Static Placement of Computation on Heterogeneous Devices*

GABRIEL POESIA, Universidade Federal de Minas Gerais, Brazil
BRENO GUIMARÃES, Universidade Federal de Minas Gerais, Brazil
FABRÍCIO FERRACIOLI, LG Electronics, Brazil
FERNANDO MAGNO QUINTÃO PEREIRA, Universidade Federal de Minas Gerais, Brazil

Heterogeneous architectures characterize today hardware ranging from super-computers to smartphones. However, in spite of this importance, programming such systems is still challenging. In particular, it is challenging to map computations to the different processors of a heterogeneous device. In this paper, we provide a static analysis that mitigates this problem. Our contributions are two-fold: first, we provide a semi-context-sensitive algorithm, which analyzes the program's call graph to determine the best processor for each calling context. This algorithm is parameterized by a cost model, which takes into consideration processor's characteristics and data transfer time. Second, we show how to use simulated annealing to calibrate this cost model for a given heterogeneous architecture. We have used our ideas to build Etino, a tool that annotates C programs with OpenACC or OpenMP 4.0 directives. Etino generates code for a CPU-GPU architecture without user intervention. Experiments on classic benchmarks reveal speedups of up to 75x. Moreover, our calibration process lets Etino avoid slowdowns of up to 720x which trivial parallelization approaches would yield.

CCS Concepts: • **Computer systems organization** → **Heterogeneous (hybrid) systems**; • **Computing methodologies** → *Randomized search*; • **Software and its engineering** → *Compilers*;

Additional Key Words and Phrases: Static Analysis, GPUs, Compiler, Heterogenous Architecture

ACM Reference Format:
Gabriel Poesia, Breno Guimarães, Fabrício Ferracioli, and Fernando Magno Quintão Pereira. 2017. Static Placement of Computation on Heterogeneous Devices. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 50 (October 2017), 28 pages. https://doi.org/10.1145/3133874

1 INTRODUCTION

Computing systems formed by processors that run at different speeds and that exchange data at a non-negligible cost have become common among the hardware used today [Cota et al. 2015; Zahran 2016]. Examples of such settings include general purpose computers equipped with graphics

*This work was partially supported by the project FAPEMIG-PRONEX-MASWeb, by FAPEMIG, CNPq, CAPES and LG. Gabriel Poesia was supported by a scholarship from Google, and later, by a scholarship from CNPq. We thank Márcio Pereira and Guido Araújo for generously granting us access to the Exynos board used in our Setup 3 (discussed in Section 5). We thank the OOPSLA referees for their time and expertise. Etino can be used on-line at http://cuda.dcc.ufmg.br/etino/.

Authors' addresses: Gabriel Poesia, Computer Science, Universidade Federal de Minas Gerais, Avenida Antônio Carlos, 6627, Belo Horizonte, MG, 31270-901, Brazil, gabriel.poesia@dcc.ufmg.br; Breno Guimarães, Computer Science, Universidade Federal de Minas Gerais, Avenida Antônio Carlos, 6627, Belo Horizonte, MG, 31270-901, Brazil, brenosfg@dcc.ufmg.br; Fabrício Ferracioli, SCA, LG Electronics, Avenida das Nações Unidas, 14171, São Paulo, SP, 04794-000, Brazil, fabricio.ferracioli@lge.com; Fernando Magno Quintão Pereira, Computer Science, Universidade Federal de Minas Gerais, Avenida Antônio Carlos, 6627, Belo Horizonte, MG, 31270-901, Brazil, fernando@dcc.ufmg.br.

2475-1421/2017/10-ART50
https://doi.org/10.1145/3133874

processing units (GPUs) [Nickolls and Kirk 2009], smartphones with a main CPU plus a separate digital signal processor [Satyanarayanan 2011], CPUs accelerated with Field-Programmable Gate Arrays (FPGAs) [Bacon et al. 2013] and distributed systems in which clients offload work to the clouds [Armbrust et al. 2010]. Henceforth, we shall call these systems *heterogeneous architectures*. The recent ability of manufacturers to combine a main computing unit with independent accelerators has made such configurations a staple in modern commodity hardware [Nickolls and Dally 2010].

The combination of different processing units into a single computational system gives application developers the capacity to produce more efficient software at a lower cost. Programmers can choose the best hardware to run each part of a complex application; however, it is not easy to benefit from this ability [Cao et al. 2012]. Finding a good match between computation and hardware involves several factors: the cost to transfer data between memory banks, the location of such data, the relative speed of processors, the affinity between program and execution model, etc [Augonnet et al. 2011; Barik et al. 2016; Diamos and Yalamanchili 2008; Wu et al. 2015]. As a consequence of this involved complexity, the design and implementation of high-performance programs that run on heterogeneous systems is still a task restricted to experts [Singh et al. 2013].

There exist today several tools, ranging from totally dynamic to fully static, that bring heterogeneous systems closer to ordinary programmers. However, we believe that they do not, yet, offer a satisfactory solution to the problem of mapping program parts to fundamentally different processors. For instance, on the completely dynamic side of this spectrum we have systems such as StarPU [Augonnet et al. 2011], WASH AMP [Jibaja et al. 2016] and Octopus-Man [Petrucci et al. 2015]. The former, being a library, requires programmers to change their code; the latter two bring an overhead high enough to make them prohibitive to resource constrained scenarios. On the other side of this spectrum, we have fully static, ahead-of-time compilers, such as dawncc [Mendonça et al. 2017, 2016], par4all [Amini et al. 2012; Guelton et al. 2012], ppcg [Verdoolaege et al. 2013] and Bones [Nugteren and Corporaal 2014]. The first tool, dawncc, offloads to the accelerator every program part that it can annotate with either OpenACC or OpenMP 4.0 directives. The other three leave this task to the developer. Both approaches meet with shortcomings. On the one hand, as we show in Section 5, unrestricted offloading often leads to slowdowns in programs that do not sport sufficient parallelism. On the other hand, leaving this task to programmers forces them to understand the code that must be compiled, in addition to the architecture at hand. Furthermore, this approach is susceptible to all the imprecisions of a human discernment. In this paper, we address all these limitations.

The goal of this paper is to reduce this gap between programmers and heterogeneous computing architectures. To this end, we have designed and implemented a static program analysis that matches hardware and computation. We describe it in Section 3. Our algorithm is *partially* context-sensitive. By "partially", we mean that it runs on a data structure that encodes part of the information available in the program's calling contexts. We call such a data structure the *Extended Call Graph*. Our algorithm can be parameterized by a cost model; thus, it achieves performance portability across different architectures. To free developers from the burden of calibrating cost models, we resort to autotuning, a tendency that has gained momentum recently [Basu et al. 2013]. Along this direction, Section 4 shows how to use simulated annealing to adjust the parameters of a cost model for a heterogeneous system.

We have used the proposed algorithm to implement a tool, Etino[1], which generates code for CPU-GPU based systems. Etino is fully automatic. To shield developers from minutia related to

[1]The name Etino comes from the initials C_2H_2, which, in our context, stand for "Colocação de Computação em Hardware Heterogêneo" – Portuguese for *Placement of Computation in Heterogeneous Hardware*. C_2H_2 is the formula of *Acetylene*, which is called Etino in Portuguese (IUPAC version of *Acetileno*). We thank Rafaela Salgado for pointing that out to us.

the computing architecture, Etino combines different techniques to: (i) find parallel regions in C code [Wolfe 1996]; (ii) estimate the nesting depth of computations [Gulavani and Gulwani 2008]; (iii) estimate the size of the arrays to be transferred between host and device [Rugina and Rinard 2005]; and (iv) infer the amount of SIMD divergent code [Sampaio et al. 2014] within a target procedure. These analyses are implemented using the LLVM compiler [Lattner and Adve 2004]. To generate code, Etino clones functions, and uses an automatic source-to-source compiler, dawncc [Mendonça et al. 2017], to insert either OpenACC [openacc.org 2012] or OpenMP 4.0 [Bertolli et al. 2014] directives in these clones.

Section 5 describes an experimental evaluation of Etino. We have tested it in three different architectures. Each one is formed by a CPU and a GPU, giving us six different processors. To perform the final translation of C programs annotated with OpenACC directives to CUDA, we have used pgcc [PGI 2016]. To handle OpenMP 4.0, we have used clang, with a few extensions of our own craft, necessary to handle features that are currently missing in that compiler (Section 5.2). We also show how to use Etino in combination with ppcg [Verdoolaege et al. 2013] to produce optimized CUDA code (Section 5.3). We have analyzed and run programs taken from four benchmarks: PolyBench, The Computer Language Benchmarks Game, dawncc's test suite and DataMining [Da Mata et al. 2013]. These benchmarks let us show that we can deliver speedups of almost 190x on embarrassingly parallel programs. Furthermore, Etino prevents slowdowns that unrestricted parallelization, performed by state-of-the-art tools such as dawncc or ppcg, could produce. Finally, our techniques can also be applied on irregular programs not usually seen in parallelization research (Section 5.5). All the results that we produce have been obtained without the intervention of developers, except for DataMining. In this case, we had to change sparse data structures, e.g., linked lists, into arrays, so that our tool could infer the memory regions to be transferred to the GPU. Etino is a complex tool, built over decades of developments in the field of static analyses. We do not claim its engineering as a novel contribution, as it relies on technology already described in the literature. However, Etino's essential idea: the code placement algorithm that we describe in Section 3, plus the technique to calibrate it, that appears in Section 4, are original contributions of this paper, which we summarize as follows:

Algorithm: Section 3 describes the first fully static scheduling algorithm for heterogeneous architectures. Our technique is totally compiler-based and fully automatic: it does not require support from a library, operating system, middleware or hardware. This algorithm has been designed to account for data location, when deciding where to map program parts. It also handles recursion naturally. Because data location is a dynamic notion, our algorithm is context-sensitive: if function f, running on processor p, calls g, then g has greater incentive to run on p. For practical reasons, we restrict it to contexts of size 2, i.e., like a 2-CFA analysis [Shivers 1988].

Calibration: code placement is heavily dependent on the underlying heterogeneous computer architecture. Therefore, our first contribution, the algorithm, should produce different mappings, depending on the available hardware. To deal with this requirement, Section 4 describes an approach based on simulated annealing to calibrate a cost model for the static-scheduling algorithm. Therefore, we free the programmer from a tedious and time consuming task. Calibration also does not require any form of user intervention to converge. To demonstrate this fact, we have tested it in three architectures.

2 OVERVIEW

We shall use the program of Figure 1 to illustrate the static scheduler that we introduce in this paper. The program in this example evaluates the matrix expression $A + B \times C + D$. Function add_cpu performs matrix addition, and function mul_cpu performs matrix multiplication. The procedure

```
  void in(float* dst, int N);
  void out(float* dst, int N);
  // dst = A + B
1 void add_cpu(float*dst, float*A, float*B, int N) {
2   int i, j;
3   for (i = 0; i < N; i++)
4     for (j = 0; j < N; j++)
5       dst[i*N+j] = A[i*N+j] + B[i*N+j];
6 }
  // dst = A * B
1 void mul_cpu(float*dst, float*A, float*B, int N) {
2   int i, j, k;
3   for (i = 0; i < N; i++)
4     for (j = 0; j < N; j++) {
5       dst[i*N+j] = .0;
6       for (k = 0; k < N; k++)
7         dst[i*N+j] += A[i*N+k] * B[k*N+j];
8     }
9 }
```

```
  // m1 = m1 * m2 + m3
1 void mad_cpu(float*m1, float*m2, float*m3, int N) {
2   float* aux = (float*)malloc(N*N*sizeof(float));
3   mul_cpu(aux, m1, m2, N);
4   add_cpu(m1, aux, m3, N); free(aux);
5 }
1 int main() {
2   int N = 10000;
3   float* X = (float*)malloc(N*N*sizeof(float));
4   float A[N*N], B[N*N], C[N*N], D[N*N];
5   in(A, N); in(B, N); in(C, N); in(D, N);
6   add_cpu(X, A, B, N);
7   mad_cpu(X, C, D, N);
8   out(X, N);
9   free(X);
10  return 0;
11 }
```

Fig. 1. This program shows that a static scheduler must take calling context into consideration.

mad_cpu uses these two functions to execute a multiply-add operation on matrices. Both, add_cpu and mul_cpu, are embarrassingly parallel routines. The former can be executed in $O(1)$ in a PRAM machine; the latter in $O(\ln N)$ [Gibbons 1988].

Given this observation, it is tempting to execute these two functions in a GPU. Such task can be accomplished in several ways. The most straightforward alternative is to recode both procedures in a language that can be compiled to the GPU, such as CUDA [Garland 2008; Kirk 2007], OpenCl [Stone et al. 2010], or PyCUDA [Klöckner et al. 2012]. Another approach would be to use a compiler that translates C code into GPU binaries, such as Baskaran *et al.*'s [Baskaran et al. 2010]. Finally, a developer can use an annotation system like OpenMP 4.0, OpenACC [openacc.org 2012] or OpenSs [Ayguadé et al. 2009], to indicate which parts of the program in Figure 1 should run on the GPU. Nevertheless, regardless of the chosen approach, the programmer must decide which functions run on the GPU or on the CPU.

Not every embarrassingly parallel code benefits from the computational power of a graphics processing unit. As an example, we consider add_cpu in Figure 1. A high-end GPU is likely to process this algorithm faster than a modern CPU; however, once we include the time to transfer the data between host and device, using the GPU may no longer be profitable. Quoting Nvidia's Best Practice Guide, "*The complexity of operations should justify the cost of moving data to and from the device*"[2]. If we assume that add_cpu manipulates square matrices of $N \times N$ cells, then we must transfer $3N^2$ elements to perform N^2 operations – a ratio of operations to elements of 1:3. On the other hand, if we consider the procedure mul_cpu, we still transfer $3N^2$ elements, but perform N^3 multiply-adds. Thus, we have $O(N)$ operations per elements, which gives us a much more efficient use of the accelerator.

The time to move data is of less consequence if said data has already been transferred. Although an obvious observation, checking the location of data – statically – is not trivial. The first call of function add_cpu, (Figure 1, main:6) reads data in the memory of the processor where main_cpu has been invoked. The second call of add_cpu (Figure 1, mad_cpu:4) will read data located at wherever the calling instance of mul_cpu, which comes immediately before, has been invoked, since the latter produces data read by the former. This observation is true due to the caller-callee relations between functions. To account for these relations, the algorithm described in Section 3 considers the *calling context* of functions. The calling context of a given function invocation is the

[2] http://docs.nvidia.com/cuda/cuda-c-best-practices-guide

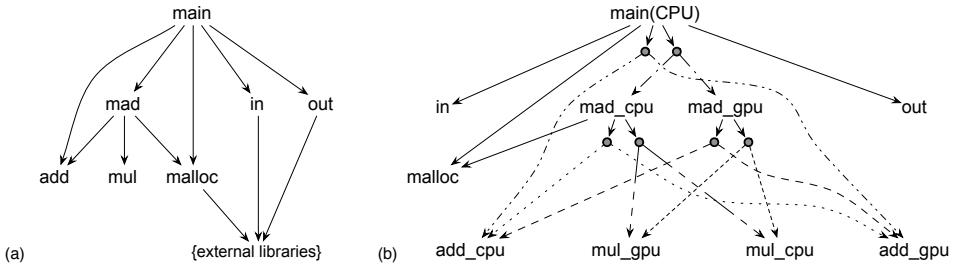

Fig. 2. (a) Call graph representing the program in Figure 1. (b) Extended call graph of the same program

```
1  mul_gpu(float* dst, float* A, float* B, int N) {
2    int i, j, k;
3    char RST_AI1 = 0; int N0 = N * N;
4    RST_AI1 |= !((dst > A + N0) || (A > dst + N0));
5    RST_AI1 |= !((dst > B + N0) || (B > dst + N0));
6    RST_AI1 |= !((A > B + N0) || (B > A + N0));
7    #pragma acc data pcopyin(A[0:N0],B[0:N0])
8                    pcopyout(dst[0:N0]) if(!RST_AI1)
9    #pragma acc kernels if(!RST_AI1)
10   for (i = 0; i < N; i++) {
11     for (j = 0; j < N; j++) {
12       dst[i*N+j] = .0;
13       for (k = 0; k < N; k++)
14         dst[i*N+j] += A[i*N+k] * B[k*N+j];
15     }
16   }
17   }}}
18 }
```

```
1  add_gpu(float *dst, float *m1, float *m2, int N) {
2    int i, j;
3    char RST_AI1 = 0; int N0 = N * N;
4    RST_AI1 |= !((dst > m1 + N0) || (m1 > dst + N0));
5    RST_AI1 |= !((dst > m2 + N0) || (m2 > dst + N0));
6    RST_AI1 |= !((m1 > m2 + N0) || (m2 > m1 + N0));
7    #pragma acc data pcopyin(m1[0:N0],m2[0:N0])
8                    pcopyout(dst[0:N0]) if(!RST_AI1)
9    #pragma acc kernels if(!RST_AI1)
10   for (i = 0; i < N; i++)
11     for (j = 0; j < N; j++)
12       dst[i*N+j] = m1[i*N+j] + m2[i*N+j];
13   }}}
14 }
1  void mad(float* m1, float* m2, float* m3, int N) {
2    float* aux = (float*)malloc(N*N*sizeof(float));
3    mul_gpu(aux, m1, m2, N);
4    add_gpu(m1, aux, m3, N);
5  }
```

Fig. 3. Code that we generate for the example in Fig. 1.

sequence of calls performed until the invocation occurs. In the example of Figure 1, add_cpu can be called from two different contexts: (i) main → add_cpu; and (ii) main → mad_cpu → add_cpu.

We summarize calling contexts via a data-structure – the Extended Call Graph (ECG). This graph contains clones of functions that can be called from different processors, and tracks caller-callee relations between them. We have one clone of each function per each processor where that routine can run. Figure 2 (a) shows the call graph of our running example, and Figure 2 (b) shows its ECG. Some functions have two clones: one – the original– runs on the CPU, the other on the GPU. Heterogeneous architectures with a wider variety of processors would give us more clones. Edges with the same line pattern indicate a decision that our algorithm must make: only one of them will remain in the graph after scheduling is done. For instance, function main has the option of calling function add_cpu either on the CPU, as originally done, or on the GPU (via add_gpu). Exactly one of these function calls must be chosen.

The Extended call graph gives us a data structure onto which we can apply a *cost model*. This model has two key parameters: data transfer time, and processor affinity. The former estimates the cost to copy data between devices; the latter estimates the profit to run a certain computation on a particular device. For reasons that will be made clear in Section 3, once we apply a CPU-GPU cost-model onto the graph of Figure 2 (b), we find that the first call of add[3], at context `main : 9`, should run on the CPU. The same model indicates that the second call, at context `main : 10` → `mad : 4`, should run on the GPU. This difference happens because function mad, the last link in the second

[3]Henceforth, when talking about routines regardless of where they run, we shall omit suffixes such as _cpu or _gpu

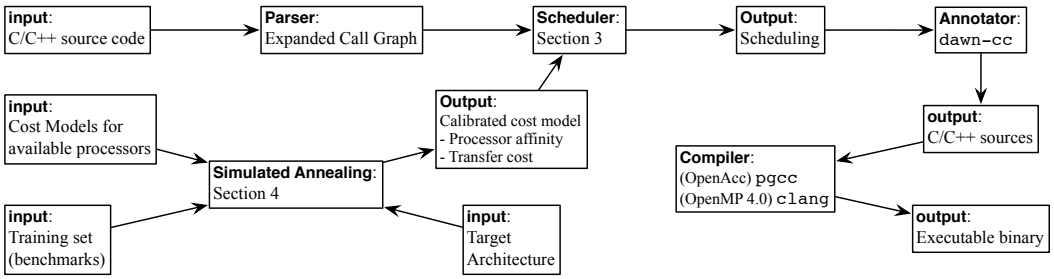

Fig. 4. Overview of our static scheduler.

context, is set to run on the GPU due to its asymptotic complexity. These results lead us to produce two versions of function add, one for the CPU, another for the GPU.

Figure 3 shows the code that we generate for this example. Currently, we annotate loops that must be sent to the graphics processing unit with either OpenACC or OpenMP 4.0 directives. These directives are inserted without user intervention, by a source-to-source compiler known as dawncc [Mendonça et al. 2017]. In this example, we end up with two versions of add, the original, called at `main : 9` (Figure 1), and a new procedure, which is invoked in the context `main : 10` → `mad : 4`. The original mul_cpu routine will no longer be necessary in this program; hence, it can be removed from the executable binary that we produce. If we run this program on a 12-core Intel Xeon CPU E5-2620 at 2.00GHz, with 16GB of memory, using $N = 3,000$, then it takes 656.0 seconds to finish[4]. If we map add and mul to an Nvidia GTX 670, then our example takes 3.52 seconds to finish. Finally, if we leave the first call of add onto the CPU, and send the second to the GPU, then we gain a slight edge, finishing in 3.30 seconds. To give the reader some perspective on these numbers, we have re-written the program to use some highly-tuned BLAS (short for Basic Linear Algebra Subprograms) libraries. In this experiment, we are using ATLAS[5], we run the re-written version of Figure 1 in 1.80 seconds using the 12 Intel CPUs. If we move to cuBLAS, then we shrink this number yet a bit, reaching 1.50 seconds on the Nvidia GPU. This example lets us show that we can bring a trivial, naïve program close to a state-of-the-art implementation without programmer's intervention.

Figure 4 shows the pipeline of this work. Section 3 presents our first contribution, a static scheduler that allocates functions to processors. This algorithm reads a source code, written in C, plus cost models for all kinds of processors available. Cost models are algorithms that have architecture-dependent parameters to estimate how costly a certain computation is. These parameters can be tuned automatically via some probabilistic search method. Section 4 presents the autotuning procedure we propose, which is executed only once for a given architecture, and is based on simulated annealing. Once we obtain a scheduling, programmers can either insert annotations manually in the code, or resort to an automatic annotator, as we do in this paper. Finally, the annotated program is given to an OpenACC or OpenMP 4.0 compliant compiler, which produces the final executable. Since the produced annotations might schedule some parts of the program to run on the GPU, such compiler will typically have an intermediate step that transforms chosen functions into CUDA or OpenCL kernels.

[4]We have compiled the original program in Figure 1, and the program that we produce automatically, with `pgcc -O2 -Munroll=c:1 -Mnoframe -Mlre -Mautoinline -Mvect=sse -Mcache_align -Mflushz -Mpre`.
[5]ATLAS is available at http://math-atlas.sourceforge.net/. We enabled it with `pgcc -lblas -fast`.

3 CONTEXT-AWARE STATIC SCHEDULING

This section introduces our notion of an Extended Call Graph (ECG), and defines the static scheduling problem (SSP). Given an ECG augmented with a cost model, the dynamic programming algorithm of Section 3.4 lets us find an optimal solution to this problem.

3.1 The Extended Call Graph

As we have explained in Section 2, to find good matches between code and computation, we resort to a data structure that we have named the Extended Call Graph (ECG). Its definition follows from the notion of a *Call Graph*. Call graphs are a well-known data-structure, but for the sake of completeness we define them again:

Definition 3.1 (Call Graph (CG)). Given a program P, its call graph CG_P is formed as follows: each function f in P gives origin to a vertex v_f. If function f calls function g, then CG_P contains an edge $v_f \rightarrow v_g$.

Example 3.2. Figure 2 (a) shows the call graph of the example in Figure 1. Each vertex represents one function, and each function in the program is represented by a unique vertex. We use one edge to represent multiple calls between functions. Thus, although main calls function in four times (line 5 of Figure 1), the call graph contains one edge from main to in.

The Extended Call Graph of a program is a version of its call graph that we use to implement our static scheduler, where nodes include context information. We define it as follows:

Definition 3.3 (Extended Call Graph (ECG)). Given a program $\mathcal{P}$, plus n processors $p_1, \ldots, p_n$, its Extended Call Graph is an expansion of CG_P, produced as follows. For each function f that can run on a processor p_i, we produce a node $v_{i/f}$. If CG_P contains an edge $v_f \rightarrow v_g$, then ECG_P contains an edge $v_{i/f} \rightarrow v_{j/g}$, for each possible i and j.

Example 3.4. Figure 2 (b) shows an extended call graph. Functions add, mad and mul can be called on the GPU or on the CPU; hence, they generate two vertices each in the ECG.

The extended call graph gives us some context information about a program, albeit in a very limited way[6]. Traditionally, a calling context is the sequence of function activations. In our case, we shall call a context an invocation of a function f at a processor p. This arrangement corresponds to keeping track of contexts of depth 1, e.g., the context of f will let us know the processor running the last function active when f was invoked. Each pair formed by a function f and a processor p_i, if f may run on p_i, leads to the creation of a new vertex $v_{i/f}$. For each function call $f \rightarrow g$ in the program's call graph, we'll create $n_f \times n_g$ edges in the extended call graph, where n_x is the number of processors that can execute function x. Because we do not keep track of calling sites within the same function, if g is called at two different points within f, we still have only one edge $v_{i/f} \rightarrow v_{j/g}$ in the extended call graph.

Discussion: shallow contexts. Our representation of context is shallow: we do not distinguish different invocation sites of the same callee within a caller. This decision is a tradeoff between precision and pragmatism. It is common wisdom that context sensitive analyses are expensive [Lhoták and Hendren 2006; Whaley and Lam 2004]. The consequence of our decision is that if a given function g is scheduled to run on processor p when called from f, then every call of g from f shall be invoked at p.

[6]For the standard representation of calling contexts, see Nielson *et al.*'s Principles of Program Analysis [Nielson et al. 2005, Ch.2].

3.2 Cost Models

A *cost model* assigns a numerical cost to each node and each edge of the ECG. Values assigned to nodes represent the cost of running a certain function on a certain processor. These numbers are estimated using various criteria of processor affinity. Values assigned to edges represent the cost of passing data between functions. If a call happens between functions running on different processors, this cost shall account for the data transfer that is implied in such an invocation. We define this notion as follows:

Definition 3.5 (Cost Model). A cost model is a pair of functions, C_v and C_e. C_v maps ECG nodes to costs: $C_v(i, f)$ is the cost of invoking function f in processor p_i. C_e maps ECG edges to costs: $C_e(v_{i/f}, v_{j/g})$ is the cost of function f calling function g, while f runs on p_i, and g runs on p_j.

3.3 Static Scheduling

The static scheduling problem consists of finding an assignment of functions to processors in such a way to minimize the cost of executing a program. Definition 3.6 states the problem for a ECG of single root. Every call graph has a single root, henceforth called main. However, it is possible that its corresponding ECG has multiple roots, one for each processor where main might run. In this case, we convert it to a singly-rooted ECG by creating a mock root node $v_{\bullet/\texttt{root}}$, and adding edges from it to every node that represents main. We let $C_v(v_{\bullet/\texttt{root}}) = 0$.

Definition 3.6 (The Static Scheduling Problem (SSP)). Given a ECG G, derived from a call graph CG, plus a cost model CM, determine a subgraph $G' \subseteq G$, with these two properties:

(1) For every edge $v_f \to v_g \in$ CG and every processor p_i that can execute function f, there must exist one, and only one, edge $v_{i/f} \to v_{j/g} \in G'$, where p_j is a processor that can execute function g.
(2) The cost of $v_{\bullet/\texttt{root}}$, with regards to CM, must be minimal, where $v_{\bullet/\texttt{root}}$ is the root node of G.

Example 3.7. A solution to the static scheduling problem is a subgraph of the ECG. Figure 6 shows such a subgraph. The part of this subgraph that is reachable from the root node represents the program that we shall schedule to execute.

3.4 An Optimal Solution to SSP

We want to build the graph S that minimizes the cost of the root node of the extended call graph. Algorithm 1 builds this graph. This algorithm relies on the fact that SSP has an *optimal substructure*, as we state in Theorem 3.9. A problem is said to have optimal substructure if an optimal solution to it can be constructed efficiently from optimal solutions of its subproblems [Bellman 1957, Chap.III.3]. In our case, the subproblem is to compute the scheduling starting at a given node of the ECG. The cost T of a node is defined as follows:

Definition 3.8 (Scheduling Cost of a Node). The scheduling cost of $v_{i/f}$ is recursively defined by the expression below, where $E = v_{i/f} \to v_{j/g}$, and S is a graph that represents a solution of SSP, starting at $v_{j/g}$

$$T(v_{i/f}) = C_v(v_{i/f}) + \sum_{E \in S} (T(v_{j/g}) + C_e(E))$$

$T(v_{\bullet/\texttt{root}})$ is the objective function of SSP (see Def. 3.6).

Intuitively, $T(v_{i/f})$ gives us the cost of running function f, i.e., $C_v(v_{i/f})$, plus the cost of executing all the other functions invoked from f. Thus, $T(v_{\bullet/\texttt{root}})$ gives us the cost of the whole

program execution according to the cost model; hence, it is the objective function that we want to minimize. Because of its optimal substructure, T can be computed in polynomial time by a dynamic programming algorithm.

```
Data: ECG G, CM (C_v, C_e)
Result: S
SCCs = findStronglyConnectedComponents(G)
C = new Table; S = new Graph
for scc in reverseTopologicalSort(SCCs) do
    for v_{i/f} in scc do
        C[v_{i/f}] = C_v(f, p_i)
        for g in calledFunctions(f) do
            p_x = arg min_{p_j ∈ Proc}(C_e(v_{i/f}, v_{j/g}) + C[v_{j/g}])
            if v_{x/g} not in scc then
                C[v_{i/f}] = C[v_{i/f}] + C_e(v_{i/f}, v_{x/g}) + C[v_{x/g}]
                S = S ∪ {v_{i/f} → v_{x/g}}
            end
        end
    end
end
```

Algorithm 1: Static scheduler. This algorithm uses the following data structures: **Scheduler** S, graph introduced in Definition 3.6. **Cumulative cost** $C : \mathit{Node} \mapsto \mathbb{N}$, the cost of running the computation denoted by node v. **Local cost** $C_v : (\mathit{Node}) \mapsto \mathbb{N}$, the cost of running the instructions that constitute function f, except call instructions, at processor p_i. **Transfer cost** $C_e : (\mathit{Node} \times \mathit{Node}) \mapsto \mathbb{N}$, the cost of transferring data from a node $v_{i/f}$ to another node $v_{j/g}$.

Theorem 3.9. *SSP has optimal substructure.*

> **Proof:** SSP takes a cost model CM as input. This cost model considers that the cost of two different calls to the same function on the same processor never varies. Furthermore, the cost of transferring parameters and results only depends on the processors on which the call occurs. Thus, to compute $T(v_{i/f})$, we compute $T(v_{j/g})$ independently, for each function g that f invokes. The implication of this fact is that T, in Definition 3.8, is a Bellman Equation [Bellman 1957], a necessary enabler of a dynamic programming algorithm. Therefore, given a node of the ECG, all decisions that must be made regarding where to place each of the function calls are independent. As such, we can optimize each decision independently by considering all possible options. □

Algorithm 1 first finds the strongly connected components of the ECG (for example using Tarjan's algorithm), and orders them in reverse topological order. In the absence of recursion, each strong component corresponds to a single node in the ECG. For each node $v_{i/f}$, in reverse topological order, Algorithm 1 builds a subgraph S starting at $v_{i/f}$ to solve SSP. This step runs independently for each node; thus, this subgraph is a static scheduling for that node. Algorithm 1 uses a table C to remember the cumulative cost associated with each node that it has visited. Because of the reverse topological order, it is guaranteed that, once a node $v_{i/f}$ is visited, all the other nodes on which $v_{i/f}$ depends were already visited and had their total costs placed in C.

Complexity. The ECG has $O(|F||P|)$ nodes, being F the set of functions in the original program and P the set of processors in the architecture at hand. Each one of the E edges in the original call graph originates $O(|P|)$ edges in the solution graph, and each edge is chosen among $O(|P|)$ options; hence, Algorithm 1's time complexity is $O(E|P|^2)$. Since $|P|$ is usually a small constant (e.g.

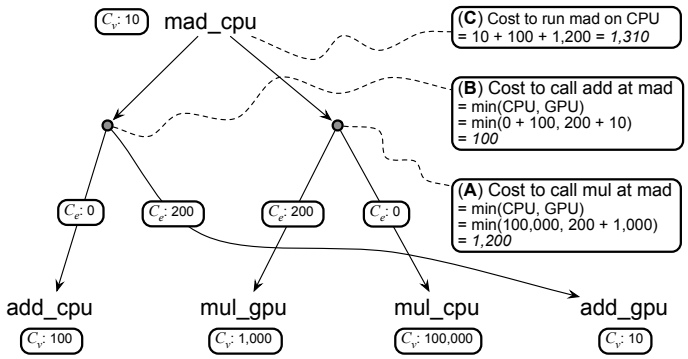

Fig. 5. Computation of costs to invoke mad (see Figure 2 (b)) in the CPU.

2 in a CPU-GPU architecture), in typical scenarios Algorithm 1 runs in linear time on the original program's size.

Example 3.10. Figure 5 shows how Algorithm 1 finds the cost of invoking function mad, (Figure 2 (b)), in the CPU. The algorithm starts from the leaves of the ECG. Nodes add_cpu, add_gpu, mul_cpu and mul_gpu represent routines that do not call other functions. Their invocation costs are queried directly through C_v. To compute the cost of node mad_cpu, we must consider four cases: (i) call add and mul in the CPU; (ii) call add in the CPU and mul in the GPU; (iii) call add in the GPU and mul in the CPU; or (iv) call add and mul in the GPU. Because the cost of calling add does not depend on the cost of calling mul, we only need to make two decisions: calling add on CPU or GPU, and calling mul on CPU or GPU. Algorithm 1 assumes that the cost of transferring data – the matrices in this example – is always constant, e.g., 200 units. Thus, we find it best for mad_cpu to invoke add on the CPU and mul on the GPU. This scheduling costs: $10_a + 1,000_b + 100_c + 0_d + 200_e = 1,310$, given by the following costs: (a) run mad_cpu; (b) run mul_gpu; (c) run add_cpu; (d) transfer data from mad_cpu to add_cpu; and (e) transfer data from mad_cpu to mul_gpu. Therefore, when calling mad at the CPU, Algorithm 1 schedules add also in the CPU, but calls mul in the GPU.

Example 3.11. Figure 6 shows the result of our algorithm when applied to the program seen in Figure 1. Our final scheduling invokes mul always on the GPU, and add on the CPU, if this function is called from main. However, if this function is called from mad, then it is invoked on the GPU.

Examples 3.10 and 3.11 used illustrative values for the parameters of the cost models. Instead of manually trying to determine effective values for a given architecture, we resort to heuristics to automatically find parameters that maximizes the effectiveness of Algorithm 1 on the hardware at hand. Section 4 describes our approach.

4 AUTOMATIC CALIBRATION OF A COST MODEL

Algorithm 1 is parameterized by a cost model. There are several ways to build a cost model for a given processor. For instance, Sim *et al.* have provided an analytical cost model for a GPU, which can be adapted to different kinds of graphics processing units [Sim et al. 2012]. In our case, the cost model does not have to reflect precisely the execution time of a function, in a processor, given a certain workload. Although cost models reflect running time, there is no requirement that these costs correlate (in the statistical sense) with the actual running time of programs. Rather, we need cost models that provide Algorithm 1 clues about the relative speed between different processors when executing the same function. Thus, the actual unit or magnitude of the costs does not matter.

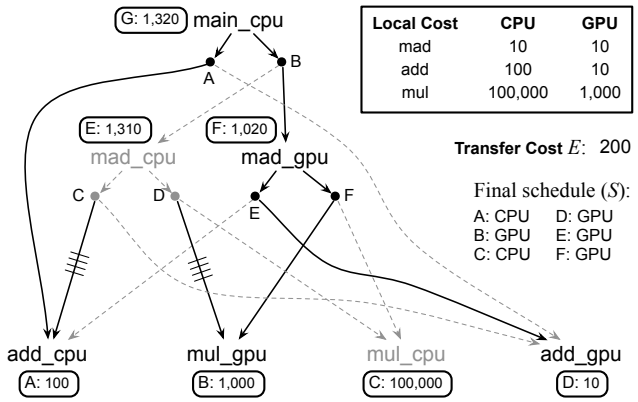

Fig. 6. Application of our static scheduler on the Extended Call Graph seen in Figure 2 (b). Solid edges represent actual invocation paths, as computed by the scheduler. The two hatched lines cannot be reached from the ECG's root note. Tags (A-F) show the calling cost of each invocation.

Computation Costs. Different processors use different cost models. Example 4.1 describes a very simple cost model for typical GPUs. We do not consider this model a contribution of this paper, since there exists a vast body of literature on the precise simulation of different computer architectures [Zeng et al. 2009]. It is only a pragmatic way of enabling static scheduling. Nevertheless, Example 4.1 describes the essential features of a cost model for a processor: a processor's cost model is an algorithm, parameterized by a number of constants, which, once given a function f, determines a value for f. The value depends on structural characteristics of f, and on the model's constants. Section 4.1 addresses the problem of determining good constants.

Example 4.1. Algorithm 2 provides a cost model for a GPU. It considers different factors mentioned by Sim *et al.* [Sim et al. 2012], including: (i) the parallelism factor; (ii) the cost of divergences; and (iii) the cost of launching a kernel. Additionally, we consider also the nesting depth of each instruction inside loops: if an instruction is contained in k nested loops, the model considers it executes $(\text{loop\_iterations})^k$ times. Instructions within divergent branches are assumed to have a cost of gpu_branch_cost; other instructions have a cost of 1. Algorithm 2 analyzes code written in plain C. If an instruction is within a parallel loop, its cost is divided by parallelism_factor, a constant representing the number of threads available in the target processor. However, Algorithm 2 does not allow the cost of an instruction to fall below 1 – the parallel execution of an instruction cannot be faster than a single execution on one processor. A top-level parallelizable loop, or one nested in another loop that is not parallelizable, will be turned into a GPU kernel. Hence, it incurs an initialization cost given by kernel_launch_cost.

A heterogeneous architecture might feature several kinds of processors. Each of them might use a potentially different formula to feed Algorithm 1 with a cost model. To illustrate these differences, Example 4.2 presents a cost model for typical multi-core CPUs. We suggest the reader to contrast this model with the one that Example 4.1 had described for a GPU.

Example 4.2. Algorithm 3 is an example of a cost model for a CPU. The more deeply nested within a loop is the instruction, the higher its cost. This simple model considers that every loop executes a fixed number of iterations. It also assumes both branches of conditional statements are always executed. Thus, the depth of the innermost loop that contains an instruction is enough to

```
Data: Function f
Result: Cost
c = 0
for instr in f do
    depth = loopDepth(instruction)
    instruction_cost = (loop_iterations)^depth
    if isWithinDivergentBranch(instr) then
        instruction_cost = instruction_cost × gpu_branch_cost
    end
    if isContainedInParallelizableLoop(instr) then
        instruction_cost = instruction_cost / parallelism_factor
    end
    instruction_cost = max(1, instruction_cost)
    c = c + instruction_cost
end
for loop in get_loops(f) do
    if isParallelizable(loop) and not
      isContainedInParallelizableLoop(loop) then
        # A kernel will be created containing this loop
        c = c + kernel_launch_cost
    end
end
return c
```

Algorithm 2: Cost model of a GPU. Parameters, e.g., factors that vary with the architecture, are underlined.

```
Data: Function f
Result: Cost
c = 0
for instr in f do
    depth = loopDepth(instr)
    c = c + (loop_iterations)^depth
end
return c
```

Algorithm 3: An example of cost model for a CPU. The only parameter is underlined.

estimate the number of times the instruction shall execute: if such depth is d, the instruction is assumed to execute $(loop\_iterations)^d$ times.

Algorithm 3 computes costs per assembly instructions. The cost of a function is a weighted sum of the costs of each of its instructions. Different kinds of instructions take a different number of cycles and resources to execute. In this paper we did not try to account for these differences when testing cost models. There exists a vast body of literature on the precise simulation of different computer architectures [Zeng et al. 2009]. It is possible to use this knowledge, already in place, to improve the precision of a cost model; however, this problem is not the focus of this paper, and we shall not try to solve it.

Transfer Costs. In addition to the cost of running code in processors, Algorithm 1 also takes into consideration the cost of moving data between processors. This cost varies with the architecture. For instance, it is more costly to move data from a CPU to a remote server running on the cloud, than to move data from a CPU to an accelerator on the same chip. The cost of transferring data naturally varies with the size of the data; however, in this paper we do not consider this variation.

The rationale behind this decision is that for small data sizes, transfer costs are negligible. Section 5 shall provide empirical evidence to corroborate this observation. Example 4.3 describes a simple cost model for inter-process communication. Like in Example 4.1, to determine the constants of this cost model, we resort to the probabilistic search method given in Section 4.1.

Example 4.3. When transfering data to a remote processor, the transfer cost tends to be dominated by arrays. When a function call is performed on the same processor, however, no transfer cost is incurred. Based on such premisses, Algorithm 4 estimates the cost of transferring data among different processors of a heterogeneous architecture.

```
Data: Origin processor S, target processor D, list of data to be moved A
Result: Cost
if S == D then
  | return 0
end
c = 0
for argument in A do
  | if isArray(A) then
  |   | c = c + array_transfer_cost
  | end
end
return c
```

Algorithm 4: Example of model to estimate the cost of transferring data, given a call $g(A)$ in function f, where f runs in processor S, and g runs in processor D.

4.1 Optimizing Cost Models with Simulated Annealing

Static cost models, such as the ones presented in this section, contain parameters that allow Algorithm 1 to account for hardware characteristics, such as the latency and throughput of the communication channel between different processors. In order to have a usable cost model, i.e. one that assigns costs to computations, those architecture-specific parameters must be assigned concrete values. This is not a simple task, however, given that parameters might interact with each other in complex ways. We call the challenge of finding good values for the parameters of a cost model given an architecture the *Calibration Problem*. Definition 4.4 states this problem more formally. This definition uses a vague notion of efficiency on purpose: a cost model might be customized to minimize energy consumption, runtime, memory transfer time, etc - or even a combination of these objectives.

Definition 4.4 (Calibration Problem). Given a cost model, parameterized by arguments $\mathcal{A} = \{a_1, \ldots, a_m\}$, $0 \leq a_i$, a training set of benchmarks $\mathcal{B} = \{b_1, \ldots, b_n\}$, and a set of processors $C = \{p_1, \ldots, p_k\}$, find an assignment $\mathcal{V} : \mathcal{A} \mapsto \mathbb{R}$ that maximizes a user-defined efficiency function of the benchmarks in $\mathcal{B}$, running on the processors of C, according to the scheduling produced by Algorithm 1.

Solutions of the Calibration Problem range on an infinite space of unknown structure; hence, we cannot design an algorithm to solve it exactly. Thus, to deal with this problem we resort to a heuristic: *Simulated Annealing* (SA) [Cerny 1985]. Simulated annealing is a probabilistic technique for approximating the global optimum of a function. SA iterates through different configurations of the system that must be optimized. In our case, a configuration is an assignment of values

to parameters $\mathcal{A}$ of the cost model. An iteration consists of slightly modifying the value of the parameters and then computing the objective function again. This value is compared to the current solution, and with a certain probability the new configuration is adopted. After a fixed number of iterations, SA is likely to find a local optimum in the search space.

Simulated Annealing requires an objective function. In our case, it consists of a measure for evaluating the quality of a configuration $\mathcal{A}$ (i.e. it must be some function $f : \mathcal{A} \mapsto \mathbb{R}$). We use the training set of benchmarks to evaluate this quality. The programs in the training set need to be deterministic, i.e. perform the same amount of computation given same inputs. Basically, we run each program in its original form on a baseline processor (e.g. the CPU), and calculate the speedup or slowdown gained by the scheduling suggested by Algorithm 1 when given a cost model parameterized by $\mathcal{A}$. If $b_1, \cdots, b_n$ are the benchmarks in the training set, we denote the average cost to run b_i on the baseline processor by $O(b_i)$. $E(b_i, \mathcal{A})$ is the cost to run b_i with the scheduling assigned to it by Algorithm 1, given a cost model parameterized by $\mathcal{A}$. From these notions, we define the quality of a set of cost model parameters $\mathcal{A}$ as follows.

Definition 4.5 (Quality of the Cost Model). The quality of a cost model, $Q(\mathcal{A})$, is defined below. We use the notation $[p] = 1$ if the predicate p is true; otherwise $[p] = 0$:

$$Q(\mathcal{A}) = \frac{1}{n}\sum_{i=1}^{n}\left([O(b_i) < E(B, \mathcal{A})]\left(\frac{E(b_i, \mathcal{A})}{O(b_i)}\right) - [O(b_i) > E(b_i, \mathcal{A})]\left(\frac{O(b_i)}{E(b_i, \mathcal{A})}\right)\right)$$

To calculate the quality of a set of parameters, Definition 4.5 adds up speedups and subtracts slowdowns, dividing the result by the number of benchmarks n. We want to find $\mathcal{A}$ that maximizes $Q(\mathcal{A})$. The division by n does not change the optimization process, but it helps us interpret $Q(\mathcal{A})$: it measures the average effect of applying Algorithm 1 when its cost model is parametrized by $\mathcal{A}$. If $Q(\mathcal{A})$ is positive, we are more likely to obtain a speedup than a slowdown once we feed Algorithm 1 with these arguments. If $Q(\mathcal{A})$ is zero, we should not expect any change. If it is negative, we are more likely to observe a slowdown.

As Example 4.6 shows, our solution to the Calibration problem consists of an instantiation of Simulated Annealing with our problem-specific features. $Q(A)$ is the objetive function we want to maximize. We use the most common choices of temperature function (linear decay) and transition probability. The algorithm starts with random values for parameters given by $\mathcal{A}$. In each iteration, it changes each value in $\mathcal{A}$ by a random amount, obtaining a new candidate solution $\mathcal{A}'$, and calculates the quality $Q(\mathcal{A}')$. If $Q(\mathcal{A}') > Q(\mathcal{A})$, then the new solution is always taken. Otherwise, it is taken with a probability that depends on $Q(\mathcal{A}) - Q(\mathcal{A}')$.

Example 4.6. Figure 7 shows a few iterations of our calibration algorithm using only the structure in Figure 2 (b) as our training set. For simplicity, we consider just two parameters of the cost model: a_1 = parallelism_factor (number of available threads – see Example 4.1) and a_2 = array_transfer_cost (see Example 4.3). We assume an architecture with two processors, a CPU (C), and a GPU (G). Calibration starts with random values for all the parameters used in the cost model. In this case, it starts with $a_1 = 100$ threads and $a_2 = 25$ GB/sec. Depending on the values of a_1 and a_2, Algorithm 1 assigns contexts to different processors. For the only program in this example's training set, we have four calling contexts: () → main, main → add, main → mad, and mad → add. For each one we have a choice of where to run the callee; hence, we describe regions by a four-elements vector.

During Simulated Annealing, the calibration algorithm tries other choices of values for the parameters by adding or subtracting random deltas to current values. New values are used to compile the programs in the training set. Speedups (the quality of the evaluated cost model) are then calculated. For instance, in the second iteration, the algorithm evaluated the cost model defined

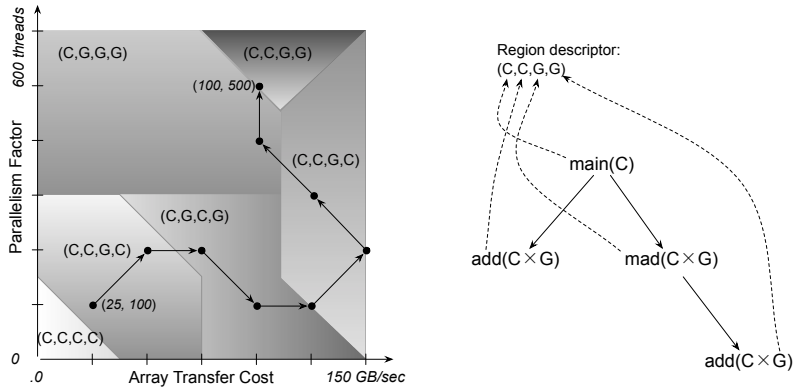

Fig. 7. Example of search using simulated annealing and two parameters. Regions are only illustrative: they do not necessarily describe the actual search space.

by $a_1 = 200$ threads and $a_2 = 50$ GB/sec. The resulting speedup is used to decide whether the algorithm keeps or discards current values. After a pre-defined number of iterations, set by the user before execution begins, we are likely to find a better cost model than the one given by the random initial values.

This example shows eight iterations of Simulated Annealing. After the eighth, we have found an assignment of values to arguments ($a_1 = 500$ threads, $a_2 = 100$ GB/sec) that gives us the best speedup among all the eight runs of the training set. Hence, this assignment is likely to help our code placement algorithm to benefit more from the target architecture. In this configuration, we run main on the CPU, and perform the call main → add on the CPU as well. The two other contexts are mapped to the GPU. This cost model is then ready to be used, without further calibration, in programs not yet seen.

4.2 Implementation Details

Thus far, we have discussed cost models and simulated annealing in a rather abstract way. To provide concrete support for such notions, we have materialized them into a tool, Etino, which, as Figure 4 clarifies, is used in combination with other compilation equipment. In the rest of this section, we discuss some details of Etino's implementation which we believe are important for readers interested in reproducing our results.

Implementing Cost Models. We use analyses already in place in LLVM to compute the cost of each function in a program, according to Algorithms 2, 3 and 4. To find divergence branches, LLVM provide us with Sampaio *et al.*'s divergence analysis [Sampaio et al. 2014]. LLVM's typed intermediate representation lets us compute the transfer cost of function arguments and return values. We use structural analyses to obtain the nesting depth of each instruction. Finally, we build a program's ECG by traversing its call graph. Algorithm 3 computes costs per assembly instructions. The cost of a function is a weighted sum of the costs of each of its instructions. Different kinds of instructions take a different number of cycles and resources to execute. In this paper we did not try to account for these differences when testing cost models.

Insertion of data-copy primitives. OpenACC provides different pragmas to move data between the CPU and the GPU. Examples of these primitives can be seen in Figure 3. The automatic insertion of copy directives requires symbolic knowledge of the size of arrays. For instance, in Figure 3 Etino

infers that array dst has N positions. To perform this inference, Etino uses the infra-structure already present in dawncc [Mendonça et al. 2017], a tool that inserts OpenACC and OpenMP 4.0 annotations in programs automatically. Our implementation is able to infer correct bounds to every array in the benchmarks that we use in Section 5. It infers ranges for about 45% of the arrays in SPEC CPU 2006. The limitation, in this case, is aliasing: dawncc uses LLVM's points-to analysis to disambiguate pointers. Whenever points-to analysis returns multiple aliases to an array, dawncc gives up finding precise bounds to it. Consequently, Etino cannot insert annotations in this case.

Discovery of Parallel Regions. Etino marks every loop that contains only accesses to bounded arrays with the OpenACC directive kernels. An array is "bounded" if we can infer its limits using dawncc's region analysis. This region analysis is implemented after the techniques first introduced by Rugina and Rinardi [Rugina and Rinard 2005]. We use pgcc's dependence analysis to detect if a loop marked with the kernels pragma is parallel. To increase the number of parallel loops, we resort to restrictification [Alves et al. 2015; Sperle Campos et al. 2016]: a technique to disambiguate pointers at runtime. The symbolic range analysis of Rugina *et al.* already gives us the necessary equipment to insert disambiguation checks. Furthermore, OpenACC includes conditional pragmas, which let us apply parallelization only in the absence of dependences created by aliasing.

Example 4.7. The temporary RST_AI1, in functions GPU_mul and GPU_add (Figure 3), is true whenever there is no overlapping between arrays accessed inside the parallel region.

Code Cloning. Whenever Etino is able to bound all the arrays within a loop that is part of a given function f, it produces a clone GPU_f of this function. This clone will augment the extended call graph of the target program; hence, allowing Algorithm 1 to schedule f on the graphics processing unit. The only difference between a function and its clone is due to the OpenACC directives that Etino creates to indicate where said clone will run.

Running Simulated Annealing. Etino runs 100 iterations of Simulated Annealing. This method is parameterized by a *temperature function*. The temperature function that Etino uses is a linear decay, starting at 10 and linearly approaching 0. To generate *neighbour states*, we independently make each of the parameters in $\mathcal{A}$ larger or smaller by a random factor of up to 50%, with uniform probability. All parameters are initialized with small random values. Each iteration of Simulated Annealing involves measuring the running time of several programs both in its sequential and parallel versions. Etino averages the running time of 5 runs. In spite of the parameter space being continuous, Etino's decisions are discrete: they ultimately lead to a solution of Algorithm 1. Thus, in many iterations, equal programs will be generated by Etino for the same benchmarks. To speed up the iteration process, Etino uses a cache to find when it has generated programs that were already evaluated in previous iterations.

5 EVALUATION

The goal of this section is to demonstrate that our algorithm effectively generates efficient code for heterogeneous architectures. To achieve this end, we shall provide answers to these questions:

- **RQ1:** Given an actual architecture, can our tuning algorithm find effective cost models that generalize well (i.e. are effective for programs not seen in the tuning process)?
- **RQ2:** Can Etino optimize programs effectively without programmer intervention?
- **RQ3:** Can Etino be used as a guiding tool, helping developers to decide where to run each program part?
- **RQ4:** How much does Etino's simplification of ignoring input size (it is fully static) affect its performance?

- **RQ5:** Is Etino's interprocedural analysis useful in helping programmers to use heterogeneous hardware?

Methodology. To answer these questions, we use programs taken from four benchmarks: PolyBench, C version[7], MgBench[8], the Computer Language Benchmarks Game[9] (BenchGame), and the DataMining Benchmark Suite [Da Mata et al. 2013]. PolyBench contains various C programs that perform linear-algebra, statistics and stencil computations. MgBench is the set of benchmarks distributed along with ipmacc [Lashgar et al. 2014], an OpenACC compliant compiler, and dawncc [Mendonça et al. 2017], the source-to-source tool that we use to insert OpenACC annotations. MgBench mixes linear-algebra and data mining algorithms. As we show in this section, compiling these programs to a GPU without an algorithm such as ours is not always profitable. BenchGame contains more traditional C programs. We use this suite to demonstrate that our technique is applicable even outside the domain of data-parallel applications. DataMining contains more complex and irregular applications, which exercise the interprocedural aspects of Algorithm 1. In total, these four benchmarks give us 33 programs: 16 from PolyBench, 6 from MgBench, 8 from BenchGame and 3 from DataMining. The 30 programs in PolyBench, MgBench and BenchGame can be processed fully automatically by dawncc to implement the scheduling computed by Etino. These benchmarks are used in Sections 5.1 and 5.2. The same is not true for the more complex programs in DataMining. Nevertheless, Etino can still suggest a scheduling for them, which is profitable if implemented by the programmer, as we show in Section 5.5. To run the benchmarks, we have used the three following setups:

Setup 1: *CPU:* Intel Xeon 2GHz; *GPU:* Nvidia GTX 670;
Setup 2: *CPU:* Intel i7 3.4GHz; *GPU:* Nvidia Tesla C2070.
Setup 3: *Chipset:* ARM Exynos 7420 Octa, *CPUs:* 4x2.1 GHz Cortex-A57 + 4x1.5 GHz Cortex-A53, *GPU:* Mali-T760MP8.

Etino requires an external compiler to translate OpenACC/OpenMP 4.0 directives into binary code. The compiler used in the first two setups was the PGI C Compiler version 16.1 64-bits (pgcc). In Setup 3, we used a modified version of LLVM/Clang that has an OpenMP backend and runtime for the Exynos architecture. Finally, in Section 5.3 we change Setup 1, to use ppcg [Verdoolaege et al. 2013], instead of pgcc. Ppcg is a source-to-source compiler that translates C code into C for CUDA. In this process, ppcg applies several optimizations in the program, such as tiling, loop fusion and loop unrolling. Our intention, by using ppcg, is to show that mappings produced by Etino can be used to guide developers in the task of choosing where each program part should run.

This Section in a Nutshell. Table 1 shows statistics about all benchmarks used: original number of functions and number of lines in their source code. We also show how many functions are assigned to each processor by Etino after it learns from training data on each of the three available architectures. Usually, Etino assigns only a small group of routines to the GPU, and most of the routines run only either on the CPU or on the GPU. However, in the larger benchmarks, such as Itemsets, a function is invoked both on the CPU from some contexts and on the GPU from others. Table 1 shows that Etino's scheduling can differ, depending on the architecture. This is a consequence of the calibration methodology discussed in Section 4.1. This methodology lets Etino achieve performance portability across architectures, since it uses training data from each hardware to choose its scheduling. As we explain in the rest of this section, this flexibility is an important factor to avoid the slowdowns that come out of reckless parallelization. The table also

[7] http://web.cs.ucla.edu/~pouchet/software/polybench
[8] https://github.com/lashgar/ipmacc
[9] http://benchmarksgame.alioth.debian.org

Table 1. Statistics about benchmarks, sorted by number of functions, and the placement that Etino produces, after the training phase on each setup. Setups 1, 2 and 3 are denoted by St. 1, St. 2 and St. 3, respectively. **LoC**: number of uncommented lines of code. **Functions**: number of functions in the original program. $X/Y/Z$ mean that X functions are placed on the CPU, Y on the GPU, with Z being in both depending on the calling context. The * symbol marks benchmarks that were transformed by Etino; the others were not deemed profitable to accelerate. Indeed, we have observed several slowdowns when mapping those programs onto different GPUs. The right side of the table shows compilation times measured on Setup 1. Etino's implementation works in three phases. Column **Time LLVM** shows the time taken by Algorithm 1. Column **Time Clang** shows the time taken to change the original C++ code – for instance, by cloning functions. Column **Time dawncc** shows the time that dawncc takes to analyze and annotate programs. We had to annotate the DataMining benchmarks manually; hence, we do not show dawncc's time for them.

Benchmark	LoC	Functions	St. 1	St. 2	St. 3	Time LLVM	Time Clang	Time dawncc
* DM/itemsets	193	12	10/3/1	10/3/1	10/3/1	0.55s	0.75s	N/A
* MG/other-nearest	262	9	7/2/0	7/2/0	7/2/0	0.09s	0.84s	1.58s
MG/vector-prod	160	8	8/0/0	8/0/0	8/0/0	0.10s	0.85s	1.83s
MG/mat-sum	171	8	8/0/0	8/0/0	8/0/0	0.10s	0.86s	2.23s
* MG/mat-mul	190	8	6/2/0	6/2/0	6/2/0	0.10s	0.87s	2.46s
* MG/lu	209	8	6/2/0	6/2/0	6/2/0	0.11s	0.89s	6.51s
* MG/k-nearest	261	8	6/2/0	6/2/0	6/2/0	0.10s	0.92s	1.12s
* MG/floyd	273	8	8/0/0	6/2/0	8/0/0	0.11s	0.96s	0.90s
* MG/cholesky	212	8	6/2/0	6/2/0	6/2/0	0.10s	0.89s	3.10s
* DM/kmeans	439	8	6/6/4	6/6/4	6/6/4	0.98s	1.74s	N/A
BG/Puzzle	87	7	7/0/0	7/0/0	7/0/0	0.08s	0.41s	1.44s
* MG/collinear-list	230	6	4/2/0	4/2/0	4/2/0	0.10s	0.97s	2.92s
* DM/knn	308	6	3/3/0	3/3/0	3/3/0	1.00s	1.66s	N/A
MG/str-matching	198	5	5/0/0	5/0/0	5/0/0	0.09s	0.94s	1.98s
MG/search-vector	150	5	5/0/0	5/0/0	5/0/0	0.08s	0.88s	1.58s
BG/SpectralNorm	64	5	5/0/0	5/0/0	5/0/0	0.09s	0.71s	1.70s
BG/Fasta	134	5	5/0/0	5/0/0	5/0/0	0.09s	0.49s	1.11s
* PB/2MM	237	4	3/1/0	3/1/0	3/1/0	0.10s	0.88s	2.81s
BG/NBody	143	4	4/0/0	4/0/0	4/0/0	0.09s	0.76s	3.51s
* PB/SYRK_M	168	3	3/0/0	1/2/0	2/1/0	0.09s	0.84s	1.72s
* PB/SYRK	186	3	3/0/0	1/2/0	2/1/0	0.09s	0.84s	2.10s
* PB/SYR2K	198	3	3/0/0	1/2/0	2/1/0	0.10s	0.85s	2.27s
PB/MVT	198	3	3/0/0	3/0/0	3/0/0	0.12s	0.84s	2.08s
PB/GRAMSCHM	214	3	3/0/0	3/0/0	3/0/0	0.12s	0.95s	2.79s
PB/GESUMMV	186	3	3/0/0	3/0/0	3/0/0	0.11s	0.63s	1.82s
* PB/GEMM	200	3	2/1/0	2/1/0	2/1/0	0.09s	0.63s	1.95s
* PB/FDTD-2D	240	3	3/0/0	2/1/0	2/1/0	0.13s	0.88s	3.38s
* PB/COVAR	221	3	3/0/0	2/1/0	2/1/0	0.10s	0.86s	2.77s
* PB/CORR	286	3	3/0/0	2/1/0	2/1/0	0.11s	0.82s	3.92s
PB/BICG	214	3	3/0/0	3/0/0	3/0/0	0.12s	0.75s	2.01s
PB/ATAX	176	3	3/0/0	3/0/0	3/0/0	0.09s	0.83s	1.86s
* PB/3MM	270	3	2/1/0	2/1/0	2/1/0	0.12s	0.90s	3.36s
* PB/3DCONV	205	3	1/2/0	1/2/0	1/2/0	0.11s	0.66s	3.94s
* PB/2DCONV	183	3	3/0/0	2/1/0	2/1/0	0.10s	0.64s	2.34s
BG/PartialSums	66	3	3/0/0	3/0/0	3/0/0	0.10s	0.59s	1.31s
BG/FannKuch	111	2	2/0/0	2/0/0	2/0/0	0.10s	0.45s	1.09s
BG/Recursive	55	1	1/0/0	1/0/0	1/0/0	0.10s	0.29s	0.76s
BG/NsieveBits	37	1	1/0/0	1/0/0	1/0/0	0.09s	0.43s	0.99s

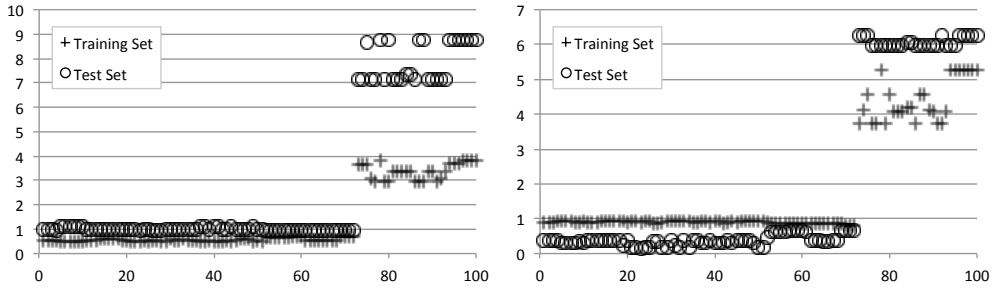

Fig. 8. Evolution of the quality of the cost model (Definition 4.5) during Simulated Annealing on Setup 1, for two folds.

shows the time taken to analyze and modify the benchmarks. The time taken by Algorithm 1 (column **Time LLVM**) is the same, regardless of the experimental setup. For the other two columns (**Time Clang** and **Time dawncc**) we report the time measured for Setup 1.

5.1 RQ1: The Effectiveness of Automatic Calibration

To answer the first research question, we evaluate Etino's capability of automatically tuning a cost model for a given architecture. We want the resulting cost model to be effective for programs not seen in the training phase. We divided the 30 evaluated programs in three arbitrary groups, mimicking the k-fold cross validation method [Kohavi 1995]. We optimize programs in each group (fold) while using the other two groups as the training set during Simulated Annealing. In this experiment we leave the DataMining benchmarks out, because dawncc, the annotator we use, does not handle those programs without user intervention.

Figure 8 shows the evolution of the Quality (as defined in Section 4.1) of the cost model for two of the three folds on Setup 1. Very similar figures are obtained (with different scales) in other folds, and also on Setups 2 and 3, which we omit to avoid redundancy. In each iteration, we run Algorithm 1 using the current cost model parameters on all benchmarks. Then, Simulated Annealing uses the quality obtained in the training set to decide whether to keep or discard the current values for the parameters. Thus, while we show the Quality in the test set in the figure, the algorithm is not aware of it. Rather, we use it to check that the algorithm is finding a general cost model: we want that Quality increases in the test set even though the algorithm is only optimizing the parameters based on the training set.

As we can observe, quality increases considerably in the training and test sets as the number of Simulated Annealing iterations mounts up, showing that the model learned from the training set also generalizes for the test set. In both figures, we observe a sharp increase on the perceived quality of the model, after approximately 70 iterations of simulated annealing. This behavior happens because simulated annealing ranges on a discrete space: changes in one parameter might lead to very different performance results. Etino starts with a conservative cost model, using only the CPU, and tries many combinations of parameters before finding values that start using the GPU profitably. In all 3 folds, it eventually converges on parameters that manage to optimize programs in the training and test sets. Therefore, given an architecture, Etino can adapt its cost model using a representative set of programs for training, in such a way that its general performance (even when applied to programs outside the training set) is also enhanced. Such evidence suggests a positive answer to our first research question.

We notice that cost model parameters do not necessarily map exactly to hardware specifications. There might exist more than one set of parameters leading to similar results in the same architecture. In Setup 1, Simulated Annealing produced different absolute numbers for each of the three folds. However, these numbers, albeit individually different, caused the code placement algorithm to perform similar decisions. We have observed another interesting fact in this experiment. Whether we started with random values or with educated guesses based on hardware parameters did not change the quality of the cost model. We observed that educated guesses help simulated annealing to converge faster. However, with enough iterations the quality of the cost model was the same.

5.2 RQ2: Fully Automatic Runtime Gains

This section evaluates Etino's capability of taking advantage of heterogeneity by comparing it to other two scheduling strategies: dawncc, which sends all parallel computations to the GPU; and **noop**, which runs programs entirely on the CPU. Figure 9 (Left) shows speedups obtained by both Etino and dawncc on every available benchmark that dawncc handles automatically[10]. These results were obtained on Setup 1; Figure 9 (Right) shows the same comparison on Setup 2, and Figure 10 on Setup 3. Speedups are relative to **noop**. We omit programs in which no computation on the GPU is performed neither by Etino nor by dawncc. To optimize each program, Etino uses the cost model obtained in the end of the training phase (see Section 5.1) reached when that program was not in the training set. A speedup of 0 indicates no change relative to sequential execution, whereas a slowdown of X times is shown as $-X$.

On Setups 1 and 2, our static scheduler obtains all of the larger speedups produced by dawncc ($\geq$ 32x in 3MM, $\geq$ 23x in 2MM, and $\geq$ 11x in GEMM and mat-mul on Setup 1). However, dawncc has serious drawbacks, especially in the more common C programs from BenchGame, in which reckless parallelization can bring slowdowns of up to $720x$. Such slowdown was observed in FannKuch on Setup 2. In several benchmarks, the speed gained by offloading does not pay off for the cost of transfering data. Large slowdowns are caused by the amplification of this effect when inappropriately offloaded functions are called in a loop. Etino is able to avoid all these cases but three, only allowing one slowdown in PolyBench (3DCONV becomes 2.53 slower), and two in MgBench (collinear-list, 1.18 times slower, and chollesky, 2.58). On the other hand, dawncc shows slowdowns in 16 programs on Setup 1 and 15 on Setup 2, several of them being more than 100 times slower.

If setups 1 and 2 give similar results, setup 3 provides us with a very different landscape. For instance, 2DCONV from PolyBench was not modified by Etino on Setup 1, and it was slowed down by dawncc there. However, on Setup 3, Etino schedules a core convolution computation on the GPU, obtaining a speedup of 43%, as did dawncc. On this Setup, Etino obtains 8 speedups - against 9 of dawncc. On the other hand, it only slows two programs down, both from MgBench (str-matching by 48%, and vector-product by 44%), while dawncc slows down 10 programs. This evidence allows us to answer our second research question positively.

To put the speedups seen in Figure 9 in perspective, we have re-written 2MM using BLAS routines, in the same way as we have done to the program in Figure 1 (Section 2). We chose 2MM because, in our opinion, it was the easiest benchmark to adapt to use BLAS. As in Section 2, we test the programs with square matrices having 3,000 rows, using the 12-core machine available in Setup 1. The sequential program, compiled with `pgcc -fast`, runs in 624.60 seconds (in one CPU core). The program optimized with Etino runs in 3.30 seconds (in the Nvidia GPU). Using BLAS – the ATLAS version – we reach 3.50 seconds (in the 12 Intel cores), and using cuBLAS,

[10]We shall omit standard error bars from these figures, because they tend to disappear in the log-scale. The largest variation we have observed in any five runs among any of our 33 benchmark was under 0.6%.

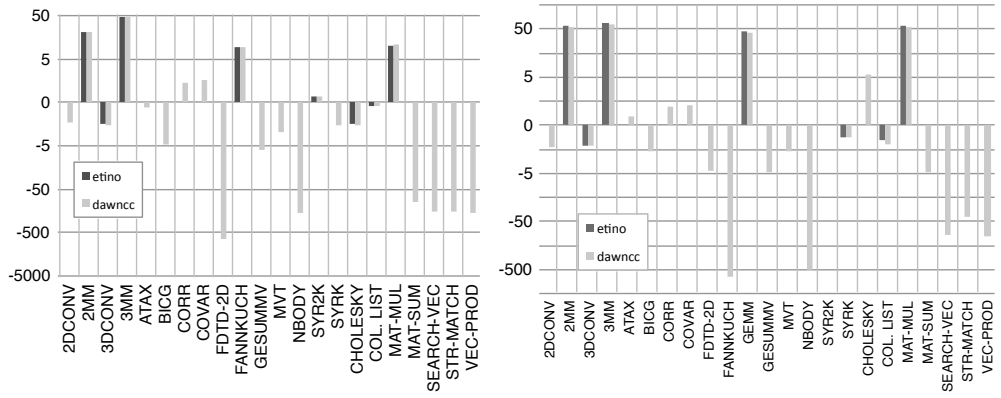

Fig. 9. (Left) Speedups obtained by (i) Etino and (ii) dawncc against original, i.e., sequential code, on Setup 1. (Right) Speedups obtained by (i) Etino and (ii) dawncc against original, i.e., sequential code, on Setup 2.

Table 2. Using Etino as a guiding tool. The annotations suggested by Etino when used with dawncc have been ported to ppcg. "Unguided" is the code produced by ppcg when every parallel loop is marked as a GPU kernel. "Guided" is the code produced with only the annotations that Etino has deemed profitable.

Benchmark	Unguided	Guided	Speedup	**Benchmark**	Unguided	Guided	Speedup
PB/2DCONV	0.823s	0.053s	15.5x	PB/2MM	2.344s	2.564s	0.9x
PB/3MM	2.997s	3.294s	0.9x	PB/ATAX	0.793s	0.034s	23.3x
PB/BICG	0.8s	0.032s	25x	PB/CORR	0.827s	0.831s	1x
PB/COVAR	0.824s	0.841s	1x	PB/GEMM	1.628s	1.858s	0.9x
PB/GESUMMV	0.793s	0.028s	28.3x	PG/GRAMSCHM	1.486s	17.731s	0.1x
PB/MVT	0.901s	0.457s	2x	PB/SYR2K	8.701s	8.825s	1x
PB/SYRK	0.955s	0.984s	1x	PB/SYRK_M	0.85s	0.867s	1x

we get 1.60 seconds (using the Nvidia GPU). Given cuBLAS's reputation, we believe that this last number is very close to the best result we could obtain in Setup 1. Thus, Etino has been able to reduce by almost 190x the runtime of 2MM, being close to staying within 2x the performance of the hand-optimized program, without any user intervention.

5.3 RQ3: Guiding Tool

In Section 5.2, Etino has been used in a fully automatic way: no intervention from users was necessary to produce our results. Nevertheless, Etino can also be used as a guiding tool, which supports programmers in deciding where to run each part of his or her code. To demonstrate this possibility, we have used Etino in tandem with the ppcg compiler [Verdoolaege et al. 2013]. In principle, it is possible to combine Etino with ppcg to produce code automatically, as we have done with dawncc. However, this would require some familiarity with the internals of ppcg, which must be adapted to read feedback from Etino. Thus, in this experiment we adopt the following methodology: we use the Etino-dawncc-pgcc[11] combination to annotate code automatically, and then port these annotations to ppcg manually.

Figure 2 shows the results of this experiment. We are using the "Setup 1" architecture, and we restrict ourselves to the PolyBench programs. Compiling the other benchmarks with ppcg would

[11]We ask the reader to be specially careful to avoid confusing pgcc, the PGI compiler, with ppcg, the polyhedral compiler of Verdoolaege *et al.*

require the generation of code to invoke functions within kernels, something that this compiler seems not to do at this point. In Figure 2 we are comparing the runtime of two variations of PolyBench: one marks as kernels every loop that is embarrassingly parallel. This includes all the initialization loops, for instance. The other marks as kernels only those loops that Etino has deemed adequate for the GPU. In four benchmarks, 2DCONV, BICG, GESUMMV and ATAX, Etino's decision was to not use the GPU at all. In fact, as we observed, we get substantial slowdowns if we run these programs on the GPU.

We got a few slowdowns by using Etino: the biggest happened in GRAMSCHM – an implementation of the Gram-Schmidt orthonormalising method. In this case, Etino has incorrectly hindered annotations. This imprecision is due to the fact that Etino has been tuned with pgcc, the PGI compiler. Performing this tuning with ppcg is not possible, because annotations are inserted manually. The tuning of a cost model, as explained in Section 4, requires hundreds of iterations of simulated annealing. Each iteration involves the generation of code and its execution. In the particular case of GRAMSCHM, compiling it with pgcc does not yield good GPU code, contrary to what we have observed with ppcg. Similarly, for 3MM, 2MM and GEMM, Etino made assumptions on the quality of the GPU code generated based on the training done with pgcc, and decided to not run some small functions on the GPU. The equivalent GPU code generated for these functions by ppcg, however, runs faster, leading to the small slowdowns observed for these benchmarks. Had we tuned Etino directly with ppcg, then these problems would not have occurred. Thus, in the effort to answer the third research question, we drew two conclusions. First, the construction of a cost model depends also on the compiler, not only on the architecture. Second, even when used in cross-compilation mode, Etino is effective as a guiding tool: we got four large speedups, against one significant slowdown.

5.4 RQ4: Sensitivity to Input Size

In this section, we evaluate how variations in input size affect Etino. This is relevant because to be fully static, Etino disregards input size. We have tested several different input sizes for each benchmark on Setup 1. Table 3 shows speedups and slowdowns produced by Etino on all benchmarks in which it uses the GPU, and for dawncc it shows the largest slowdown obtained in each benchmark suite. As before, positive numbers denote speedups, whereas negative numbers denote slowdowns (i.e. -2x means "two times slower"). Previously, we had used the default input size from each benchmark; Table 3 shows results for 4 input sizes: small, medium, large and huge. The smallest input makes the sequential program run in about 100ms; huge input is the largest we were able to run on Setup 1, given the available memory, or the smallest size that makes the benchmark run in more than 5 minutes (in FannKuch). In this experiment, Etino's schedulings produced slowdowns for smallest inputs, where the absolute impact is the smallest. However, when input size grows, the speedup tends to grow as well. In the two benchmarks in which Etino always got slowdowns, this ratio tends to shrink, and for huge inputs it is less than 1% in `collinear-list` and about 18% in 3DCONV. On the other hand, dawncc produces several slowdowns that behave quite differently. In FannKuch, for instance, slowdown grows with input size. In `search-vec`, in which data transfer is the major overhead when running on the GPU, slowdown keeps at about 102x. In MVT, slowdown also shrinks with input size, but it is still 318% on the huge input. Therefore, Etino's schedulings showed a much more desirable behavior with varying input size: slowdowns happen when the absolute impact is minimal. Thus, answering our fourth research question: small inputs, contrary to large inputs, contribute negligibly to the runtime of our benchmarks and bear little impact on Etino's performance.

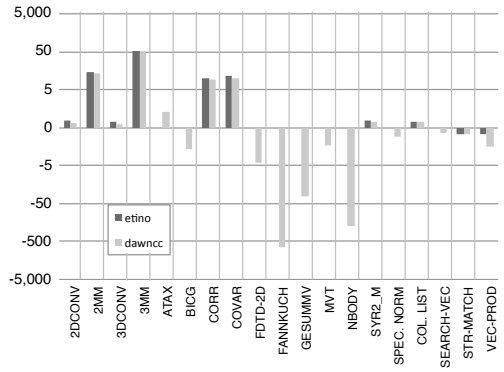

Fig. 10. Speedups obtained by (i) Etino and (ii) dawncc against original, i.e., sequential code, on Setup 3.

Table 3. Evolution of speedups/slowdowns on Setup 1 for several input sizes. For Etino, we show all benchmarks which were modified on this setup. For dawncc, we show benchmarks for which Etino suggests no modification, e.g,. code should run on the CPU only, but dawncc still runs on the GPU. Etino avoids the slowdowns observed in these programs.

Tool	Benchmark	Speedup on Small Input	Speedup on Medium Input	Speedup on Large Input	Speedup on Huge Input
etino	cholesky	-9.76x	-2.60x	-1.13x	1.06x
	2MM	-3.78x	20.10x	113.09x	214.13x
	3MM	-1.95x	29.04x	151.92x	258.41x
	SYR2K	-2.12x	1.28x	1.73x	1.79x
	GEMM	-5.77x	10.64x	70.03x	167.83x
	3DCONV	-37.00x	-11.66x	-2.79x	-1.18x
	collinear-list	-2.65x	-1.22x	1x	1x
dawncc	FannKuch	-119.08x	-190.66x	-346.11x	-465.79x
	search-vector	-103.48x	-103.42x	-105.75x	-102.45x
	MVT	-33.37x	-12.11x	-5.16x	-3.18x

5.5 RQ5: The Importance of Interprocedurality

Algorithm 1 uses a context of depth one to take scheduling decisions. We can demonstrate empirically that using contexts, even of depth one, has two important advantages, at least in our setups. First, some functions, when observed in isolation, should run on the CPU, but when nested, should run on the GPU. This combination increases the ratio of computation per memory transfer in the full application; hence, amortizing the cost of transferring data. Second, there are situations in which the same function should run on the CPU and on the GPU, in actual programs. An example of such situation was given in Section 2.

These advantages are unlikely to surface on small benchmarks, such as PolyBench, MgBench or BenchGame. Thus, to demonstrate them, we have applied Etino onto three different algorithms whose implementation has been taken from [Da Mata et al. 2013]: k-means clustering, kNN classification, and Frequent Itemset Mining (apriori). These are general purpose applications, more complex and more irregular than the other benchmarks that we have used; hence, they exercise more aspects of our algorithm. However, even though Algorithm 1 handles them automatically, dawncc is not able to annotate them without user intervention. They have been written in a

functional style based on linked lists. We had to replace these lists by arrays to enable automatic annotation.

K-means and Frequent Itemsets illustrate the benefits of our context representation. These benchmarks have functions that, depending on their callee, Etino assigns to the CPU or to the GPU in all the three setups. For the largest inputs available for each program, we got, after Etino's parallelization, speedups of 14.6x in K-means, and 6.2x in Frequent Itemsets on Setup 1. Other setups led to similar gains. The KNN classifier is also irregular and difficult to parallelize automatically. Etino has assigned three, out of the six functions in this benchmark, to the GPU. Each function is invoked at only one calling site; hence, no function can be assigned to different processors depending on context. Yet, if we were to analyze these functions individually, without taking the calling stack into consideration, only one of them would find its way to the GPU. In KNN, Etino obtains the best speedups observed in the DataMining suite: with a 10-dimensional dataset of $100,000$ points, answering $10,000$ queries with $k = 31$, the parallel version is $\approx$ 75x faster than its sequential counterpart on Setup 1.

6 RELATED WORK

The work that we have presented in this paper touches different facets of compiler research, related to scheduling, autotuning and static analyses. In this section, we explain how we compare against some of this previous work.

Mapping Program Parts into Heterogeneous Processors. Recent years have seen many new approaches for code placement in heterogeneous architectures. Figure 11 provides the reader with a perspective on where Etino stands, when compared to some of this previous art. We have grouped different technologies according to how they answer the following questions:

- *Does the method use runtime information?* In this case, we have completely static or dynamic approaches, and hybrid techniques. Static approaches tend to incur less runtime overhead; dynamic techniques tend to be more precise, as they benefit from knowledge only available once the program executes. Hybrid approaches either instrument programs to take application's workload into consideration [Margiolas and O'Boyle 2016; Piccoli et al. 2014], or use a profiler to feedback runtime information to the compiler. Notice that, in this paper, runtime information is used to calibrate a cost model. However, once the model is built, it works for every application in a given architecture, without the need of a profiling phase.
- *Does the method require intervention from users?* In this category we have guided or fully automatic approaches. Guided techniques force the user to touch the code that must be optimized. Usually, this modus operandi asks for some familiarity with the program. Users can carry out interventions by either coding the program using some specific library, *à la* StarPU, some programming language, such as OpenCL, by inserting annotations into the program, e.g. when using OpenACC, or by using a combination of these tasks, as the Swift programming language which requires manual annotations so that the compiler can automatically partition the application between client and server [Chong et al. 2007].
- *Does the method adapt to the target architecture?* Adaptive techniques can be customized to a given architecture. Customization might be achieved through code instrumentation [Piccoli et al. 2014], through a shadow process in charge of migrating tasks [Nishtala et al. 2017; Petrucci et al. 2015], or through a pre-computed model [Sim et al. 2012]. We use machine learning to find this model.

It is possible to expand the taxonomy seen in Figure 11 into further directions. For instance, some approaches target heterogeneous processors running the same instruction set, such as those that use big.LITTLE cores [Nishtala et al. 2017; Petrucci et al. 2015]. Others target processors that use different ISAs, such as CPUs and GPUs. This diversity is not accounted for by Figure 11.

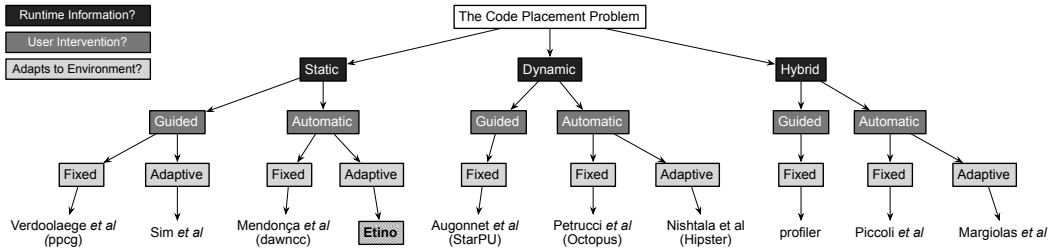

Fig. 11. The taxonomy of work related to the placement of code on heterogeneous architectures.

Nevertheless, we believe that the figure already provides the reader with an intuition on the novel aspects of Etino, given that our tool, to the best of our knowledge, stands alone as the first totally static, automatic and adaptive method to place computations in heterogeneous hardware.

We believe that this ability to handle programs as they are, without modification, is one of Etino's big assets. To support this statement, we have compared it with a guided approach, with which we had some familiarity: StarPU [Augonnet et al. 2011]. Before we tell our anecdote, we warn the reader that we are comparing two very different systems, which were designed and implemented with varying purposes. We have re-written one of the DataMining benchmarks, DM/kmeans using StarPU's library. Given our limited experience with this system, we opted for the most straightforward implementation: we create one task to find the nearest centroid of a given point. To perform this re-engineering, we had to modify one file, which had 120 lines of code. Such intervention resulted in a file with 219 lines: 10 lines modified from the original code, and 109 lines added to it. This process of converting a file to use the StarPU API is not difficult, but it is tedious. We had to: (i) write versions of the task for the different available processors; (ii) write some code to initialize these tasks; and (iii) write code to invoke the tasks asynchronously. The original version of the benchmark runs in $4m12.250 \pm 0.5$ secs (Setup 2); the version produced by Etino-pgcc runs in 16.506 ± 0.5 secs; and the version converted to StarPU runs in $1m50.443 \pm 0.5$ secs. The fact that StarPU does not outperform Etino in this case does not mean that Etino's approach is more efficient. StarPU cannot see that the several tasks are doing the same thing; hence, it cannot bundle them together in the GPU, to benefit from the SIMD nature of that hardware. We could raise the granularity of tasks, grouping them manually. But, in this case, the implementation becomes more complex. Etino, in turn, does not produce code that runs concurrently on the CPU and GPU, something that StarPU does.

Autotuning-Based Approaches to Calibrate Costs. Much work has been done to design new analytical cost models that characterize heterogeneous devices [Sim et al. 2012; Song et al. 2013], and tune these cost models so to enable better placement of computation in said devices [Bajrovic et al. 2016; Basu et al. 2013; Muralidharan et al. 2016]. We chose to tune our cost model using simulated annealing, because this technique is a black-box method. It suits any cost model and objective function the user chooses to use. Using simulated annealing to tune cost models for heterogeneous devices is not a new idea. Lin *et al.* [Lin et al. 2016] have applied it, in combination with a genetic algorithm, to configure the dimension of a GPU kernel, in number of threads per block or number of registers per threads. Along another direction, Hartono *et al.* [Hartono et al. 2009] have used the Orio autotuner, which uses simulated annealing among other techniques, to find a good mix of optimizations for high-performance kernels. In our case, kernels are always the same – the decision that we take is where to run them. Thus, both Lin *et al.*'s and Hartono *et al.*'s [Hartono et al. 2009] approaches are complementary to ours.

Context Sensitive Analyses. There exists a vast literature about context sensitive analyses. Aho *et al.* [Aho et al. 2006, Ch.12.5] provide an overview on this topic. Borrowing some terminology from Might *et al.* [Might et al. 2010], we use a top-two frame abstraction, keeping track of the caller of each function, but not the "caller-of-the-caller" for instance. As pointed by Might *et al.*, this method naturally yields a polynomial time algorithm to enumerate contexts. The disadvantage of this approach is the following: if we assume two call strings like $f_1 \to g \to h$, and $f_2 \to g \to h$, then our algorithm only considers the location of g when determining the location of h. In other words, neither f_1 nor f_2 will have a direct influence on the scheduling decision of h. They still have indirect influence, as they determine the location of g. Therefore, by choosing shallow contexts – an approach widely adopted in industrial quality compilers [Lhoták and Hendren 2006] – we are trading precision for scalability.

7 CONCLUSION

This paper has presented a compiler-based algorithmic framework to schedule code in heterogeneous architectures. This framework consists of a static scheduling algorithm (seen in Section 3); and a technique based on simulated annealing to tune the cost model that said algorithm uses (seen in Section 4). We believe that this combination of techniques is the first purely compiler-based approach to schedule code in heterogeneous devices. Given the raising popularity of such architectures, and the recent advances made in terms of static analyses, we expect to see other approaches of similar goal in the coming years.

During the implementation of Etino, we have faced several questions that, at that time, we left unanswered. We believe that they constitute interesting problems, which we leave open for researchers. Firstly, we leave the work of adapting Etino to generate code that uses multiple processors concurrently. We have not explored such a possibility in this work; however, recent forays in the field of dynamic scheduling seem to indicate that concurrency is not only possible, but also highly profitable in a heterogeneous architecture [Wen and O'Boyle 2017]. Another problem that we think deserves some attention concerns input size. The lack of input size information at compile time may be compensated by making Etino insert checks into the program to guide decisions at run time. Finally, a fair question that this paper touched, but did not answer, concerns granularity: is the granularity of a function a good choice for static code placement in heterogenous architectures? Loops and Single-Entry-Single-Exit regions are natural scheduling units, which, in addition to functions, a tool like Etino could consider.

REFERENCES

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison Wesley, Boston, MA, USA.

Péricles Alves, Fabian Gruber, Johannes Doerfert, Alexandros Lamprineas, Tobias Grosser, Fabrice Rastello, and Fernando Magno Quintão Pereira. 2015. Runtime Pointer Disambiguation. *SIGPLAN Not.* 50, 10 (2015), 589–606.

M. Amini, C. Ancourt, F. Coelho, B. Creusillet, S. Guelton, F. Irigoin, P. Jouvelot, R. Keryell, and P. Villalon. 2012. *PIPS Is not (only) Polyhedral Software.* Technical Report. IMPACT.

Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. 2010. A View of Cloud Computing. *CACM* 53, 4 (2010), 50–58.

Cedric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-Andre Wacrenier. 2011. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurr. Comput. : Pract. Exper.* 23, 2 (2011), 187–198.

Eduard Ayguadé, Rosa M. Badia, Francisco D. Igual, Jesús Labarta, Rafael Mayo, and Enrique S. Quintana-Ortí. 2009. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In *Euro-Par*. Springer, Heidelberg, Germany, 851–862.

David Bacon, Rodric Rabbah, and Sunil Shukla. 2013. FPGA Programming for the Masses. *Queue* 11, 2 (2013), 40:40–40:52.

Enes Bajrovic, Robert Mijakovic, Jiri Dokulil, Siegfried Benkner, and Michael Gerndt. 2016. Tuning OpenCL Applications with the Periscope Tuning Framework. In *HICSS*. IEEE, New York, NY, USA, 5752–5761.

Rajkishore Barik, Naila Farooqui, Brian T. Lewis, Chunling Hu, and Tatiana Shpeisman. 2016. A Black-box Approach to Energy-aware Scheduling on Integrated CPU-GPU Systems. In *CGO*. ACM, New York, NY, USA, 70–81.

Muthu Manikandan Baskaran, J. Ramanujam, and P. Sadayappan. 2010. Automatic C-to-CUDA Code Generation for Affine Programs. In *CC*. Springer, Heidelberg, Germany, 244–263.

Protonu Basu, Mary Hall, Malik Khan, Suchit Maindola, Saurav Muralidharan, Shreyas Ramalingam, Axel Rivera, Manu Shantharam, and Anand Venkat. 2013. Towards Making Autotuning Mainstream. *Int. J. High Perform. Comput. Appl.* 27, 4 (2013), 379–393.

Richard Ernest Bellman. 1957. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA.

Carlo Bertolli, Samuel Antao, Alexandre Eichenberger, Kevin O'Brien, Zehra Sura, Arpith Jacob, Tong Chen, and Olivier Sallenave. 2014. Coordinating GPU Threads for OpenMP 4.0 in LLVM. In *LLVM-HPC*. IEEE, New York, NY, USA, 12–21.

Ting Cao, Stephen M. Blackburn, Tiejun Gao, and Kathryn S. McKinley. 2012. The Yin and Yang of power and performance for asymmetric hardware and managed software. In *ISCA*. ACM, New York, NY, USA, 225–236.

Vlado Cerny. 1985. Thermodynamical approach to the traveling salesman problem. *Journal of Optimization Theory and Applications* 45, 1 (1985), 41–51.

Stephen Chong, Jed Liu, Andrew C Myers, Xin Qi, Krishnaprasad Vikram, Lantian Zheng, and Xin Zheng. 2007. Secure web applications via automatic partitioning. *ACM SIGOPS Operating Systems Review* 41, 6 (2007), 31–44.

Emilio G. Cota, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P. Carloni. 2015. An Analysis of Accelerator Coupling in Heterogeneous Architectures. In *DAC*. ACM, New York, NY, USA, 202:1–202:6.

Leonardo Luiz Padovani Da Mata, Fernando Magno QuintãO Pereira, and Renato Ferreira. 2013. Automatic Parallelization of Canonical Loops. *Sci. Comput. Program.* 78, 8 (2013), 1193–1206.

Gregory F. Diamos and Sudhakar Yalamanchili. 2008. Harmony: An Execution Model and Runtime for Heterogeneous Many Core Systems. In *HPDC*. ACM, New York, NY, USA, 197–200.

Michael Garland. 2008. Parallel Computing Experiences with CUDA. *IEEE Micro* 28 (2008), 13–27. Issue 4.

Alan Gibbons. 1988. *Efficient Parallel Algorithms*. Cambridge University Press, Cambridge, UK.

Serge Guelton, Mehdi Amini, and Béatrice Creusillet. 2012. Beyond Do Loops: Data Transfer Generation with Convex Array Regions. In *LCPC*. Springer, Heidelberg, Germany, 249–263.

Bhargav S. Gulavani and Sumit Gulwani. 2008. A Numerical Abstract Domain Based on Expression Abstraction and Max Operator with Application in Timing Analysis. In *CAV*. Springer, Heidelberg, Germany, 370–384.

Albert Hartono, Boyana Norris, and P. Sadayappan. 2009. Annotation-based Empirical Performance Tuning Using Orio. In *IPDPS*. IEEE, New York, NY, USA, 1–11.

Ivan Jibaja, Ting Cao, Stephen M. Blackburn, and Kathryn S. McKinley. 2016. Portable Performance on Asymmetric Multicore Processors. In *CGO*. ACM, New York, NY, USA, 24–35.

David Kirk. 2007. NVIDIA Cuda Software and GPU Parallel Computing Architecture. In *ISMM*. ACM, New York, NY, USA, 103–104.

Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. 2012. PyCUDA and PyOpenCL: A Scripting-based Approach to GPU Run-time Code Generation. *Parallel Comput.* 38, 3 (2012), 157–174.

Ron Kohavi. 1995. A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection. In *IJCAI*. Morgan Kaufmann, Burlington, MA, USA, 1137–1143.

Ahmad Lashgar, Alireza Majidi, and Amirali Baniasadi. 2014. IPMACC: Open Source OpenACC to CUDA/OpenCL Translator. *CoRR* abs/1412.1127 (2014), 1–9.

Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*. IEEE, New York, NY, USA, 75–88.

Ondřej Lhoták and Laurie Hendren. 2006. Context-Sensitive Points-to Analysis: Is It Worth It?. In *CC*. Springer, Heidelberg, Germany, 47–64.

Chih-Sheng Lin, Shih-Meng Teng, and Pao-Ann Hsiung. 2016. Auto-tuning for GPGPU Applications Using Performance and Energy Model. *J. Syst. Archit.* 62, C (2016), 40–53.

Christos Margiolas and Michael F. P. O'Boyle. 2016. Portable and Transparent Software Managed Scheduling on Accelerators for Fair Resource Sharing. In *CGO*. ACM, New York, NY, USA, 82–93.

Gleison Mendonça, Breno Guimarães, Péricles Alves, Márcio Pereira, Guido Araújo, and Fernando Magno Quintão Pereira. 2017. DawnCC: Automatic Annotation for Data Parallelism and Offloading. *TACO* 14, 2 (2017), 13:1–13:25.

Gleison Mendonça, Breno Guimaraes, Péricles Alves, Márcio Pereira, Guido Araújo, and Fernando Magno Quint ao Pereira. 2016. Automatic Insertion of Copy Annotation in Data-Parallel Programs. In *SBAC-PAD*. ACM, New York, NY, USA, 1–8.

Matthew Might, Yannis Smaragdakis, and David Van Horn. 2010. Resolving and Exploiting the k-CFA Paradox: Illuminating Functional vs. Object-oriented Program Analysis. In *PLDI*. ACM, New York, NY, USA, 305–315.

Saurav Muralidharan, Amit Roy, Mary Hall, Michael Garland, and Piyush Rai. 2016. Architecture-Adaptive Code Variant Tuning. *SIGPLAN Not.* 51, 4 (2016), 325–338.

John Nickolls and William J. Dally. 2010. The GPU Computing Era. *Micro* 30 (2010), 56–69.

John Nickolls and David Kirk. 2009. *Graphics and Computing GPUs. Computer Organization and Design, (Patterson and Hennessy)* (4th ed.). Elsevier, Amsterdam, Netherlands, Chapter A, A.1 – A.77.

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 2005. *Principles of program analysis.* Springer, Heidelberg, Germany.

Rajiv Nishtala, Paul Carpenter, Vinicius Petrucci, and Xavier Martorell. 2017. Hipster: Hybrid Task Manager for Latency-Critical Cloud Workloads. In *HPCA*. ACM, New York, NY, USA, 1–11.

Cedric Nugteren and Henk Corporaal. 2014. Bones: An Automatic Skeleton-Based C-to-CUDA Compiler for GPUs. *TACO* 11, 4 (2014), 35:1–35:25.

openacc.org. 2012. *The OpenACC Application Programming Interface.* Technical Report 2.5. CAPS, Cray, Nvidia and PGI.

Vinicius Petrucci, Michael A. Laurenzano, John Doherty, Yunqi Zhang, Daniel Mossé, Jason Mars, and Lingjia Tang. 2015. Octopus-Man: QoS-driven task management for heterogeneous multicores in warehouse-scale computers. In *HPCA*. ACM, New York, NY, USA, 246–258.

PGI. 2016. Compiler User's Guide. (2016). http://www.pgroup.com/ doc/pgiug.pdf.

Guilherme Piccoli, Henrique N. Santos, Raphael E. Rodrigues, Christiane Pousa, Edson Borin, and Fernando M. Quintão Pereira. 2014. Compiler Support for Selective Page Migration in NUMA Architectures. In *PACT*. ACM, New York, NY, USA, 369–380.

Radu Rugina and Martin C. Rinard. 2005. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *TOPLAS* 27, 2 (2005), 185–235.

Diogo Sampaio, Rafael Martins de Souza, Sylvain Collange, and Fernando Magno Quintão Pereira. 2014. Divergence Analysis. *ACM Trans. Program. Lang. Syst.* 35, 4, Article 13 (2014), 36 pages.

Mahadev Satyanarayanan. 2011. Mobile Computing: The Next Decade. *SIGMOBILE* 15, 2 (2011), 2–10.

Olin Shivers. 1988. Control-Flow Analysis in Scheme. In *PLDI*. ACM, New York, NY, USA, 164–174.

Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim, and Richard Vuduc. 2012. A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications. In *PPoPP*. ACM, New York, NY, USA, 11–22.

Amit Kumar Singh, Muhammad Shafique, Akash Kumar, and Jörg Henkel. 2013. Mapping on Multi/Many-core Systems: Survey of Current and Emerging Trends. In *DAC*. ACM, New York, NY, USA, Article 1, 10 pages.

Shuaiwen Song, Chunyi Su, Barry Rountree, and Kirk W. Cameron. 2013. A Simplified and Accurate Model of Power-Performance Efficiency on Emergent GPU Architectures. In *IPDPS*. IEEE, New York, NY, USA, 673–686.

Victor Hugo Sperle Campos, Péricles Rafael Alves, Henrique Nazaré Santos, and Fernando Magno Quintão Pereira. 2016. Restrictification of Function Arguments. In *CC*. ACM, New York, NY, USA, 163–173.

John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Des. Test* 12, 3 (2010), 66–73.

Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral Parallel Code Generation for CUDA. *TACO* 9, 4 (2013), 54:1–54:23.

Yuan Wen and Michael F.P. O'Boyle. 2017. Merge or Separate?: Multi-job Scheduling for OpenCL Kernels on CPU/GPU Platforms. In *GPGPU-10*. ACM, New York, NY, USA, 22–31.

John Whaley and Monica S. Lam. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*. ACM, New York, NY, USA, 131–144.

Michael Wolfe. 1996. *High Performance Compilers for Parallel Computing* (1st ed.). Adison-Wesley, Boston, MA, USA.

Wei Wu, Aurelien Bouteiller, George Bosilca, Mathieu Faverge, and Jack Dongarra. 2015. Hierarchical DAG Scheduling for Hybrid Distributed Systems. In *IPDPS*. IEEE, New York, NY, USA, 156–165.

Mohamed Zahran. 2016. Heterogeneous Computing: Here to Stay. *Queue* 14, 6 (2016), 40:31–40:42.

Hui Zeng, Matt Yourst, Kanad Ghose, and Dmitry Ponomarev. 2009. MPTLsim: A Cycle-accurate, Full-system Simulator for x86-64 Multicore Architectures with Coherent Caches. *SIGARCH Comput. Archit. News* 37, 2 (2009), 2–9.