architectural parameters can be obtained from processor specifications and existing benchmark tools. Nevertheless, the identification of an algorithm class currently remains a manual effort. In separate work, we aim to identify classes automatically under the assumption that a basic C implementation is available.

Currently, the boat hull model predicts performance for a single primitive on a single (multi-threaded) processor. If an application such as fast focus on structures would be executed on a system consisting of multiple processors, kernel execution and data transfer might partly overlap. Future work aims to provide a methodology to support complete homogeneous and heterogeneous multi-processor systems.

We believe that the boat hull model can be a useful tool for system and application designers, provided that the algorithm classification can successfully classify a given application. Therefore, we aim to validate the boat hull model and the algorithm classification against a different domain in future work. For example using the PolyBench benchmark suite [18], which contains various linear algebra kernels and solvers.

Lastly, we work towards integrating the boat hull model as a run-time component into our skeleton-based source-to-source compiler 'Bones' [16]. This enables performance prediction at run-time based on the available hardware, which can be used for task scheduling and mapping onto a suitable processor.

7. CONCLUSION

In this work, we have introduced a method to estimate performance of applications on multi-core and many-core architectures prior to the implementation and optimization of target specific code.

Our new method of performance prediction is based on an existing algorithm classification and a modification to the *roofline model*. We modified the existing roofline model such that it generates multiple rooflines, each specific for a given algorithm class. This new model, the *boat hull model*, gives a prediction of an algorithm's execution time on a given processor based on the properties of the corresponding algorithm class. The boat hull model gives a much tighter bound on performance compared to the roofline model because of the integration of class specific information. In contrast to performance models and architecture simulators, the boat hull model gives a performance prediction prior to the implementation and optimization of target architecture specific code. Moreover, the presented model does not even require code to be available, pseudo-code or a high-level description can be sufficient to predict an application's performance.

In this paper we applied our methodology to different GPUs and CPUs. We evaluated the model for a number of synthetic benchmarks as well as for real-life examples, from which we have shown that the boat hull model is applicable in practice. We furthermore demonstrated the use of the new model for a case-study application. The application's algorithms were classified and their performance predicted using the boat hull model. We implemented a GPU accelerated version of the application and showed that our prediction for the complete application is within 8% of the measured performance.

8. REFERENCES

[1] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A View of the Parallel Computing Landscape. *Communications of the ACM*, 52:56–67, October 2009.

[2] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu. An Adaptive Performance Modeling Tool for GPU Architectures. In *PPoPP '10: 15th Symposium on Principles and Practice of Parallel Programming*. ACM, 2010.

[3] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing CUDA Workloads using a Detailed GPU Simulator. In *ISPASS '09: International Symposium on Performance Analysis of Systems and Software*. IEEE, 2009.

[4] W. Caarls, P. Jonker, and H. Corporaal. Algorithmic Skeletons for Stream Programming in Embedded Heterogeneous Parallel Image Processing Applications. In *IPDPS '06: 20th International Parallel and Distributed Processing Symposium*. IEEE, 2006.

[5] D. K. G. Campbell. Towards the Classification of Algorithmic Skeletons. Technical Report YCS 276, University of York, 1996.

[6] L. Carrington, M. M. Tikir, C. Olschanowsky, M. Laurenzano, J. Peraza, A. Snavely, and S. Poole. An Idiom-finding Tool for Increasing Productivity of Accelerators. In *ICS '11: International Conference on Supercomputing*. ACM, 2011.

[7] B. Catanzaro, A. Fox, K. Keutzer, D. Patterson, B.-Y. Su, M. Snir, K. Olukotun, P. Hanrahan, and H. Chafi. Ubiquitous Parallel Computing from Berkeley, Illinois, and Stanford. *IEEE Micro*, 30:41–55, March 2010.

[8] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1991.

[9] S. H. Fuller and L. I. Millett. Computing Performance: Game Over or Next Level? *IEEE Computer*, 44:31–38, January 2011.

[10] Y. He, Z. Ye, D. She, B. Mesman, and H. Corporaal. Feasibility Analysis of Ultra High Frame Rate Visual Servoing on FPGA and SIMD Processor. In *ACIVS '11: Advanced Concepts for Intelligent Vision Systems*. Springer Berlin, 2011.

[11] S. Hong and H. Kim. An Integrated GPU Power and Performance Model. In *ISCA '10: 37th Annual International Symposium on Computer Architecture*. ACM, 2010.

[12] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the Future of Parallel Computing. *IEEE Micro*, 31:7–17, September 2011.

[13] K. Keutzer and T. Mattson. A Design Pattern Language for Engineering (Parallel) Software. In *Intel Technology Journal*, 2010.

[14] J. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture Newsletter*, pages 19–25, December 1995.

[15] C. Nugteren and H. Corporaal. A Modular and Parameterisable Classification of Algorithms. Technical Report No. ESR-2011-02, Eindhoven University of Technology, 2011.

[16] C. Nugteren and H. Corporaal. Introducing 'Bones': A Parallelizing Source-to-Source Compiler Based on Algorithmic Skeletons. In *GPGPU-5: 5th Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 2012.

[17] NVIDIA. *CUDA C Programming Guide 4.0*, 2011.

[18] L.-N. Pouchet. PolyBench: The Polyhedral Benchmark Suite. http://www.cse.ohio-state.edu/~pouchet/software/polybench/.

[19] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc. A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications. In *PPoPP '12: 17th Symposium on Principles and Practice of Parallel Programming*. ACM, 2012.

[20] S. Williams. *Auto-tuning Performance on Multicore Computers*. PhD thesis, University of California, Berkeley, 2008.

[21] S. Williams, A. Waterman, and D. Patterson. Roofline: an Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52:65–76, April 2009.