Resource Allocation for Hardware Implementations of Map

Richard Townsend Martha A. Kim Stephen A. Edwards
Columbia University
{rtownsend,martha,sedwards}@cs.columbia.edu

Abstract

The map operation, in which a function is applied independently to each element in a collection to produce a new collection, appears in many settings and is easy to parallelize. While a straightforward implementation in hardware will consist of multiple functional units with buffers to balance variable execution times, the best trade-off between these two components is not obvious. Too many buffers wastes resources that could otherwise perform computation; too few buffers causes functional units to lie idle waiting for empty buffers. Our work considers this abstract problem, derives worst-case workload distributions, then shows how to trade functional units for buffers to maximize throughput. Our results can be used by designers and compilers alike to produce efficient parallel implementations of map.

1. Introduction

We propose techniques for efficiently implementing in hardware the common, "embarrassingly parallel" map operation, in which a function is applied independently to each element in a collection to produce a new collection. This operation, familiar to functional programmers since at least the 1960s [3], now appears in many languages and settings.

Expressing operations as a map helps to expose parallelism. Consider a brute-force computation to display the Mandelbrot set [2]. While a natural implementation in an imperative language might consist of three nested loops (one for each dimension of the image and a third to perform the complex iteration), opportunities for parallelizing become clearer if the computation is thought of as a map: a set of complex coordinates (each pixel in the image) fed to a function that tests, iteratively, whether that particular coordinate is part of the Mandelbrot set.

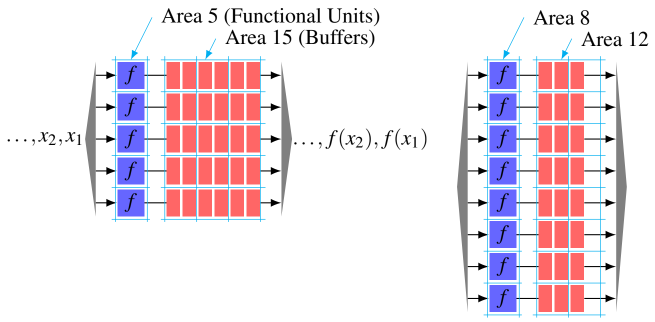


Figure 1. Parallel map circuits. These consume a sequence of inputs $x_1, x_2, \ldots$, distribute them to functional units that apply a function f to each input, buffer the results, and recombine the results in order $f(x_1), f(x_2), \ldots$. Under an area budget of 20 ($A = 20$) with each buffer half the size of a functional unit ($\beta = 0.5$), these two configurations are optimal—they produce the highest possible throughput under the worst possible distribution of processing times required to evaluate the $f(x_i)$.

We consider the problem of implementing a particular map operation in hardware given finite resources. While also commonly implemented in software or even parallel software such as Google's MapReduce [1], we consider high-speed parallel hardware implementations. Figure 1 illustrates the structure of the systems we are considering: a sequence of data $(x_1, x_2, \ldots)$ arrive on the left and are distributed among the functional units. Each functional unit computes a result that it places in its output buffer. Finally, these results $(f(x_1), f(x_2), \ldots)$ are collected from the buffers and sent out in the order in which they arrived. In a conventional MapReduce framework, the inputs and outputs to the map operation are not ordered. We focus on implementing map in the functional sense, which entails an operator being applied to an ordered list.

Our main focus is on the trade-off between adding more functional units (components that perform the operation on a single data element) and adding more buffering. Buffering can help maintain functional unit utilization when element processing times vary widely. When the per-element processing time is constant, the need for buffering is limited since a simple round-robin scheduler will ensure that

ASBD '14, June 14, 2014, Minneapolis, MN, United States.
Copyright © 2014 ACM . . . $15.00.
http://dx.doi.org/10.1145/

all the functional units are always occupied. But when the per-element time varies widely and unpredictably, a simple round-robin strategy will leave many units starved for data when they must wait for another unit to finish processing more expensive data.

Others have considered related resource allocation problems, but not exactly this one. Purnaprajna et al. [4] propose a genetic algorithm approach, but their focus is general embedded system designs as opposed to the specific case of map. Yeung et al. [6], in their implementation of a MapReduce library supporting FPGAs and GPUs, mention how they allocated resources to functional units, but do not address buffer sizing. Shan et al. [5] present a MapReduce design for FPGAs and discuss resource allocation with respect to functional units, but they assume the simple case of constant per-element processing times, removing the need for buffering.

Here, we present a resource allocation methodology for both functional units and buffers that maximizes throughput. We explore how different allocations affect throughput under different constraints, including input job variation and the relative costs of buffers and functional units. This exploration provides us with insight on how the number of functional units and the size of their output buffers affects throughput.

We then discuss the worst-case throughput an implementation could experience. The situation leading to the worst case is formalized and empirically tested. Using the worst-case behavior as a lower bound on throughput, we empirically derive an optimal allocation of resources that maximizes throughput in the worst case.

We conclude with a comparison of maximal throughput allocations across different area budgets and relative buffer to functional unit costs. This comparison reveals that the worst-case throughput increases as the area budget increases and as the relative cost of buffers to functional units decreases.

2. Background

Our hardware implementation of map (Figure 1) applies a function f to an ordered list of data elements, $x_1, x_2, \ldots$. Input elements are distributed to functional units that compute $f(x)$ and store the result in a local FIFO buffer. If the buffer is full and unable to accept another element, the functional unit must stall until there is room. Although the buffers allow the functional units to finish out of order, the remainder of the circuit gathers the processed data elements from the buffers in order to reconstruct the input order, producing the output sequence $f(x_1), f(x_2), \ldots$.

To reason about resources, we introduce A, the total area to be distributed between unit-area functional units and buffer slots of area β. Thus, a system with N functional units and M buffer slots per unit must satisfy

$$N(1+\beta M) \leq A.$$

Figure 1 shows two configurations for the constraints $A = 20$ and $\beta = 0.5$. The left configuration corresponds to $N = 5$, $M = 6$; the right one is $N = 8$, $M = 3$. It turns out that these two configurations are optimal in the sense that they produce the highest throughput under worst-case workload distributions; we will explain all this below.

2.1 Workloads and Completion Time

We measure throughput as the time a system takes to finish computing a given input sequence. This completion time depends not just on the hardware configuration, but also on the input workload, i.e., the number, order, and processing time of the input elements. We write W for the sum of the processing times of all elements (e.g., the time to process them all on a single functional unit). Ideally, we would like to ensure that all functional units constantly do useful work, but this is not always possible. We write S for the total time all functional units spend stalling (waiting for space in their output buffers). For N functional units, the wall clock time to complete all processing is therefore

$$\frac{W+S}{N}.$$

In the best case, which can be attained with constant job completion time and negligible start-up and flushing overhead, the completion time approaches

$$\frac{W}{N}.$$

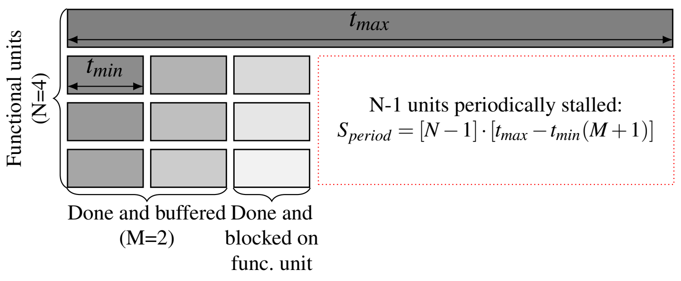
We focus instead on characterizing and compensating for the worst-case workload, which we discuss below.

2.2 Simulating Our Model

To evaluate the performance of our hardware map circuits, we constructed a C simulator to run input workloads through different system configurations. We construct a workload of size W by sampling a range of positive integers (each value represents the execution time for a particular data sample) until the sum of the sampled values is W. We used this method to craft workloads whose processing times follow uniform random, Gaussian, and other probabilistic distributions.

We also implemented a Mandelbrot set generator to model a specific kind of real-world workload. Different regions of the Mandelbrot set produce different workload distributions. We assumed each iteration took constant time so that the cost of evaluating each point was the number of iterations it took for the value to diverge up to a maximum number we set.

Our simulator makes various simplifying assumptions. We impose no limit on the number of elements that can be distributed to or gathered from the functional units at once. We also assume that the start up and shut down time of the system is negligible relative to the time spent doing real work.


t_{max}
t_{min}
Functional units (N=4)
N-1 units periodically stalled:
$S_{period} = [N-1] \cdot [t_{max} - t_{min}(M+1)]$
Done and buffered (M=2)
Done and blocked on func. unit

$$W_{period} = t_{max} + t_{min} \cdot (N-1)(M+1)$$
$$\#_{periods} = \left\lfloor \frac{W}{W_{period}} \right\rfloor$$
$$W_{remainder} = W - \#_{periods} \cdot W_{period}$$
$$S_{remainder} = \begin{cases} N \cdot t_{max} - W_{remainder} & \text{if } W_{remainder} \geq t_{max}, \\ N \cdot W_{remainder} - W_{remainder} & \text{if } W_{remainder} < t_{max}. \end{cases}$$
$$S = \#_{periods} \cdot S_{period} + S_{remainder}$$

Figure 2. The pattern that repeats itself periodically in the worst case input and the equations formalizing the pattern.

3. Worst-Case Workloads

The worst-case workload maximizes the total time spent stalling, S, for a given amount of real work, W. While it is impossible to have all N functional units stalling at once,[1] it is possible for $N-1$ units to stall simultaneously. This occurs when there is one long job followed by enough short jobs to fill the buffers of the other functional units, thereby blocking those functional units until the long job completes. Once the long element is finished, the system can drain the filled buffers, allowing the other functional units to resume useful work.

Figure 2 depicts this worst-case pattern and includes a closed-form expression for the total stall time, S, that a configuration with N functional units and M buffer slots per functional unit incurs on a worst-case workload with job processing times ranging from t_{min} to t_{max}.

Figure 3 illustrates experimentally how the completion time for a particular system configuration (a number of functional units and the size of their buffers) varies from the worst-case workload to the best (uniform processing times, implying stall-free behavior). To generate these plots, we ran a variety of workloads with the same total amount of work on various system configurations and plotted the completion time of each workload. The abscissa of each plot denotes the number of functional units; each plot represents a different number of buffers per functional unit.

[1] Were all N functional unit to stall, it would mean every functional unit has finished processing its input element, each local buffer is full, and the gather side of the system is waiting to collect some element x that has yet to be processed. But x had to have been processed already, since the system only gathers elements that have already been distributed and every functional unit is finished processing its element. Then the gather side of the system would have already collected x, contradicting our assumption that it was waiting to collect x.

The envelope drawn around each plot represents the best- and worst-case workloads: the solid line on the right of each is the worst-case workload described in Figure 2; the dotted line on the left is the best-case workload (uniformly unit-time work units). All the other points represent various other workloads, mostly random.

Figure 3 demonstrates that adding buffers improves the benefits of multiple functional units, but that these benefits quickly diminish after about four buffer slots per unit. As the number of buffers per unit increases, the vertical extent of the envelope diminishes, meaning the variance among workloads diminishes.

4. Finding Optimal Configurations

We now address the central question in our work: how best to divide area between functional units and buffers. A key parameter is β: the relative cost of buffers and functional units. We consider cases when functional units are larger than buffers—when our parameter β is less than 1—because we expect computing data to be more expensive than merely storing it. Something like the Mandelbrot set generator is illustrative: a functional unit includes at least one multiplier (it could easily use four floating-point multipliers), whereas each unit of buffering might only store eight bits.

Figure 4 shows our experimental results, which provides a design guide. Each series of points represents how completion time on a worst-case workload varies as functional units are replaced with buffers. Points on the left represent systems with many functional units and minimal buffering; moving right involves increasing buffer sizes at the expense of the number of functional units.

Figure 4 provides a technique for finding the optimal configuration under the worst case workload. For a specific β (size of the functional unit relative to a buffer, something determined by the function being mapped and how it is implemented), the optimal trade-off between functional units and buffers occurs at the minimum point on the curve.

Moving to the left on these curves illustrates how eliminating buffering can slow worst-case workloads. The extreme leftmost point corresponds to having no buffers, and having at least single-place buffers always provides some improvement on a worst-case workload. However, when buffers are costly, only a few suffice: beyond a certain point, adding buffers actually decreases performance because the area could be better spent on functional units performing more operations in parallel, even though less buffering means functional units stall more. The rightmost point on each series corresponds to a nonsensical system: a single functional unit with an enormous buffer. The buffer is useless because a single functional unit never stalls no matter the size of the buffers.

Interestingly, the optimal configuration is often not unique. This is most pronounced when the cost of buffers is high, and we begin to see discretization effects. Since in this ex-

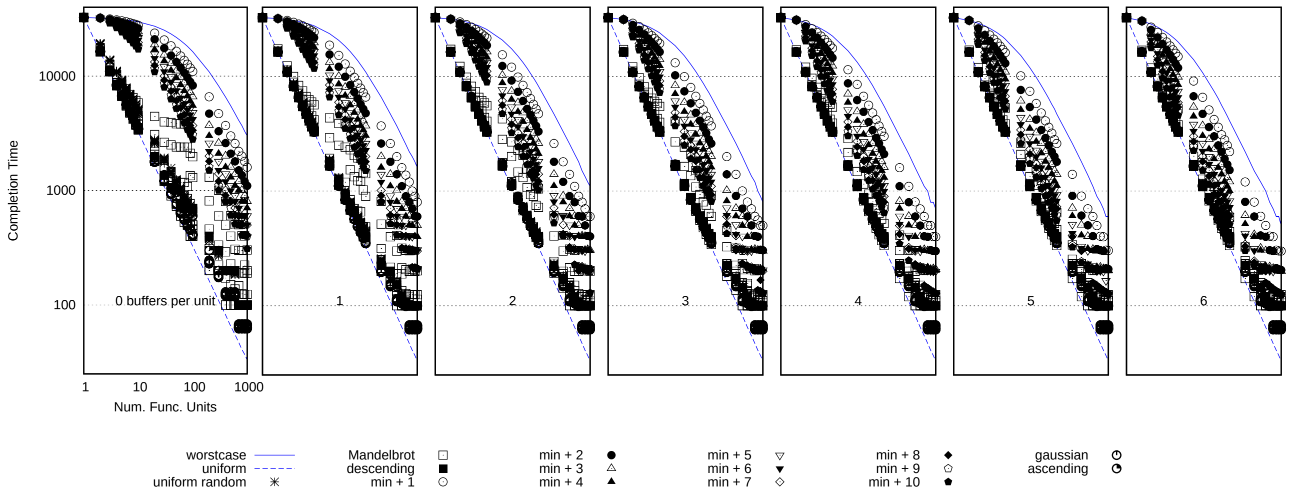


Figure 3. Comparing completion times of workloads across system configurations. These graphs show that the worst-case and best-case workloads provide upper and lower bounds on completion time, respectively. We generated each plot by holding the number of buffer slots per functional unit constant and varying the number of functional units on the x-axis. Each point is the result of a non-worst-case workload being computed by a given system configuration.

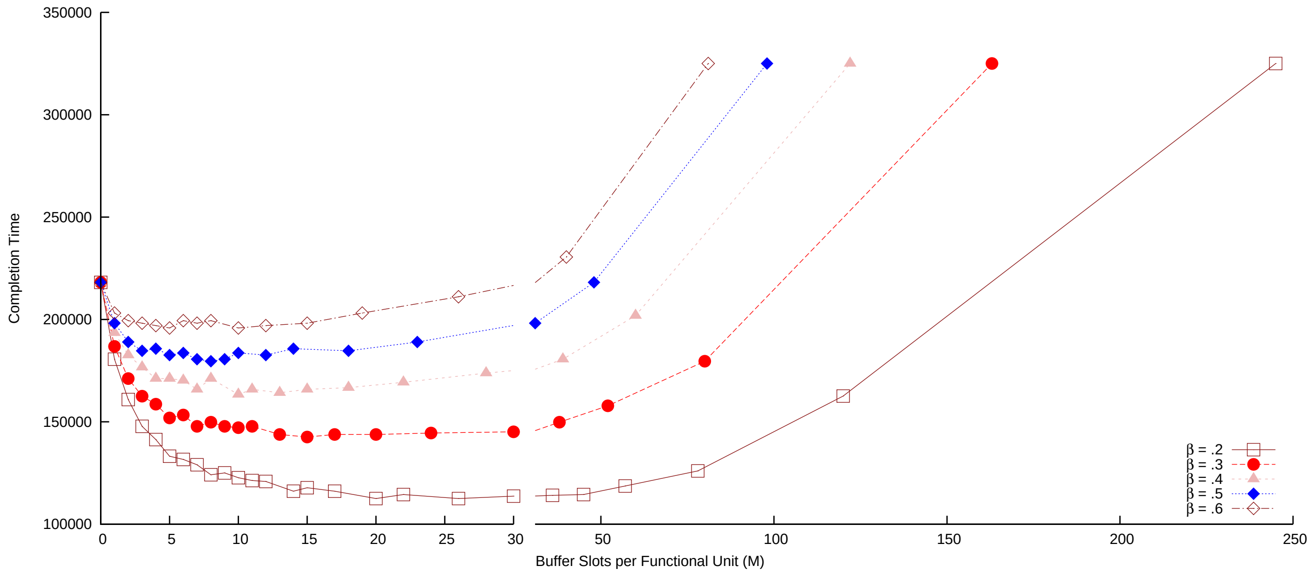


Figure 4. Completion times on a fixed-length worst-case workload as a function of buffer slots for various buffer costs. Each implementation consumes 50 area units ($A = 50$). The optimal implementation for a given β (area cost of a buffer slot relative to a functional unit) is the lowest point on that series—the shortest completion time. Note that most series have multiple minima. The details (but not total work) of each worst-case workload is tailored to each system configuration.

periment we set the granularity of functional units fairly low (we can only have exactly $1, 2, 3, \ldots,$ or 50 functional units), there are cases where we assume area is left unused.

We see a similar common completion time for the configurations that maximize functional units with zero buffers. The omission of buffers results in a high level of parallelism, which results in lower completion time than when we chose to maximize buffer slots. However, the fact that we have so many functional units and zero buffers drastically increases the probability of functional unit stalls, leading to sub-optimal completion time.

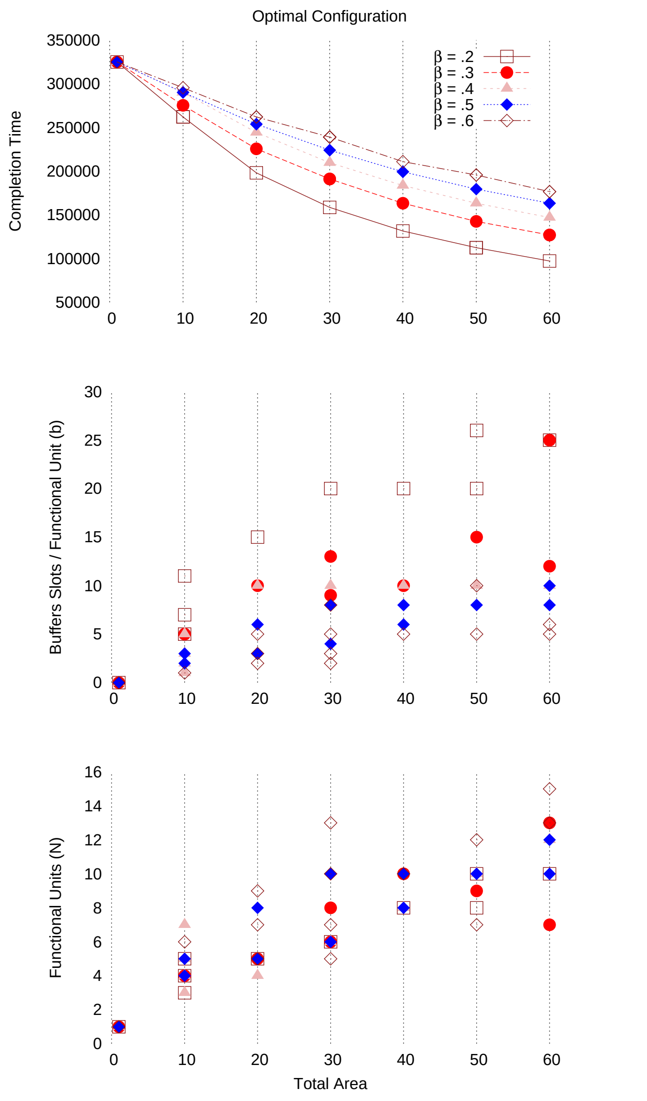


Figure 5. Sensitivity of the Optimal Configuration to Changes in Chip Area. In many cases there are multiple optimal configurations for a given area budget and relative buffer cost

As β decreases, the difference between the best configuration and the worst configuration increases. For example, when $\beta = 0.3$ the best configuration runs approximately 2× faster than the worst configuration, while the speedup for $\beta = 0.2$ is closer to 3×. Furthermore, the completion time for the optimal configuration decreases as β decreases. This makes sense, since when buffer slots are smaller relative to functional units, more buffer slots can fit onto a chip with a given number of functional units N, decreasing the total stall time.

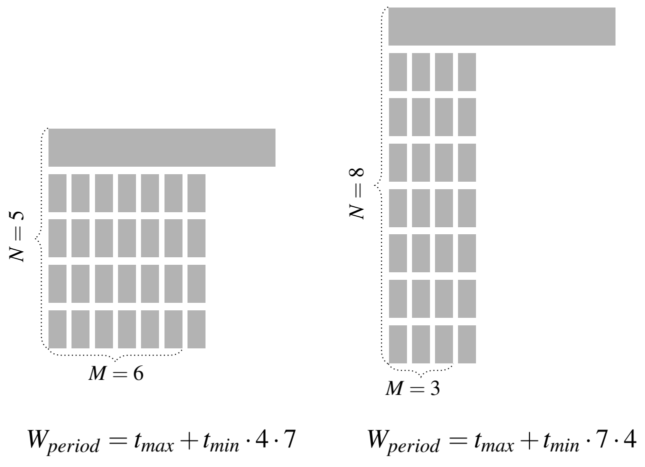

$N = 5$
$M = 6$
$N = 8$
$M = 3$
$W_{period} = t_{max} + t_{min} \cdot 4 \cdot 7$
$W_{period} = t_{max} + t_{min} \cdot 7 \cdot 4$

Figure 6. The behavior of the worst-case workloads for two optimal configurations shown in Figure 1. Although different, they consume the same area, and perform the same total amount of work per period at the same periodic rate, resulting in equal throughput.

5. Sensitivity of the Optimal Configuration

Figure 5 shows how changing the area budget affects the completion time and design of the optimal configuration. We first see that increasing the area budget decreases the completion time of the optimal configuration, which is not surprising since we either use area to increase parallelism or decrease overall stalls.

The bottom two plots show how resources were allocated to obtain these optimal configurations. The middle plot shows the number of buffer slots per functional unit (M) on the y-axis, while the bottom plot measures the number of functional units (N). In some cases there are multiple optimal points for a given area budget and size ratio. This indicates that there were multiple resource allocations for that area budget and relative buffer cost that led to the same, minimal, completion time. We could then use another metric such as power consumption to break ties between these optimal allocations.

The two block diagrams from Figure 1 represent the two optimal system configurations seen in Figure 5 as dark blue diamonds at $A = 20$. Figure 6 depicts the periodic worst-case behavior of these two configurations. On the right side of Figure 6, each filled-in rectangle represents an input element and each empty rectangle represents a functional unit stall. The worst case consists of one maximally long element followed by enough small elements to fill up the buffers. The long element requires ten times the processing time of a small element. By inspecting the number of elements in each configuration's block, we see that they do the same amount of processing in each block (1 long job and 28 small jobs) and both blocks take the same amount of time to complete since the processing time of a long job is the same in each configuration. This equivalence in processing speed will lead the configurations to finish processing a worst-case workload of the same size at the same time, which explains how they could both be an optimal system configuration for a given area budget and relative buffer cost.

6. Conclusions

We considered an abstract implementation of a parallel map operation in hardware and how best to devote limited area resources to maximize throughput. Starting from a simple model with unit-size functional units, buffers of size β, and an area budget of A units, we devised a worst-case distribution of element execution times that lead to the lowest throughput, then showed how to best distribute resources between functional units and buffers in such a setting. Although we focused solely on area restrictions, this model could be manipulated to include other metrics like power and timing, and we hope to explore such modifications in future work.

Through simulation, we showed how the optimal trade-off between functional units and buffers changes as the relative cost of buffers drops and how sensitive the optimal trade-off is for various buffer costs. One unexpected result was that there were often multiple optimum configurations that were fairly different.

This work was done in the context of a large project designed to optimize the parallel implementation of algorithms on custom hardware. Once we integrate our findings into this project we will get a better sense of how far real workloads can deviate from our worst-case model. Next, we will use our work, especially the shape of the curves in Figure 3, to derive a heuristic design aide that can automatically make a reasonable trade-off between buffer sizes and the number of functional units in a synthesis setting. Our ultimate goal is to make such trade-offs the job of a compiler, not the designer.

References

[1] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of Operating System Design and Implemetation (OSDI)*, Dec. 2004.

[2] B. B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman and Company, 1982.

[3] J. McCarthy, P. W. abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin. *LISP 1.5 Programmer's Manual*. MIT Press, 1962.

[4] M. Purnaprajna, M. Reformat, and W. Pedrycz. Genetic algorithms for hardware-software partitioning and optimal resource allocation. *Journal of Systems Architecture*, 53(7):339–354, July 2007. .

[5] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang. FPMR: mapreduce framework on FPGA; a case study of rankboost acceleration. In *Proceedings of the 18th annual ACM/SIGDA international symposium on FPGAs*, pages 93–102, New York, NY, 2010.

[6] J. H. C. Yeung, C. C. Tsang, K. H. Tsoi, B. S. H. Kwan, C. C. C. Cheung, A. P. C. Chan, and P. H. Leong. Map-reduce as a programming model for custom computing machines. In *Proceedings of the Symposium on FPGAs for Custom Computing Machines (FCCM)*, pages 149–159, Washington, D.C., 2008.