Acceleration Targets: A Study of Popular Benchmark Suites

Lisa Wu and Martha A. Kim,

Department of Computer Science, Columbia University

{lisa,martha}@cs.columbia.edu

1 Introduction

Using dark silicon [14, 3] to deploy specialized accelerators is an idea that is gaining traction in the architecture community [4, 5, 6, 1]. The underlying rationale is that specialized hardware and its attendant efficiency is the most effective way to draw performance in the anticipated power-limited scenarios. Given the cost associated with designing, verifying, and deploying an accelerator, conventional wisdom dictates that a particular operation becomes an economical and realistic acceleration target when it is used across a range of applications.

In this study, we survey a set of popular benchmark suites, assessing the potential of several acceleration targets within them. In particular, we explore the following three questions:

- Do the benchmarks exhibit any common functionality at or above the function level?

- What impact does the language or programming environment have on the potential acceleration of a suite of applications?

- How many unique accelerators would be required to see benefits across a particular benchmark suite? Does this change across suites and source programming languages?

2 Methodology

To explore these questions, we profile four benchmark suites: SPEC2006 (C) [11], SPECJVM (Java) [12], Dacapo (Java) [2], and Unladen-Swallow (Python) [13]. Each source language provides a slightly different set of potential acceleration targets. For example, SPEC2006 is written in C and offers two target granularities: individual functions or entire applications. In contrast, a Java benchmark offers three granularities: methods, classes (i.e., all of the methods for a particular class), and entire applications. We classify each of these potential targets as *fine*, *medium*, or *coarse* granularity according to Table 1.

For each class of acceleration targets, we sort the targets by decreasing execution time across the entire benchmark suite. Assuming that building an accelerator for a particular target (1) provides infinite speedup of the target, and (2) incurs no data or control transfer overhead upon invocation or return, we compute an upper bound on the speedup of the overall suite for the most costly target(s). We repeat this

Benchmark Suite	Granularity		
	fine	*medium*	*coarse*
SPEC2006	function	–	application
SPECJVM	method	class	package
DACAPO	method	class	package
UNLADEN-SWALLOW	function	–	object

Table 1: Acceleration Targets for Each Suite

analysis for each target granularity in each benchmark suite, as outlined in Table 1.

3 Results and Analysis

Our results show that popular benchmark suites exhibit minimal functional level commonality. For example, it would take 500 unique, idealized accelerators to gain a 48X speedup across the SPEC2006 benchmark suite. The C code is simply not modular for acceleration, and few function accelerators can be re-used across a range of applications. For benchmarks written in Java, however, we see more commonality as language level constructs such as classes encapsulate operations for easy re-use. The question remains whether building 20 accelerators for SpecJVM or 50 accelerators for Dacapo is worth the investment for the 10X speedups to be had. In the particular Python benchmark suite we used, we found that the applications made minimal use of the built-ins (e.g., dict or file) resulting in very minimal opportunity for acceleration beyond the methods themselves. Our intuition is that this may be an artifact of a computationally-oriented performance benchmark suite, and is likely not reflective of the overall space of Python workloads.

4 Conclusion

Our analyses of SPEC2006 confirm what C-cores [14], ECO-cores [10], and DYSER [5] also found: that when accelerating unstructured C code, the best targets are large swaths of highly-application-specific code. Our Java analyses indicate some hope for common acceleration targets in classes, though the advantage of targeting classes over individual methods appears modest. Across the board, our data show that filling dark silicon with specialized accelerators will require systems containing tens or even hundreds of accelerators. In light of this, we believe the infrastructure associated with these accelerators (e.g., networks, memory models [7, 9, 8], and toolchains[14]) will only increase in importance.
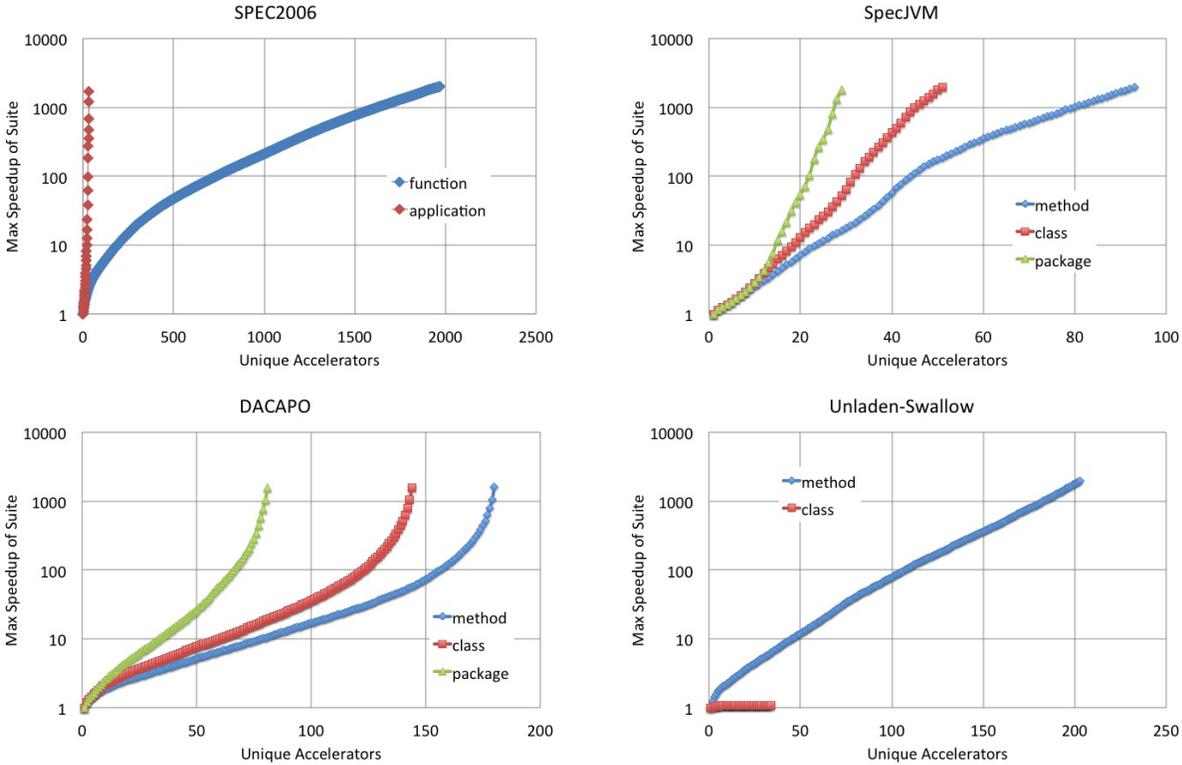
Figure 1: Max speedup of benchmark suite for {fine, medium, and coarse}-granular acceleration targets.

References

[1] C. Cascaval et al. A taxonomy of accelerator architectures and their programming models. *IBM Journal of Research and Development*, 54(5):1–10, 2010.

[2] The Dacapo Benchmark Suite. `http://dacapobench.org/`.

[3] H. Esmaeilzadeh et al. Dark silicon and the end of multicore scaling. In *ISCA*, pages 365–376, 2011.

[4] N. Goulding-Hotta et al. GreenDroid : A mobile application processor for a future of dark silicon. *IEEE Micro*, 31(2):86–95, 2011.

[5] V. Govindaraju et al. Dynamically specialized datapaths for energy efficient computing. In *HPCA*, pages 503–514, 2011.

[6] R. Hameed et al. Understanding sources of inefficiency in general-purpose chips. In *ISCA*, pages 37–47, June 2010.

[7] J. Kelm et al. Cohesion: a hybrid memory model for accelerators. In *ISCA*, pages 429–440, June 2010.

[8] M. Lyons et al. The accelerator store framework for high-performance, low-power accelerator-based systems. *IEEE Computer Architecture Letters*, 9(2):53–56, Feb. 2010.

[9] B. Saha et al. Programming model for a heterogeneous x86 platform. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2009.

[10] J. Sampson et al. Efficient complex operators for irregular codes. In *Proceedings of the 17th International Symposium on High Performance Computer Architeture (HPCA)*, pages 491–502. ACM, Feb 2011.

[11] Standard Performance Evaluation Corporation. `http://www.spec.org/cpu2006/`.

[12] Standard Performance Evaluation Corporation. `http://www.spec.org/jvm2008/`.

[13] Unladen Swallow Benchmarks. `http://code.google.com/p/unladen-swallow/wiki/Benchmarks`.

[14] G. Venkatesh et al. Conservation cores: reducing the energy of mature computations. In *ASPLOS*, pages 205–218, Mar. 2010.