Libra: Tailoring SIMD Execution using Heterogeneous Hardware and Dynamic Configurability*

Yongjun Park Jason Jong Kyu Park Hyunchul Park† Scott Mahlke

Advanced Computer Architecture Laboratory
University of Michigan
Ann Arbor, MI, USA
{yjunpark, jasonjk, parkhc, mahlke}@umich.edu

Abstract

Mobile computing as exemplified by the smart phone has become an integral part of our daily lives. The next generation of these devices will be driven by providing an even richer user experience and compelling capabilities: higher definition multimedia, 3D graphics, augmented reality, games, and voice interfaces. To address these goals, the core computing capabilities of the smart phone must be scaled. However, the energy budgets are increasing at a much lower rate, requiring fundamental improvements in computing efficiency. SIMD accelerators offer the combination of high performance and low energy consumption through low control and interconnect overhead. However, SIMD accelerators are not a panacea. Many applications lack sufficient vector parallelism to effectively utilize a large number of SIMD lanes. Further, the use of symmetric hardware lanes leads to low utilization and high static power dissipation as SIMD width is scaled. To address these inefficiencies, this paper focuses on breaking two traditional rules of SIMD processing: homogeneity and static configuration. The Libra *accelerator increases SIMD utility by blurring the divide between vector and instruction parallelism to support efficient execution of a wider range of loops, and it increases hardware utilization through the use of heterogeneous hardware across the SIMD lanes. Experimental results show that the 32-lane Libra outperforms traditional SIMD accelerators by an average of 1.58x performance improvement due to higher loop coverage with 29% less energy consumption through heterogeneous hardware.*

1. Introduction

The mobile devices market, including cell phones, netbooks, and personal digital assistants, is one of the most highly competitive businesses. The computing platforms that go into these devices must provide ever increasing performance capabilities while maintaining low energy consumption in order to support advanced multimedia and signal processing applications. Application-specific integrated circuits (ASICs) are the most common solutions for meeting these requirements, performing the most compute-intensive kernels in a high performance but energy-efficient manner. However, several features push designers to a more flexible and programmable solution: supporting multiple applications or variations of applications, providing faster time-to-market, and enabling algorithmic changes after the hardware is constructed.

Processors that exploit instruction-level parallelism (ILP) provide the highest degree of computing flexibility. Modern smart phones employ a one GHz dual-issue superscalar ARM as an application processor. Higher performance digital signal processors are also available such as the 8-issue TI C6x. However, ILP processors have scalability limits including many-ported register files (RFs) and complex interconnects. Alternately, single-instruction multiple-data (SIMD) accelerators provide high efficiency because of their regular structure, ability to scale lanes, and low control logic overhead. They have long been used in the desktop space for high performance multimedia and graphics functionality. But, their combination of scalable performance, energy efficiency, and programmability make them ideal for mobile systems [24, 9, 15, 27].

In order to fully utilize the SIMD hardware, it is necessary for the programmer or compiler to extract sufficient data-level parallelism (DLP). Automatic loop vectorization is available in a variety of commercial compilers including offerings from Intel, IBM, and PGI. Classic scientific computing (regular structure, large trip count loops, and few data dependences) are naturally well-matched to SIMD accelerators. But, in many respects, the mobile terminal has become a general-purpose computer. Thus, like the desktop, only a small percentage of mobile applications look like classic scientific computing. The computation is not dominated by simple vectorizable loops, but by loops containing significant numbers of control and data dependences to handle the complexity of modern multimedia standards. As a result, applications have varying amounts of vector parallelism ranging from none to some to large amounts. The net effect is that SIMD hardware goes unused for a large fraction of application execution and thus cannot be counted on to provide significant performance gains.

A second but inter-related problem with SIMD computing is low hardware utilization even when vector loops are executed. The use of homogeneous hardware (e.g, identical lanes) is one of the best advantages of SIMD datapaths by reducing design cost and complexity. But, the utilization of the most complex components of a SIMD lane is often disproportionally lower than the simple components. For example, the H.264 video decoding application is dominated by simple integer operations (adds, subtracts, shifts) and an average of only 2.2% and 1.3% of the total dynamic instructions are multiplies and divides [8]. This is not an outlying data point, most multimedia and visual computing applications have small fractions of multiply, divide and other expensive operators. For 128-bit SIMD (4 lanes), such utilization rates may not matter, but as SIMD widths are scaled to increase performance to 1024 bits (32 lanes) or more, the problem becomes serious due to poor area utilization and high static power dissipation.

To attack these problems, we propose a customizable SIMD accelerator that is capable of tailoring its execution strategy to the running application, referred to as the *Libra*. Libra employs two key concepts, *heterogeneity* and *dynamic configurability*, to achieve broader applicability and better energy efficiency than traditional SIMD accelerators. Heterogeneity allows lanes to have different functionali-

*To appear in the 45th International Symposium on Microarchitecture (2012).
†Currently with Programming Systems Lab, Intel Labs, Santa Clara, CA

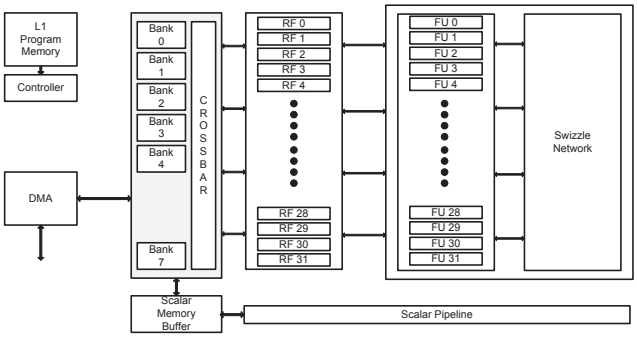


Figure 1: A traditional 32-lane SIMD accelerator.

ties and better match functional capabilities with expected operator distributions. Dynamic configurability enables lane resource to execute as a traditional SIMD processor, be re-purposed to behave as a clustered VLIW processor, or combinations in between. Dynamic configurability also enables efficient sharing of expensive resources between lanes (e.g., multipliers) by interleaving independent instructions with each lane's expensive instruction so as to hide resource contention. Libra consists of an array of simple processing elements (PEs) that are tightly interconnected by a scalar operand network. Groups of four PEs form PE groups that are normally driven by a single instruction stream. Each group can behave as a building block for a SIMD processor (e.g., PEs behave as SIMD lanes) or a VLIW processor (e.g., PEs behave as a cluster of function units). The compiler maps 1 or more loops to the Libra accelerator by combining and configuring clusters of PE groups to efficiently exploit the available DLP and ILP.

This paper offers the following three contributions:

- An in-depth analysis of the available ILP/DLP parallelism and its variability in three representative mobile application domains: computer vision applications, commercial media applications optimized in industry level, and game physics engine applications.
- The design of a unified loop accelerator that can effectively support future mobile applications with varying performance requirements and characteristics. To achieve this objective, we offer three key features:
  1. Scalability: Libra can meet high performance requirements by simply increasing the number of clusters, whereas most current accelerators suffer from poor scalability.
  2. Configurable performance: Libra can dynamically tune the ILP/DLP-support capability in order to successfully support ILP-intensive, DLP-intensive, and ILP/DLP-mixed applications, as well as tolerate performance degradation due to its heterogeneity.
  3. Energy efficiency: Simple hardware implementation achieves high energy-efficiency with competitive performance.
- A light-weight design and organization of a configurable processing element for supporting simple latency hiding techniques and sharing expensive resources.

2. Background and Motivation

In this section, we examine the limitations of traditional SIMD accelerators based on an analysis of various mobile applications. We first introduce the target benchmarks and the baseline architecture, and find two main sources of inefficiencies in SIMD accelerators. We then propose high-level solutions to overcome these challenges that facilitate designing efficient hardware and maximizing the utilization of existing resources.

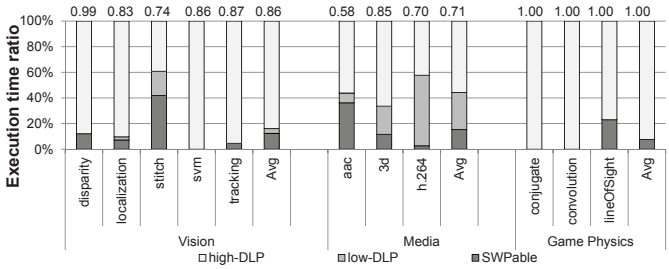


Figure 2: Loop categorization: The components of the bar indicate ratio of execution time in SWPable loops, low-DLP, and high-DLP SIMDizable loops. The ratio of loop execution time over total execution time is indicated as a number above each bar.

2.1. Benchmarks Overview

Three classes of mobile benchmarks are used for this application analysis that contain varying degrees of vector parallelism. The benchmarks consist of:

- Vision benchmark: We evaluated a subset of the SD-VBS benchmark suite [26] for mobile vision applications. As these benchmarks are not originally optimized for a specific target architecture, we manually modified these benchmarks to increase the opportunities for efficient execution with function inlining and loop unrolling. All the benchmarks are functionally verified on QCIF[1] input data sizes, which is widely used on mobile devices.
- Media benchmark: Three mobile media applications are selected: AAC decoder (MPEG4 audio decoding, low complexity profile), H.264 decoder (MPEG4 video decoding, baseline profile, qcif) [13], and 3D (3D graphics rendering) [3]. These benchmarks are optimized for DSPs in the production-quality level and a large portion of the loops have a high potential degree of ILP and are software pipelinable.
- Game physics benchmark: Three common kernels are extracted from mobile game applications [2]. First, lineOfSight plays an important role of separating visible objects and non-visible objects. Sound effects, collision detection and other functions involving linear equations often exploit convolution and the conjugate gradient method. The three kernels mostly consist of high DLP loops.

2.2. Baseline Architecture

A SIMD architecture that is based on SODA [15] is used as the baseline SIMD accelerator. This architecture has both SIMD and scalar datapaths. The SIMD pipeline consists of a multiple-lane datapath where each lane has an arithmetic unit working in parallel. Each datapath has two read-ports, one write-port, a 16 entry register file, and one ALU with a multiplier. The number of lanes in the SIMD pipeline can vary depending on the characteristics of the target applications. The SIMD Shuffle Network (SSN) is implemented to support intra-processor data movement. The scalar pipeline consists of one 32-bit datapath and supports the application's control code. The scalar pipeline also handles DMA (Direct Memory Access) transfers.

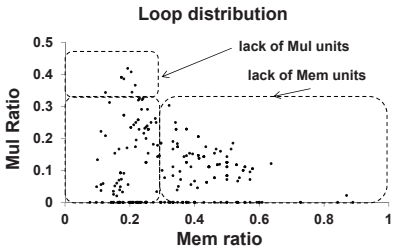
2.3. Limitations for Current SIMD Accelerators

2.3.1. Loop Characterization Applications typically have many compute intensive kernels that are in the form of nested loops.

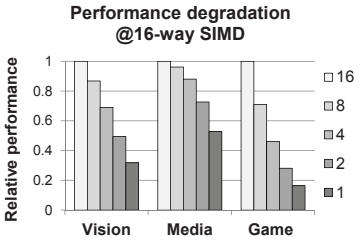
[1] We used QCIF (176x144) image size for uniformity of benchmarks, and the similar trend appears on higher resolution images.

	Vision	Media	Game	Total
Avg ratio(MEM)	0.44	0.26	0.27	0.32
Avg ratio(MUL)	0.15	0.10	0.22	0.16
ratio of MEM loop	0.93	0.36	0.33	0.54
ratio of Mul loop	0	0.04	0	0.01

(a)



(b)



(c)

Figure 3: Resource utilization: (a) average ratio of dynamic instruction count of expensive instructions and ratio of Mem/Mul dominant loops, (b) loop distribution over ratio of Mem/Mul, and (c) performance degradation on a SIMD at different number of Mem/Mul resources.

Among these kernels, we analyze the available ILP and DLP of the innermost loops and find the maximum natural vector width that is achievable. To extract the maximum degree of ILP, we found the *Software pipelinable* innermost loops: 1) counted loop, 2) no subroutine call, and 3) no multiple exits/backedges. Control flows inside the innermost loops are solved using if-conversion. Among the software pipelinable (SWPable) innermost loops, we also identify the *SIMDizable* innermost loops which can utilize DLP. We apply the conditions used by the Intel compiler [12] to determine if a loop is SIMDizable and the minimum iteration count is set to the maximum available SIMD width (natural SIMD width).

2.3.2. SIMD Width Variance over Loops Figure 2 shows the relative execution time of SWPable loops and SIMDizable loops to total execution time on a simple 1-issue ARM processor. As we use a 16-lane SIMD processor for this experiment, SIMDizable loops with natural SIMD width smaller than 16 are categorized into low-DLP loops. On average, there is a substantial amount of time (87%) spent on SWPable or SIMDizable loops as expected. An interesting question here is how many applications are not well-matched to a wide SIMD accelerator. Unfortunately, 4 of 11 applications are highly dependent on SWPable and low-DLP loops, which means that not all the lanes can be utilized. For example, traditional SIMD cannot decrease the execution time of an AAC application more than 60% of the total loop execution time because around 40% of the time is spent on SWPable loops. In general, the game physics benchmarks have high levels of data parallelism, vision benchmarks have modest data parallelism, and media benchmarks have low degrees of data parallelism. Results in Figure 2 confirm that a simple SIMD accelerator cannot effectively support the range of mobile applications. Even with a perfect support for DLP, SWPable and low-DLP loop execution result in low utilization of SIMD resources. Therefore, further consideration is required to fully utilize the SIMD resources on the execution of non-fully SIMDizable loops.

2.3.3. Resource Utilization Variance To maximize the total utilization of computation resources, the number of each resource should be decided based on the average fraction of dynamic instructions. While current CPUs solve these challenges by out-of-order execution of parallel instructions on multiple execution units, current SIMD architectures cannot solve this problem due to its homogeneous nature: the datapath of each SIMD lane has the same functionalities, even for expensive units such as memory and multiply units. These characteristics are unfavorable in terms of efficiency because not all execution units are active every cycle, and expensive units are much less utilized (an average of only 32% for a memory unit and 16% for a multiply unit (Figure 3(a))). A traditional solution for this problem is to turn off the unused resources by clock/input gating, but this solution does not eliminate leakage power. Power gating is unlikely a practical solution because idle periods for expensive units tend to be relatively short.

Another challenge is the diversity of instruction distribution across/inside applications. Even if we are somehow able to place a specific number of each execution unit based on average fraction, careful consideration is also required because the fraction varies greatly. In Figure 3(a), for example, the ratio of multiply instruction varies from 10% to 22% across three application domains. We also define a loop to be memory/multiplication dependent if the fraction of memory/multiplication instructions are more than 33% of the total instructions. Figure 3(b) shows a distribution of the loops according to the ratio of memory/multiply instructions. Based on Figure 3(a) and (b), more than 54% of the loops in the three benchmark sets highly depend on the memory instructions, and therefore, normal ALU functional units can be idle due to the memory operation bottleneck if only 33% of memory resources exist. On the contrary, multiplication is not the critical performance bottleneck if 33% of multiplication resources exist because only 1% of the loops are multiplication dependent. As a result, the high diversity in the instruction distribution will make most loops to not be effectively accelerated due to the lack of enough resources, or to waste resources due to the excess resources, if the SIMD accelerator simply allocate resources based on specific rules such as average fraction or one per four lanes.

2.4. Insights for the Traditional SIMD

Based on the application analysis, we found several fundamental sources of SIMD inefficiency. First, a traditional wide SIMD accelerator may be over-designed since the overall performance will be saturated at some point and limited by non-high-DLP loops where the SIMD accelerator is poorly utilized. Second, lane uniformity makes the SIMD datapath inefficient due to over-provisioning expensive resources. Third, the high variation in the resource requirements of loops makes the problem more difficult than simple sharing of expensive resources would accomplish. A central challenge here is how to decrease over-provided resources on traditional SIMD accelerators and to overcome the inflexibility in order to more effectively utilize the hardware.

3. Libra Architecture

3.1. Overview

The Libra accelerator presented here is a unified accelerator for mobile applications that allows flexible execution of loops by customizing the configuration adaptive to their key characteristics. The Libra

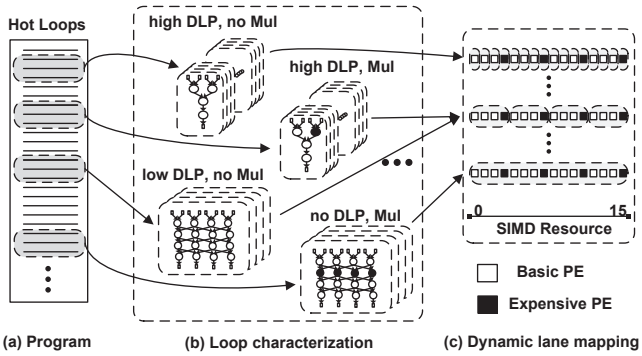


Figure 4: Mapping loops to Libra: (a) identify hot loops, (b) find the available DLP and resource requirement of each expensive operation, and (c) change the configuration based on the characteristics of each loop.

accelerator is based on traditional SIMD accelerators and has several important extensions for providing both high energy-efficiency and performance improvement. First, Libra is composed of a non-uniform lane structure for power efficiency: only a subset of lanes has expensive but infrequently used execution units. Furthermore, dynamic configurability of logical lanes helps Libra in executing a target loop in an efficient manner with high utilization. In Libra, a group of logical lanes is executed in a SIMD manner, where the logical lane is configured by a group of processing elements (PEs). DLP is exploited in the form of parallel execution of logical lanes, and ILP is exploited inside each logical lane in a way that each PE execute different operations. Therefore, Libra is able to flexibly tune the ILP/DLP-support capability by changing the logical lane configuration.

Figure 4 shows a conceptual view of the execution of Libra. First, several hot loops are identified as candidates to be accelerated utilizing the Libra architecture(Figure 4(a)). Second, software-pipelinable loops are selected, and the DLP availability is also determined as discussed in Section 2.3.1(Figure 4(b)). In this step, several additional key characteristics such as the amount of potential ILP in the loopbody and the ratio of expensive instructions are also considered. Finally, a best matched logical lane configuration for each loop is chosen by the compiler (Figure 4(c)). In Figure 4, we assume a 16-lane heterogeneous SIMD including 12 basic and 4 expensive PEs. Based on this, each PE constitutes one logical lane for full DLP support to execute high-DLP loops having only simple instructions, intermediate numbers of PEs form each logical lane for ILP/DLP hybrid execution to support low-DLP loops or expensive operation-intensive loops, and one large logical lane for full ILP execution is configured for non-DLP loops. Note that fully exploiting SIMD parallelism does not always outperform exploiting ILP on heterogeneous structures. Section 3.1.1 and 3.1.2 explain the core concept of Libra in detail with evidence of its effectiveness.

3.1.1. Heterogeneity Heterogeneous lane organization, based on average fraction of resource utilization, is required in order to enhance power efficiency: all the lanes support simple integer operations and only a subset of the lanes support expensive operations. When an expensive instruction is fetched, the accelerator stalls until this subset of lanes generates results for all lanes, then resumes execution. This structure delivers a high level of power efficiency due to the expensive resource removal, but significant performance degradation will occur when executing expensive operation-intensive code. Figure 3(c) illustrates the performance degradation as the number

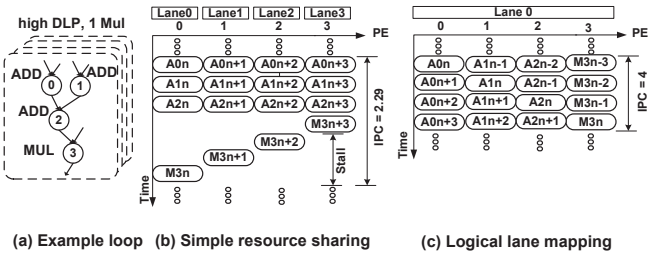


Figure 5: Dynamic configurability on a 4-lane heterogeneous SIMD (lane 3 has a multiplier): (a) a simple high-DLP loop with 1 multiply, (b) performance degradation due to stalls during multiply execution, (c) logical lane formation removes stalls by instruction pipelining.

of multiplier/memory units decreases on a 16-lane SIMD accelerator. Each bar shows the relative performance normalized to that of the homogeneous SIMD when each heterogeneous SIMD has specific number of expensive resources. From this graph, substantial amounts of performance degradation exist in vision and game benchmark because they are highly dependent on expensive operations and incur a number of stalls to handle these operations. However, media benchmarks are not highly affected by the proportion of these expensive resources because the performance is already constrained by low DLP.

3.1.2. Dynamic Configurability Dynamic configurability of lanes helps the heterogeneous SIMD accelerator in dealing with the aforementioned problems. One logical lane can consist of one PE for highly SIMDizable loops with no expensive instructions, and also consist of multiple PEs for non/low-SIMDizable loops or loops having expensive instructions. The resulting SIMD width is decided by the number of logical lanes and each logical lane executes the same instruction stream in lockstep. Inside a logical lane, ILP is exploited to use multiple lanes in parallel, and therefore it can efficiently distribute instructions between simple lanes and expensive lanes.

The effectiveness of dynamic lane mapping can be explained by the simple following performance equation. In the equation, we compare the total performance of the simple SIMD and the Libra SIMD by the metric of IPC (instruction per cycle). The IPC of SIMD can be calculated by the multiplication of IPC of one lane (IPC_{lane}) and the minimum of the number of PEs (N_{SIMD}) and the available degree of DLP (N_{DLP}) of the target loop (Equation (1)). Similarly, the IPC of Libra can be the multiplication of IPC of one logical lane ($IPC_{logical\_lane}$), consisting of m PEs, and the minimum of the number of logical lanes ($\frac{N_{SIMD}}{m}$) and the degree of DLP of the loop (Equation (2)). Therefore, when executing non/low-DLP loops, Libra can easily outperform the basic SIMD because it only requires better performance of a logical lane than that of a PE, and it is always true as a logical lane exploits ILP with multiple PEs inside(Equation (3)). Dynamic configurability is also able to address the performance degradation problem on the heterogeneous SIMD. When executing high-DLP loops, Libra outperforms SIMD when the IPC of a logical lane is higher than that of m PEs. Although the ILP performance is normally inferior to DLP performance because of its dependences and complexity, Libra can frequently be better due to the heterogeneity. Figure 5(a), (b) and (c) shows the superiority of Libra. Figure 5(b) and (c) show the execution of a simple high-DLP loop having a multiply instruction on both the simple SIMD and Libra which have one multiplier on the PE 3. In this example, the IPC of SIMD is less than the IPC of Libra when one large logical lane is configured due to a number of stalls.

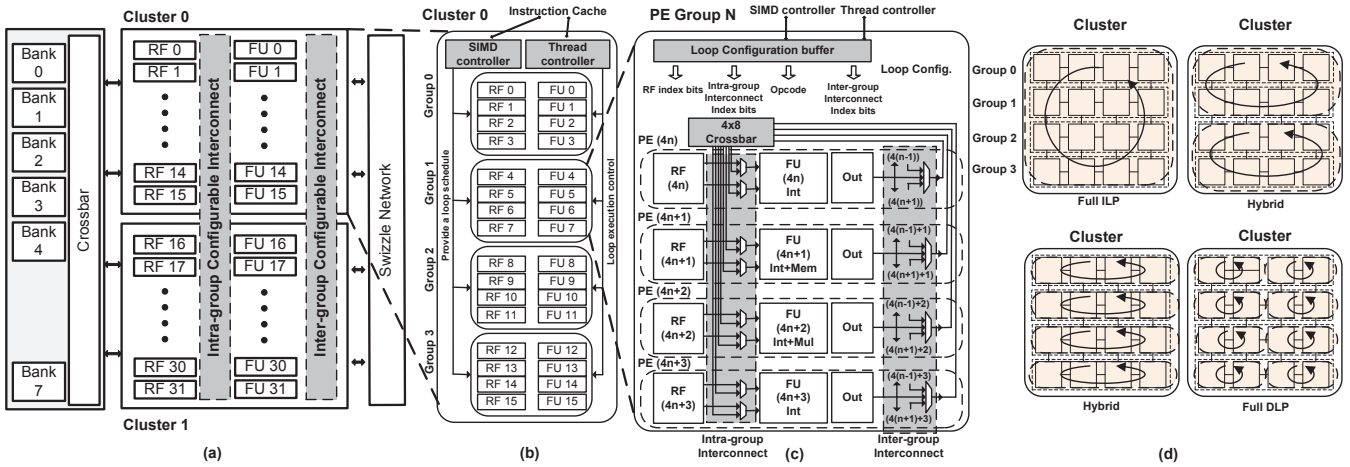


Figure 6: The 32-PE Libra architecture: (a) a 2-cluster Libra accelerator, (b) a cluster, (c) an example of a single PE group: PE 1 supports memory operation and PE 2 supports multiply operation, and (d) execution modes.

$$IPC_{SIMD} = \min(N_{SIMD}, N_{DLP}) \times IPC_{lane} \quad (1)$$

$$IPC_{Libra} = \min(\frac{N_{SIMD}}{m}, N_{DLP}) \times IPC_{logical\_lane} \quad (2)$$

$$IPC_{Libra} > IPC_{SIMD},$$

$$when \begin{cases} IPC_{logical\_lane} > IPC_{lane}, & \text{if } \frac{N_{SIMD}}{m} > N_{DLP} \\ IPC_{logical\_lane} > m \times IPC_{lane}, & \text{if } \frac{N_{SIMD}}{m} < N_{DLP} \end{cases} \quad (3)$$

3.2. Microarchitectural Details

The Libra architecture with eight PE groups (32 PEs) is shown in Figure 6(a). Differently from the traditional SIMD, the Libra datapath consists of 2 groups of clusters, which can be configured to create logical SIMD lanes of 2, 4, 8, and 16 PEs based on the loop characteristics. Each of the clusters is composed of 4 PE groups. The SIMD controller performs the role of managing the logical lane status to exploit SIMD parallelism, while the thread controller manages the ILP-exploiting method inside the logical lane. Each PE group contains 4 PEs. Each of the PEs has an FU and a register file, which can be thought as one lane of the traditional SIMD. Only one of the PEs in a PE group has a multiplier while another has a memory unit. Differently from the traditional SIMD, each PE group also has two kinds of reconfigurable interconnects inside and across PE groups in order to achieve flexible configuration of logical lanes.

Key features of Libra architectures are as follows:

Scalability: The resources are fully distributed including FUs, register files, and interconnections. PE groups have dense interconnections inside but each PE group is sparsely connected with neighbors. As a result, area and power costs increase approximately proportional to the number of resources, which makes Libra as scalable as a simple SIMD.

Polymorphic Lane Organization: PE groups can be aggregated to form a larger logical lane in order to exploit the existing ILP inside the loop body, or be split into multiple small logical lanes in order to exploit DLP over loop iterations.

Resource Sharing: In heterogeneity, the major challenge is how to determine the number of expensive resources and how to efficiently share them between logical lanes when necessary. To flexibly handle this, we place the expensive resources based on the average utilization and provide a sharing mechanism between them in two categories. A more detailed description is provided in Section 3.3.3.

Simple Multi-threading Mechanism: Even though a logical lane provides a number of parallel resources, efficient use of the available resources is limited due to the low ILP of the loopbody. Therefore, we extended the ILP into loop-level parallelism through modulo scheduling [20]. Modulo scheduling generally provides a decent performance improvement by parallelizing instructions over loop iterations and hiding long latency between back-to-back instructions. However, several Libra specific features, such as SIMD capability and fully-distributed nature, diminish the effectiveness of modulo scheduling. To compensate for this, simple static multi-threading with list scheduling is proposed in Section 3.4.

3.2.1. PE Group A detailed illustration of a single Libra PE group is provided in Figure 6(c). A PE group consists of four PEs each with a 32-bit FU and a 16-entry register file with 2-read/1-write ports (write ports can be added to support threading). Integer arithmetic operations are supported in all four FUs but multiply and memory operations are available in only one FU per PE group (PE1 for memory and PE 2 for multiplication in Figure 6(c)). The FUs inside are modified to connect with each other with a dense 4x8 full crossbar network for passing data between the FUs without writing back to the RF. This allows the PE groups to exploit ILP in a distributed nature. In order to retain scalability, the Libra architecture has a simple and fully distributed across-PE group interconnect. Only FUs are connected between the corresponding neighbors in adjacent PE groups. In addition to these components, a loop configuration buffer is added to store instructions for modulo/list scheduled loops. The buffer is a small SRAM that saves the configuration information including instructions, register addresses and interconnect index bits of the current loop. The interconnect between the loop buffer and SIMD/Thread controllers in the cluster is used to transfer instructions for executing loops. The hardware components and execution mechanism for SIMD/ILP support is explained in detail in Sections 3.3 and 3.4.

3.2.2. Cluster A cluster is a high-level basic unit that consists of four PE groups and several additional features for flexible loop execution support: the SIMD controller and the thread controller. The SIMD controller is a small controller to manage the logical lane organization inside the cluster, including the number of logical lanes and the SIMD width of memory transfer. It receives the information from the instruction cache. In addition, the SIMD controller also gets the configuration for one logical lane from the instruction cache

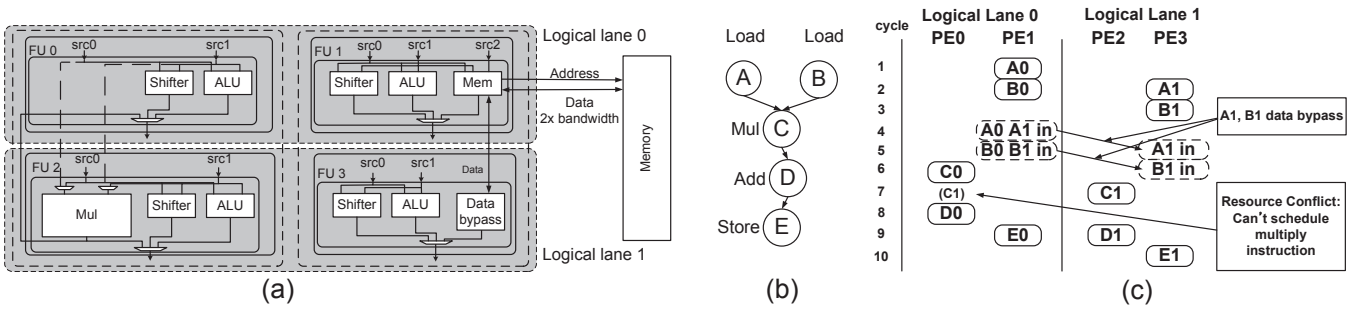


Figure 7: Resource sharing support: (a) hardware modification: PE 0 and 2 share the multiplier and PE 1 and 3 share the memory unit, (b) example loop body dataflow graph, and (c) actual schedule: 1-cycle difference between lanes for resource contention avoidance.

and transfers it to each PE group. A thread controller is responsible for executing loops. It also gets the information about which mode is selected from the instruction cache and orchestrates the loop execution. When modulo scheduling is selected, it just executes the loop sequentially, and, when multi-threading is selected, it executes the loop in the order of the thread sequence table. The information is statically set during compile time and is fetched from the instruction cache. Multiple clusters can execute one large loop or can execute multiple parallel loops separately.

3.2.3. Configuration Process Loop execution of Libra can be divided into two stages: configuration and execution. Configuration stage is forming logical lanes and sending configuration bits to all the loop buffers of each PE-group. For every loop, the instruction cache contains both logical lane organization information and configuration bits for one logical lane. The SIMD controller gets these information from the instruction cache and then sends the configuration bits to the loop buffers of the PE groups based on the logical lane configuration. The thread controller also gets the information about the execution mode and sequence table, if required, from the instruction cache. This process takes 3-5 cycles on average before the loop buffer receives the configuration bits for the first cycle and the time varies depending on the size of the logical lane. The thread controller starts the execution when the first cycle configuration is ready on all the loop buffers.

3.2.4. Memory Support The memory operation of the Libra system needs support for both scalar and SIMD memory access. For scalar memory access, the local memory has the same number of banks as the number of total memory units. For SIMD access, the local memory also needs to support contiguous access across all logical lanes in parallel. Therefore, for the 32-PE Libra system, a 64kB local memory is used, consisting of 8 memory banks where each bank is a 2-wide SIMD containing 1024 32-bit entries. As shown in Section 2.3.1, all memory transfers have the same strides over iterations in SIMDizable loops. Therefore, when several logical lanes execute the same instructions for SIMDized loops, a single address calculation followed by a wide memory operation is performed. The data is then distributed to different logical lanes. Multiple memory units inside a logical lane need to generate their own memory addresses. The SIMD width of each access and the number of different addresses are determined by the logical lane configuration, which is saved in the SIMD controller.

3.2.5. Communication with a Host Processor The Libra architecture is a co-processor similar to a GPU and interfaces with a host processor such as ARM using memory. The data transfer is performed through a standard AMBA bus along with a DMA.

3.3. Execution Model

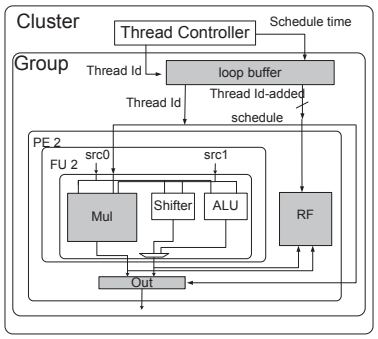
This section describes the three different execution modes of the Libra architecture, which are full ILP, hybrid, and full DLP modes. We first explain how each mode operates and then provide proof of how the three modes can effectively support different kinds of loops. The example provided assumes a four-PE group cluster as shown in Figure 6(d).

3.3.1. Full ILP Mode In this mode, the Libra architecture decides to use all the PEs as one large logical lane. The SIMD controller spreads different configuration informations into the loop buffer of each PE group. The execution mechanism is the same as the loop acceleration technique of common VLIW solutions but the performance might be slightly worse than previous solutions because the Libra architecture sacrifices both centralized resources and dense across-PE group interconnects. Applications which have a high proportion of non-SIMDizable loops mostly utilize this mode for acceleration.

3.3.2. Hybrid Mode When a loop is SIMDizable, a cluster has the possibility of either having several small logical lanes or forming a large logical lane. In this case, the Libra architecture may choose to use a hybrid mode with a cluster having at least two logical lanes, each having at least one PE group. With smaller logical lanes, the performance usually increases since SIMDization provides an opportunity to increase performance by the same amount as the degree of DLP. Also the routing overhead decreases with small logical lanes, further boosting performance. Figure 6(d) also has two examples of hybrid mode execution. The SIMD controller distributes the same configuration information and live values to the loop buffer and RFs of each logical lane. When a loop lacks sufficient level of DLP or has a moderate proportion of expensive resources, hybrid mode can achieve the best performance.

3.3.3. Full DLP Mode When a loop is highly data-parallel but has a low degree of ILP, the resources (PEs) cannot be effectively utilized because the degree of ILP in the loop cannot meet the minimum degree of the PE group. To compensate for the lack of ILP, the Libra architecture supports separation of PE groups, forming two smaller logical lanes. As a result, SIMD parallelism can make up for insufficient ILP in the loops (also in Figure 6(d)). Hence, a cluster has a total of eight logical lanes executing in lockstep. Distinct from loops with a small number of instructions, loops with unbalanced resource usage can also be well matched to a full DLP mode, unlike the hybrid mode. As mentioned in Section 2.3.3, the hybrid mode cannot fully utilize resources in a PE group since performance of loops with a high proportion of memory operations are constrained by the memory unit.

The major challenge in full DLP mode is determining how to



(a)

Cycle	Thread Id	Loop buffer address
0	0	0
1	0	1
2	1	0
3	1	1
4	0	3
5	0	4
6	0	5
7	1	3
8	1	4
9	1	5

(b)

Original Cycle	Original Configuration
0	A
1	B
2	NOP
3	C
4	D
5	E

(c)

Cycle	Thread-aware configuration
0	A0
1	B0
2	A1
3	B1
4	C0
5	D0
6	E0
7	C1
8	D1
9	E1

(d)

Figure 8: Multi-threading support & compiler support: (a) hardware modification: shaded components are modified, (b) sequence table in the thread controller, (c) loop buffer, and (d) final multi-threaded schedule.

share expensive resources between two small logical lanes in a PE group. The first category for resource sharing is expensive but infrequently used functionalities such as the multiply operation. As shown in Figure 3(a), the average ratio of multiply is as low as 16% and only 1% of loops are multiply-dominant, and therefore simple sharing between two half-PE groups does not incur performance degradation. The second category is frequently used functionalities such as memory operations as shown in Figure 3(a). These instructions are already a performance bottleneck and simple sharing cannot enhance the overall performance. Therefore, this shared resource should lead to double the performance in a lightweight manner.

We accomplish these requirements using simple hardware modifications as shown in Figure 7(a). One PE group is mapped into two small logical lanes with (PE 0, PE 1) and (PE 2, PE 3). Based on the application analysis, only PE2 supports multiply operations and PE 1 supports memory operations. To ensure that both logical lanes support all functionalities, PE 0 and PE 2 share the multiplier and PE 1 and PE 3 share the memory unit. To share the multiplier, PE 0 connects input and output ports to the multiplier of PE 2. A memory controller in PE 1 is shared with PE 3 in a different manner. When the memory controller receives a memory operation command, only PE1 communicates with the memory with double bandwidth and send/receives the data of PE 3 through a bypass logic.

To execute the same instructions in both logical lanes using the above modifications, the following processes are required:

- The compiler must not schedule multiply instructions in a row, because the multiplier needs a spare cycle after the cycle in which the multiply instruction is scheduled in order to handle the operation of the other logical lane. However, other instructions can be placed since they have no resource or writeback contention. Memory instructions can be scheduled without any restrictions as the hardware supports double bandwidth.
- The SIMD controller has the instruction configuration only for one logical lane. The controller transfers the same configuration into the loop buffer of both logical lanes with one-cycle difference to avoid resource contention.

Figure 7(b) is an example of a full DLP mode execution. For a simple dataflow graph of the loop body, the latency of the load and multiply operations are set to 4 and 2. Due to the small size and high memory dependent characteristic of the loop body, a full DLP mode is selected and each PE group is separated into two logical lanes. Identical schedules based on two PEs are transferred into the loop buffer in the PE group with one cycle difference between logical lane 0 and logical lane 1 (see Figure 7(c)). Different memory operations can execute in the same cycle as shown in cycle 2 but different multiply instructions cannot be scheduled at cycle 7 because logical lane 1 needs to use the multiplier in that cycle.

3.4. Improving ILP Performance

Although modulo scheduling has proven to be an effective solution to exploit ILP over loops, it is not always the best solution because 1) original iteration count is divided by DLP capability, and therefore, the smaller iteration count may not compensate for the prolog and epilog overheads even in moderate DLP loops [23] and 2) sparse interconnection between PEs and no centralized RFs make the quality of the schedule worse. As a result, we suggest supporting list scheduling [6] of the loop body as another option to exploit ILP. When either there is not much total ILP in the loop, or the hardware cannot benefit from increased ILP, list scheduling can outperform modulo scheduling since it does not incur the overhead of modulo scheduling: handling modulo information such as staging predicates.

The remaining problem of adapting list scheduling to hide idle cycles comes from long latency instructions such as multiply and memory operations. To solve this problem, we propose a simple multi-threading scheme with fast context switching. Assuming the Libra architecture supports two threads, a loop with large number of iterations is divided into two threads with identical loops with half number of iterations. The two threads are then executed on the same logical lane. To make the scheme simple, a switch of running threads is allowed only when all the PEs are idle. Each thread has its own register file space divided by the number of threads, similar to what a GPU does, and therefore no context change overhead exists. The schedule with multiple threads is statically decided at compile time. The multi-threading technique is simple but highly effective and is a realistic solution because of the following two reasons: 1) low register pressure: loops with small number of instructions have a small amount of data to save in the register file and list scheduling does not require additional register overhead, and 2) a high chance of hiding latency: this technique is applied only to SIMDizable loops executing on small logical lanes, thus increasing the probability that all FUs are idle.

Although multi-threading looks promising, the Libra architecture faces a number of challenges in reality. There are three essential challenges and we present the lightweight solutions incorporated in the Libra architecture:

Context Saving: The fully distributed nature of Libra allows temporal data to be saved in the register files as well as the output buffer

in order to directly transfer the data between FUs. As a result, the output buffer data of each thread should also be saved in addition to the register files. The register file is divided into the same number of threads. The parts are then addressed by the thread ID. However, the output buffer is originally a simple flip-flop without addressing support. Therefore, it is substituted by an *n*-entry register file addressed by thread ID(*n*: the number of threads supported). The output data can thus remain unchanged when another thread is executed.

Writeback Contention Avoidance: Handling multi-latency instructions is not a simple problem if the output data from a multi-latency instruction is generated when the other thread is executing. To solve this problem, multi-latency FUs need to save the thread ID when the input is issued and be connected to the output buffer (small register file) with an additional port addressed by the original input thread ID. Since only a single additional port is required for multiple FUs with the same latency, the overhead is negligible. For the Libra architecture, only two ports are added to the whole PE group to support a multiplier and a memory controller.

Code Bloat: Since multiple threads are scheduled at compile time, the loop buffer of each PE group needs to contain the entire schedule information of all threads for each cycle. This causes the code bloat problem, requiring an increased loop buffer size which incurs a power overhead. However, an important observation to point out is that the schedules of different threads are essentially the same, just with different execution times. We can, therefore, solve the problem by 1) saving the schedule configuration of only one thread and 2) adding a simple sequence table which contains a thread ID and the corresponding loop buffer address pointing to the actual schedule configuration. The thread controller contains the basic information for supporting multi-threading and the sequence table.

Figure 8 shows an illustration of the Libra architecture with an emphasis on modified features(shaded components) to support multi-threading, assuming that the architecture supports execution of two threads. The loop buffer contains configuration information for only one thread as shown in Figure 8(c). Therefore, its size is the same as when one thread is executed. The thread controller in the cluster has a tiny sequence table containing the actual thread ID and the address of the configuration saved in the loop buffer. Figure 8(b) depicts an example sequence table for two thread execution. Since two threads are executed in this example, the space of RF is divided by two and the output buffer is a 2-entry register file. By reading the sequence table from cycle 0 to cycle 9, the thread controller transfers the thread ID and loop buffer address for each cycle to the loop buffer. From this information, the loop buffer generates the final configuration by reading the appropriate configuration and adding a thread ID to the register file address (see Figure 8(d)). The multiplier gets the thread ID and has a separate data bus due to the multi-latency functionality. When the original configuration B has the multiply operation for FU 2, the result data from thread 0 and B configuration can be stored in the output buffer at cycle 2 without any writeback contention.

3.5. Decision Flow

In order to maximize the performance and resource utilization, the Libra architecture depends on an intelligent selection of the configuration between the number of logical lanes and the size of each logical lane. The system flow is shown in Figure 9. Applications run through a front-end compiler, producing a generic Intermediate Representation (IR), which is unscheduled and uses virtual registers. The compiler also has a high-level machine specific information, including the number of resources, size of register files, the size of a cluster, and the number of supported micro-threads. In addition to this, the compiler needs to have profile information about the iteration counts of loops and memory alias information. Given the IR, hardware and profile information, the compiler categorizes loops into two basic types: SWPable and SIMDizable loops. The compiler then decides the logical lane configuration of a cluster for each loop (resource allocation). If a loop is not SIMDizable but only SWPable, the entire cluster is assigned to the loop. If a loop is proved as SIMDizable, the compiler finds the best configuration based on the provided information such as average iteration count, instruction and dependency information of the loop. Briefly speaking, the compiler tries to fully exploit SIMD parallelism by securing the maximum number of logical lanes without performance degradation due to the instruction imbalance. However, it also performs broad design space explorations by changing the number of logical lanes. This is because 1) sometimes the effectiveness of DLP is not clear when the divided trip count is small and the instruction number is not too small, and 2) the scheduler uses a heuristic way to generate the modulo schedule. After deciding the lane configuration, the compiler chooses the method to exploit ILP inside the logical lane. Finally, the compiler performs modulo scheduling or list scheduling. It then generates the final schedule and the configuration information.

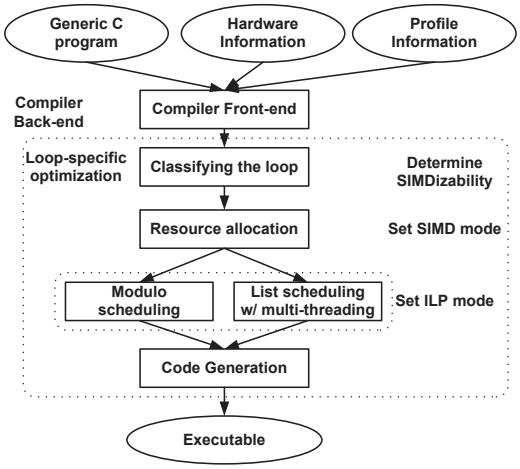


Figure 9: Decision flow of the Libra architecture.

4. Experiments

4.1. Experimental Setup

Target Architecture To evaluate the effectiveness of the Libra architecture, three example implementations with different sizes are used: 16 (one cluster, four PE groups), 32 (two clusters), and 64 (four clusters) PEs. Four FUs per cluster are able to perform load/store instructions to access the data memory with four-cycle latency while another four FUs support two-cycle pipelined multiply instructions. The Libra is compared against two other accelerators in our experiment. We generate 4(cluster)×4(PE), 8×4, and 16×4 heterogeneous VLIWs having the same organization of PEs as corresponding Libra architectures. The wide SIMD architecture as discussed in Section 2.2 is used and the number of SIMD resources can vary from 16 to 64, having the same heterogeneous FU structure.

Target Applications As discussed in Section 2.1, the evaluation is conducted for subsets of three domains. Max 20 top loops having a high execution time are selected for vision and game physics benchmarks, and 144 loop kernels, varying in size from 4 to 142

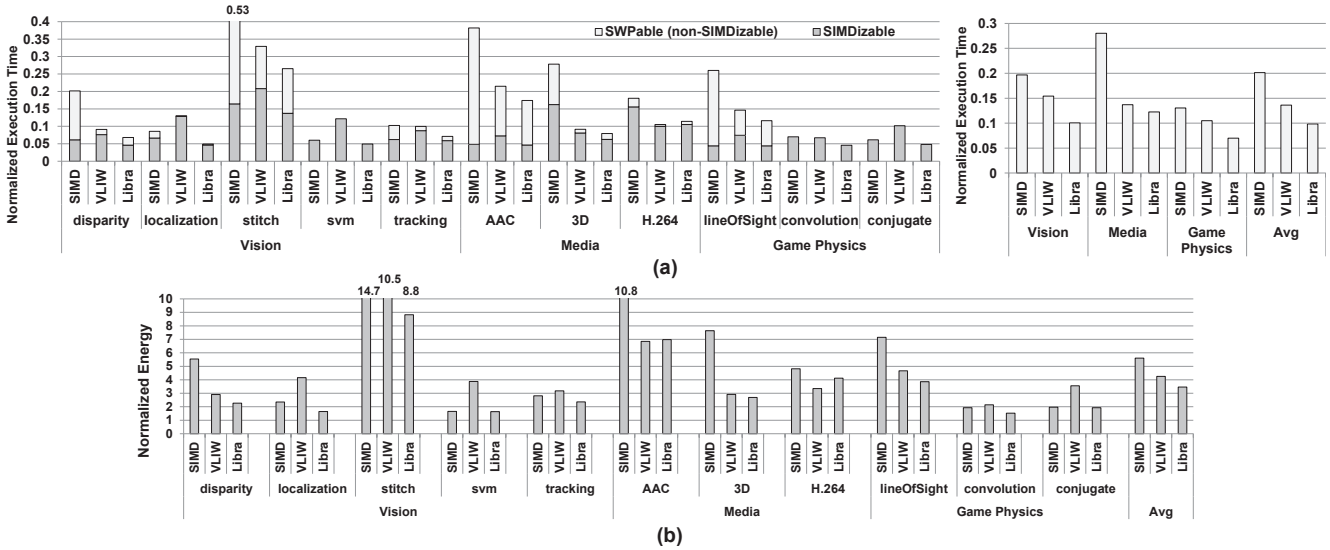


Figure 10: Performance/Energy comparison of 32-PE Libra/SIMD/VLIW architectures: (a) total loop execution time and (b) energy consumption. All the data are normalized to that of a simple in-order core.

operations, are extracted from the media benchmark because the ratio of execution time to the total execution time of the top 20 loops is too small. High number of loops in the media benchmarks and several major loops in the vision benchmarks have conditional statements, while the gaming benchmarks do not have them. In order to eliminate all internal branches, we applied if-conversion for these loops.

Compilation and Simulation The industrial tool chain developed by SAIT [5] is used for compilation and simulation of Libra. The IMPACT compiler [19] is used as the frontend compiler. Basic list scheduler [6], edge-centric modulo scheduling (EMS) [20]-based modulo scheduler, and simple loop-level SIMDization scheduler using a SODA-style [15] wide vector instruction set are implemented in the backend compiler. Based on the original modulo scheduler, we developed a scheduler that can support both flexible execution of Libra and list scheduling with static multi-threading technique. The performance is generated by the cycle-accurate code schedule of loops, accounting for the configuration overhead.

Performance Measurement For fair comparison, both list scheduling and modulo scheduling are applied and the better performing schedule is picked for the SIMD accelerator. For VLIW, loop unrolling is applied when a loopbody size is too small and its resources may not be fully utilized. Multi-threading technique of Libra is also not applied for a fair comparison of the performance of the three architectures. This issue is discussed in Section 4.6.

Power/Area Measurements All architectures are generated in RTL Verilog, synthesized with the Synopsys design compiler, and place-and-routed with the Cadence Encounter using IBM SOI 45nm regular Vt standard cell library in slow operating conditions with a 0.81V operating voltage. Synopsys PrimeTime PX is used to measure the power consumption based on the utilization. The Artisan Memory Compiler is used to determine the area and the power of the memory operation using a 0.81 Volts operating voltage. The target frequency of Libra is 500MHz[2] similar to the latest mobile GPUs.

[2]The FO4 delay of this process is about 13ps.

4.2. Performance/Energy Evaluation

We compared the performance of a 32-PE Libra architecture with identically sized VLIW (8×4) and SIMD(32-wide) architectures. Performance results are measured as the total loop execution time when each loop is scheduled by the method the target architecture supports. Figure 10(a) shows a plot comparing the performance of the three architectures normalized to the simple 1-issue inorder core. For individual benchmarks, the graph also indicates the fraction of two different loop categories: SIMDizable and SWPable loops.

For benchmarks with a high ratio of non-SIMDizable loops such as stitch, AAC, and lineOfSight, SIMD shows severe performance degradation, whereas VLIW and Libra show a fair performance improvement. Libra outperforms even VLIW because it can accelerate SIMDizable regions more efficiently. On the other hand, both the SIMD and Libra deliver a substantial performance improvement for benchmarks with mostly SIMDizable loops, while VLIW suffers. The Libra also shows better performance than SIMD because it effectively accelerates applications having low-SIMDizable loops (3D, H.264) and its ILP capability also helps Libra to adequately tolerate the lack of expensive resources for high-SIMDizable loops (convolution, conjugate). Overall, Libra shows the best performance in all benchmarks except H.264 benchmark. This is because of the slightly lower performance gain on SWPable regions due to its distributed nature. Among average result of each domain, performance gain of Libra is the highest on game physics. As a result, Libra shows a performance gain of 2.04x and 1.38x over SIMD and VLIW, respectively.

Despite using the same amount of computation resources, performance-only comparison may not be fair due to the different interconnection strategies among the architectures. An energy comparison may yield a better comparison considering both performance and hardware overhead. Figure 10(b) shows the energy consumption of three architectures and the results are also normalized to the 1-issue core. This graph shows a similar trend to Figure 10(a). On average, even though SIMD added extra logics for handling sharing resources (Figure 5(b)), VLIW shows 16% more power consumption because of bigger RFs and complex control logics, and Libra shows 20% more power consumption due to more interconnects and

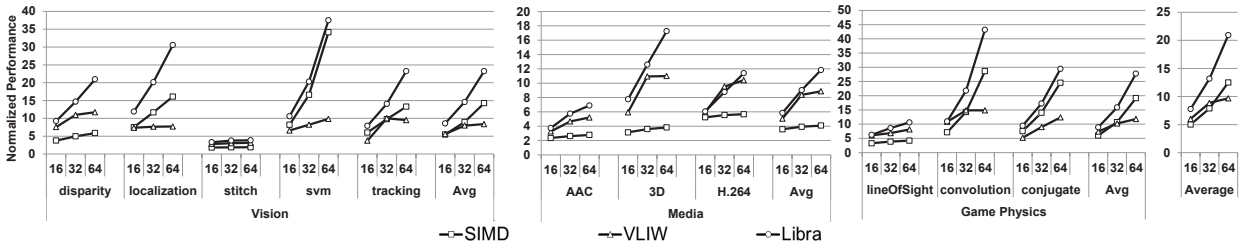


Figure 11: Scalability of Libra/SIMD/VLIW architectures: the Libra architecture is highly scalable for most of benchmarks, while SIMD and VLIW cannot be scalable for several benchmarks.

Libra-specific overhead such as a loop-buffer and a thread controller. Based on these power differences, the Libra saves 38% and 19% energy compared to SIMD and VLIW, respectively[3]. As a result, the Libra architecture shows a fair amount of performance improvement in addition to high energy efficiency by providing a more suitable acceleration scheme for each loop.

4.3. Scalability

Figure 11 shows the performance of each architecture normalized to a 1-issue core for different sizes across three benchmark domains. The number of PEs varying from 16 to 64 are shown on the X-axis. The results show high scalability of the Libra architecture in all benchmark domains.

In the vision and game domain benchmarks, applications are not specially optimized to the SIMD-style architecture, but the performance is highly scalable as the number of PEs increases because most loops are simple and highly SIMDizable. Only the stitch is barely scalable because the application is mostly sequential as the dominating loop has only a small number of iterations. In the media domain, the Libra accelerator performance also fairly increases as it scales to more PEs. Compared to other architectures, VLIW performance results are frequently saturated because modulo scheduling of a big size loopbody(often unrolled) on a large number of PEs is too complex to exploit ILP, while Libra solves this problem by scheduling a small loopbody in a small logical lane and applying the same schedule to multiple logical lanes. The SIMD results are also constrained by lack of expensive resources and program complexity. To summarize, the Libra architecture can increase its performance with larger resources when the application has enough total ILP/DLP parallelism.

4.4. From the Homogeneous SIMD to the Heterogeneous Libra

Section 4.2 and 4.3 evaluate three different architectures consisting of the same computation resources. The key question here is how much Libra surpasses the traditional SIMD architecture. To answer this question, we compared the performance and energy consumption of the heterogeneous Libra and the homogeneous SIMD. The heterogeneous Libra has a quarter of memory/multiply resources and the homogeneous SIMD has the same number of memory/multiply resources as the total number PEs. Figure 12 shows the average of relative performance and energy consumption of Libra over SIMD for different sizes. In terms of performance, Libra outperforms SIMD and the difference increases in proportion to the size (Figure 12(a)). This is because 1) the lack of expensive resources can be effectively compensated for by forming logical lanes and 2) the lane utilization of the traditional SIMD is lower for a larger size due to the program characteristics.

In terms of the energy consumption, Libra still shows similar results as its performance improvement because significantly less computational units can reduce the overall power overheads, and the result is better on larger size. For example, the 32-PE heterogeneous Libra consumes 11% more power than the same size homogeneous SIMD due to 12% power savings on FUs with 23% overheads (Figure 12(c)). On average, Libra shows 101%, 71%, and 56% energy consumption compared to the traditional SIMD.

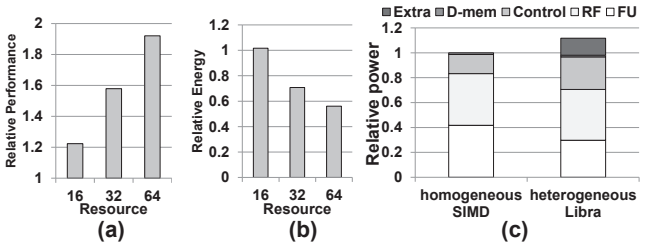


Figure 12: Performance/energy improvement of the heterogeneous Libra over the same sized homogeneous SIMD: (a) performance, (b) energy consumption, and (c) power breakdown with five categories: FU, RF, control logic, memory, and architecture specific additional logic.

4.5. Acceleration Mode Selection

Our experiments so far have focused on the overall performance of the Libra architecture compared to other architectures, showing considerable performance enhancement. In this section, we evaluate the effectiveness of flexible lane mapping to answer the question if Libra really needs to provide various intermediate sizes of logical lanes between SIMD and VLIW. Figure 13(a) shows the execution time distribution at different logical lane sizes for the three application domains on the 16, 32, and 64-PE Libra. On average, all available modes are used for considerable fraction of time and no dominating logical lane size exists, which proves the effectiveness of flexible lane mapping. Furthermore, the lane sizes are selected adaptive to the domain characteristics. For vision benchmarks, 2-PE small sized logical lane is dominant because most loops are small and memory operation dominant. In media benchmarks, large logical lanes are used for a high fraction of the execution because of lack of DLP. Game physics uses a 4-PE logical lane in substantial fraction to execute high-DLP loops with some ILP. Figure 13(b) compares the normalized performance of Libra to that when only one specific logical lane configuration is allowed to execute benchmarks. The results of this graph further prove the effectiveness of flexibility by showing that any fixed mode execution cannot win over the flexible execution.

[3]Figure 10(b) does not mean that a simple 1-issue core is 3x energy efficient than Libra because the performances are different. For a performance-equivalent comparison, Libra is much more efficient than the simple core.

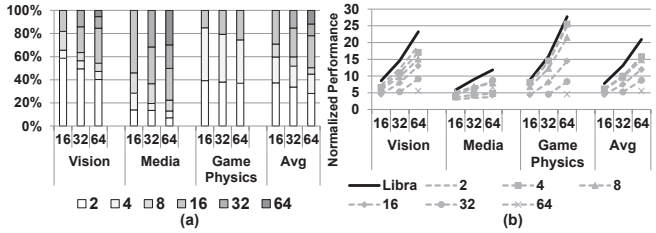


Figure 13: Mode selection: (a) execution time distribution at different logical lanes, (b) flexible vs. fixed execution.

4.6. Multi-threading Effectiveness

As discussed in Section 3.4, a simple multi-threading functionality is added to Libra. In this section, we evaluate the effectiveness of this functionality. Figure 14(a) shows the performance improvement on SIMDizable loops only, since this technique can be only applied to SIMDizable loops. On average, a performance gain of 12-16% is achieved, and this is up to 28% more effective in vision benchmarks because the majority of loops are small and multi-threading is most effective in small size logical lane mapping. Figure 14(b) shows the execution time distribution for different logical lane sizes when multi-threading is applied. Compared to Figure 13(a), a substantial amount of 2 and 4-PE logical lane execution is substituted with multi-threading. Overall, multi-threading is effective for small logical lanes when executing SIMDizable loops.

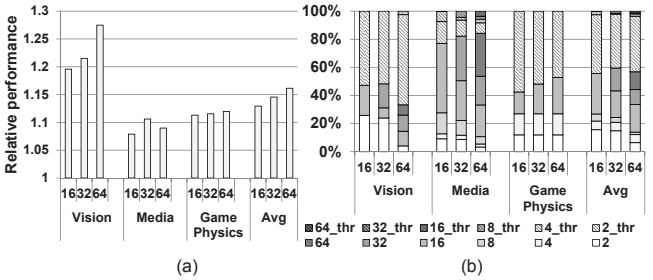
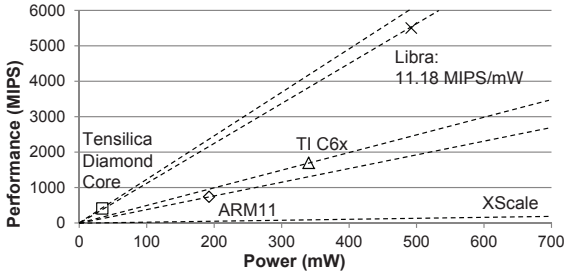


Figure 14: Multi-threading effectiveness: (a) performance improvement for SIMDizable loops, (b) execution time distribution at different logical lanes.

4.7. Power and Area Measurement

We measured the average power when the 32-PE Libra architecture executes the H.264 benchmark at 500 MHz. A power and an area consumption breakdown for various components that are part of the architecture are shown in Figure 15(b). Compared to the normal SIMD, the power consumption of the routing logic is larger due to its dynamic configurability, but FU power is smaller due to the smaller number of expensive units. A SIMD controller and four loop buffers, and a thread controller are added to a cluster. The power consumption of a SIMD controller and four loop buffers is substantial because the loop buffer is implemented as 64-entry wide two-port SRAM and the data is read every-cycle. In addition to this, the thread controller also consumes 0.7% of total power because the sequence table is a 256 entry 8 bit two-port SRAM. The total area of the 32-PE Libra architecture is 2.0 mm^2.

Based on the power and performance data, we compared the efficiency of Libra to other architectures using data shown in [11]. Based on Figure 15(a), the Libra architecture achieves 11.18 MIPs/mW and most of the other well-known solutions show lower efficiency. The Tensilica Diamond Core is slightly more efficient than the Libra architecture, but the actual performance is not enough to successfully execute compute-intensive media applications.



(a)

Component	Power(mW)	Ratio(%)	Area(um^2)	Ratio(%)
SIMD FUs	131.3	26.7%	341909	17.1%
SIMD RFs	180.2	36.6%	405963	20.3%
SIMD Pipeline + Routing + Scalar Pipeline	115.5	23.5%	117721	5.9%
Instruction Control (SIMD controller + Loop buffer)	56.0	11.4%	471984	23.6%
Thread controller	3.2	0.7%	37714	1.9%
D-mem (64kB)	5.9	1.2%	626550	31.3%
Total	492.2	100.0%	2001840	100.0%

(b)

Figure 15: (a) Power/Performance comparison, and (b) power and area breakdown of the 32-PE Libra architecture.

5. Related Works

Many previous works have focused on accelerators to address the challenges of improving computing efficiency. Some exploit only one type of parallelism and others introduce some flexibility to support more than one type of parallelism. Figure 16 compares and shows the major differences between Libra and prior works.

		ILP	DLP	Heterogenity	Configurable Performance	Scalability	Power Efficiency
DLP Accelerator	SIMD	No	High	No	No	High	High
	GPU	Low	High	Limited	No	High	Low
	Embedded GPU	Low	High	Limited	No	High	High
ILP Accelerator	ADRES	High	No	Yes	No	Low	High
DLP + ILP Accelerator	Imagine	High	High	Yes	No	High	Low
Flexible Accelerator	AnySP	Low	High	No	Limited	High	High
	SIMD-Morph	High	High	No	Limited	Low	High
	TRIPS, SCALE	High	High	Yes	Yes	High	Medium
	Libra	High	High	Yes	Yes	High	High

Figure 16: Comparison to prior work

Accelerators for multimedia usually focus on one type of parallelism without adaptive configuration. Conventional SIMD [9, 15] only supports DLP and misses the opportunity of improving performance with other form of parallelism. By Amdahl's law, low-DLP regions quickly become the bottleneck of applications. Conventional SIMD also wastes expensive resources due to imbalanced utilization. While the latest GPUs [18, 17] support the limited level of heterogeneity and embedded GPUs such as Qualcomm Adreno [4] and ARM Mali [1] are power-efficient, GPUs have the same fundamental weakness as other data-parallel accelerators.

ILP accelerators, such as ADRES [16], tackle the problem in another way by exploiting ILP with the help of modulo scheduling. Even though it has high scalability by providing distributed architecture, the throughput quickly saturates as the number of resources increases due to the scheduling difficulty as shown in PPA [21]. Hybrid accelerators such as the Stanford Imagine [7] use the VLIW-SIMD scheme but the fixed configuration frequently incurs a lack or waste of resources.

Recently, several architectures have tried to embrace flexibility in a conventional SIMD accelerator in order to support multiple application domains with different characteristics. AnySP [27] targets mobile applications such as 4G wireless communication and high-definition video coding. AnySP achieves the goal efficiently by simply chaining two SIMD lanes and supporting limited thread level parallelism, but underutilization in low-DLP loops is still inevitable due to the lack of general policy to support ILP. SIMD-Morph [10] employs subgraph matching to accelerate sequential code region. Despite their fair performance gain, their simple ILP/DLP mode transition policy cannot adaptively adjust the degree of ILP and DLP inside a specific code region. For example, it is impossible to fully utilize the SIMD-Morph for a low-DLP code region since an insufficient degree of DLP cannot be supplemented by ILP exploitation, while Libra can. In addition, they are still homogeneous SIMD, and therefore, cannot improve utilization and power efficiency.

TRIPS [25] and SCALE [14] are also similar to this work. TRIPS integrates ILP, DLP and TLP, and SCALE exploits both vector parallelism and TLP. They are targeting more the desktop/server space, and therefore, need expensive architectural features such as inter-cluster networks, additional multiple fetch units, and specialized caches for generality. However, Libra focuses on more efficient execution of loops with minimal hardware modifications.

Avoiding resource contention of expensive instructions by pipelined execution is also introduced in an instruction-systolic array architecture [22]. However, systolic execution may incur severe performance degradation on high number of PEs because of the pipelining delay, while Libra limits sharing only between two logical lanes in full DLP mode.

6. Conclusion

The popularity of mobile computing platforms has led to the development of feature-packed devices that support a wide range of software applications with high single-thread performance and power efficiency requirements. To efficiently achieve both objectives, SIMD-based architectures are currently proposed. However, the SIMD is not able to efficiently support a wide range of mobile applications due to several limiting factors: limited availability of high trip count vector loops and the homogeneous nature of the hardware. To enhance the applicability of SIMD and improve its inherent energy efficiency, we break two long-standing traditions of SIMD design: identical lanes and static configuration. The *Libra* accelerator adapts the SIMD lane resources to target application. The Libra architecture customizes the lane configuration based on the loop structure from many resource-constrained logical lanes for highly data-parallel loops, to a modest number of lanes with moderate resources, up to a single resource-rich logical lane that is effectively a multicluster VLIW. A 32-PE Libra system achieves an average 1.58x speedup over the traditional SIMD system, and the gain becomes higher as the number of PEs increases. Through a judicious mechanism to share expensive resources, Libra also achieves a 29% reduction in energy compared to the SIMD system. We believe that as industry requires higher performance with high energy efficiency, the proposed scalable architecture puts more resources to work in order to meet this demand.

7. Acknowledgments

Thanks to Gaurav Chadha, Anoushe Jamshidi, Dongsuk Jeon and Yoonmyung Lee for all their help and feedback. We also thank Krste Asanovic for shepherding this paper. This research is supported by Samsung Advanced Institute of Technology and the National Science Foundation under grants CCF-0916689 and CNS-0964478.

References

[1] ARM Mali Graphics Hardware - http://www.arm.com/products/multimedia/mali-graphics-hardware/.
[2] Cuda toolkit. - http://developer.nvidia.com/cuda-toolkit.
[3] Glbenchmark - http://www.glbenchmark.com/.
[4] Qualcomm Adreno - http://www.qualcomm.com/solutions/multimedia/graphics/.
[5] Samsung advanced institute of technology - http://www.sait.samsung.co.kr/.
[6] T. Adam, K. Chandy, and J. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12):685–690, Dec. 1974.
[7] J. H. Ahn et al. Evaluating the Imagine stream architecture. In *Proc. of the 31st Annual International Symposium on Computer Architecture*, pages 14–25, June 2004.
[8] M. Alvarez, E. Salami, A. Ramirez, and M. Valero. A performance characterization of high definition digital video decoding using h.264/avc. In *2005 IEEE International Symposium on Workload Characterization*, pages 24–33, Oct. 2005.
[9] H. Bluethgen, C. Grassmann, W. Raab, and U. Ramacher. A programmable platform for software-defined radio. In *Intl. Symposium on System-on-a-Chip*, pages 15–20, Nov. 2003.
[10] G. Dasika, M. Woh, S. Seo, N. Clark, T. Mudge, and S. Mahlke. Mighty-morphing power-simd. In *Proc. of the 2010 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, Oct. 2010.
[11] K. Fan, M. Kudlur, G. Dasika, and S. Mahlke. Bridging the computation gap between programmable processors and hardwired accelerators. In *Proc. of the 15th International Symposium on High-Performance Computer Architecture*, pages 313–322, Feb. 2009.
[12] Intel. Intel compiler, 2009. software.intel.com/en-us/intel-compilers/.
[13] H. Kalva. The H.264 video coding standard. *IEEE MultiMedia*, 13(4):86–90, 2006.
[14] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic. The vector-thread architecture. In *Proc. of the 31st Annual International Symposium on Computer Architecture*, 2004.
[15] Y. Lin et al. Soda: A low-power architecture for software radio. In *Proc. of the 33rd Annual International Symposium on Computer Architecture*, pages 89–101, June 2006.
[16] B. Mei et al. ADRES: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix. In *Proc. of the 2003 International Conference on Field Programmable Logic and Applications*, pages 61–70, Aug. 2003.
[17] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.
[18] NVIDIA. GeForce GTX 200 GPU architectural overview, 2008. http://www.nvidia.com/docs/IO/55506/GeForce_GTX_200_GPU_Technical_Brief.pdf.
[19] OpenIMPACT. The OpenIMPACT IA-64 compiler, 2005. http://gelato.uiuc.edu/.
[20] H. Park, K. Fan, S. Mahlke, T. Oh, H. Kim, and H. seok Kim. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proc. of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 166–176, Oct. 2008.
[21] H. Park, Y. Park, and S. Mahlke. Polymorphic pipeline array: A flexible multicore accelerator with virtualized execution for mobile multimedia applications. In *Proc. of the 42nd Annual International Symposium on Microarchitecture*, pages 370–380, Dec. 2009.
[22] J. Park, H. Yang, G. Park, S. Kim, and C. C. Weems. An instruction-systolic programmable shader architecture for multi-threaded 3d graphics processing. *Journal of Parallel and Distributed Computing*, 70(11):1110–1118, 2010.
[23] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, Nov. 1994.
[24] R. M. Russell. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72, Jan. 1978.
[25] K. Sankaralingam et al. Exploiting ILP, TLP, and DLP using polymorphism in the TRIPS architecture. In *Proc. of the 30th Annual International Symposium on Computer Architecture*, pages 422–433, June 2003.
[26] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. L. S. Garcia, S. Belongie, and M. B. Taylor. SD-VBS: The san diego vision benchmark suite. In *2009 IEEE International Symposium on Workload Characterization*, pages 55–64, Oct. 2009.
[27] M. Woh, S. Seo, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. AnySP: Anytime Anywhere Anyway Signal Processing. In *Proc. of the 36th Annual International Symposium on Computer Architecture*, pages 128–139, June 2009.