GPU

Seminar Report

Submitted in partial fulfillment of the requirements
for the degree of

Master of Technology

by

Hitesh Kumar Sahu
(Roll No. 213079013)

Under the guidance of

Prof. Virendra Singh

Department of Electrical Engineering
Indian Institute of Technology Bombay
September 2022

Acknowledgement

I express my gratitude to my guide Prof. Virendra Singh for providing me the opportunity to work on this topic.

Hitesh Kumar Sahu
Electrical Engineering
IIT Bombay

Contents

List of Figures

Chapter 1

Completed Work

Hello

1.1 Analyzing and Leveraging Decoupled L1 Caches in GPUs

1.1.1 Introduction

Graphics Processing Unit (GPU) architectures are a high-performance parallel computing systems. GPU provides very fast and energy efficient execution for many general purpose applications which can be paralleled. GPU employs a conventional two-level cache hierarchy where each core incorporates a private L1 cache and all the GPU cores are connected via a Network-on-Chip (NoC) to a shared and banked L2 cache. But the conventional cache hierarchy system in GPU results in inefficient utilization of the valuable on-chip L1 caches mainly due to tight coupling between L1 cache and the GPU cores.

In this Paper, it is proposed to separate L1 caches from the GPU cores i.e. Decoupled L1 caches (DC-L1).Each DC-L1 cache is accessed by multiple GPU cores. Aggregating DC-L1 caches improves their individual bandwidth utilization and reduces data replication across the DC-L1s as more cores are accessing a given DC-L1. Designing the DC-L1 caches and the NoC to build a shared DC-L1 cache organization that eliminates cache line replication across the DC-L1 caches is proposed.

Shared DC-L1 organization has some serious drawbacks e.g. NoC area/power overheads, scalability, and clocking challenges. To eliminate the drawbacks a clustered shared DC-L1 cache design is proposed that limits data replication. This cache design enables a cluster of GPU cores to access a cluster of shared DC-L1 caches, thus eliminating data replication within the cluster and reducing it across the GPU.

1.1.2 Literature Survey

Inefficiency of Tightly Coupled L1 Caches

1. Cache Line Replication across L1: On a miss in the baseline private L1 cache, each core independently fetches the required data from the L2 cache. This may lead to replication across L1s if the cores request the same cache line, leading to wasted L1 cache capacity.

2. Low L1 Cache Utilization: The tight coupling of the L1 caches and GPU cores along with the many-to-few communication pattern (between the L1s and the L2

banks) puts more pressure on the few L2 banks and less pressure on the many L1 caches. This leads to low bandwidth utilization of the per-core L1 caches.

Decoupled-L1 Cache and NoC Design

- In Figure 1.1 (A) shows the DC-L1 node design. A DC-L1 node simply contains the DC-L1 cache , two queues to handle the traffic from/to the GPU core, and two queues to handle the traffic to/from the L2 and memory partitions. A GPU core is a lite core which is similar to the baseline GPU core but without the L1 data cache and the associated Miss Status Holding Registers (MSHR).
- The NoC is divided into two parts due to DC-L1 cache introduction. The first NoC (B) connects the GPU cores and the DC-L1 nodes. The second NoC (C) connects the DC-L1 nodes and the L2 memory. The design of both NoCs is determined by the number of DC-L1 nodes and the cache organization.

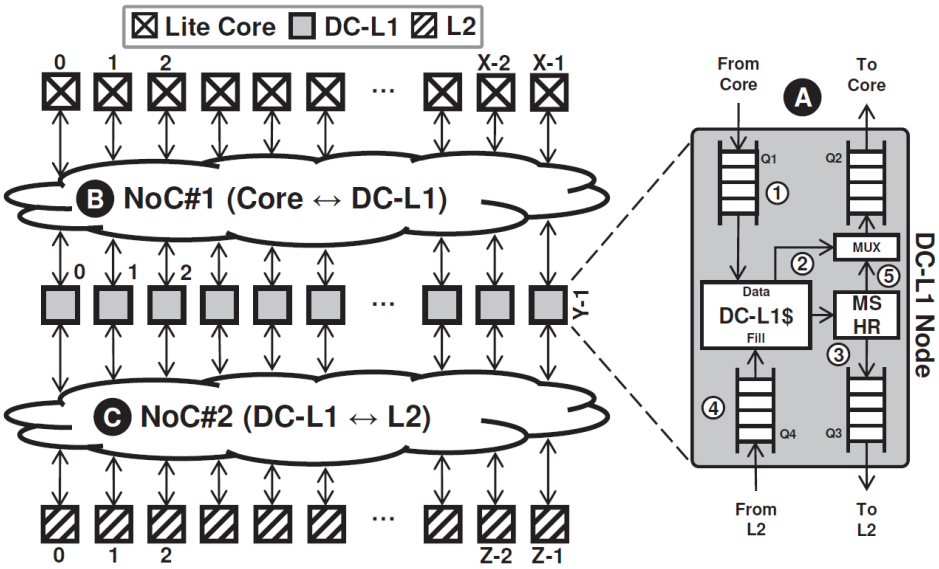

Figure 1.1: Decoupled-L1 node and NoC design

Private DC-L1 Cache

- Given a system with X GPU cores and X DC-L1 nodes, where each DC-L1 node hosts a single DC-L1 cache with size C, we aggregate these X DC-L1 cache to form Y bigger DC-L1 cache (X Y). Each of the Y larger DC-L1 cache has a size of (X $\times$ C)/Y and is hosted in a DC-L1 node.
- Under this design, each DC-L1 node is accessed privately by a group of N = X/Y cores via an N $\times$ 1 crossbar in NoC#1. We refer to this private aggregated DC-L1 design as PrY .

Shared DC-L1 Design

- A shared DC-L1 organization requires, any core can access to any DC-L1 node. Figure 1.3 shows one possible design. In this design, 80 GPU cores are connected to 40 DC-L1 nodes via an 80$\times$40 crossbar in NoC#1. We refer to this design as Sh40 (or ShY in general, where Y is the total number of DC-L1 nodes).

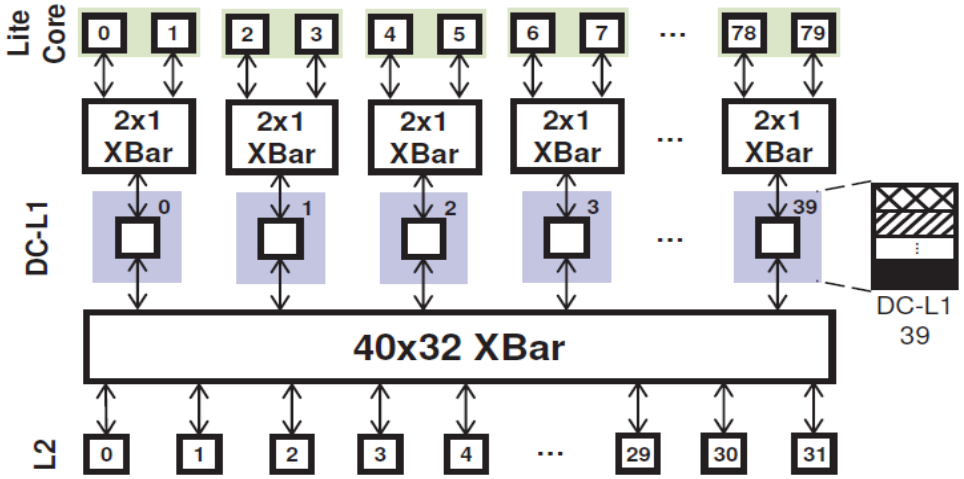

Fig. 5: Pr40 design.

Figure 1.2: Pr40 design

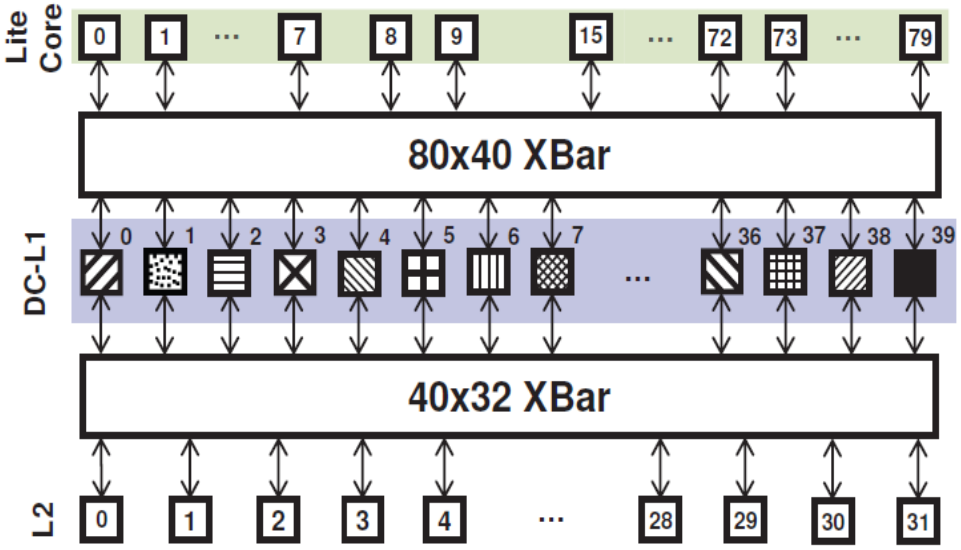

Figure 1.3: Sh40 design

Clustered DC-L1 Design

- The NoC area and power overhead of Sh40 (80×40 crossbar) used in NoC#1 is overpowering. Though Sh40 is capable to enable the fully-shared cache organization. On the other hand, with Pr40, replication is still high, but the overall NoC area and static power is reduced. This presents a trade-off between the reduction in replication and the NoC area/power.

- In clustered design shown in Fig 1.4, a cluster of M DC-L1 nodes is accessed by N cores via an N × M crossbar in NoC#1. We refer to this design as ShY +CZ, where Y is the total number of DC-L1 nodes and Z is the number of clusters (Z = Y/M). The L2 slices and that each DC-L1 within a cluster is assigned a unique address range due to the shared nature of both L1 and L2 caches, a given DC-L1 will communicate only with a few L2 slices. Therefore, instead of using a full Y × L crossbar in NoC#2 to connect the Y DC-L1 nodes to the L L2 slices (L = M, L mod M = 0), a given DC-L1 will communicate only with O = L/M L2 slices via an

Z × O crossbar in NoC#2.

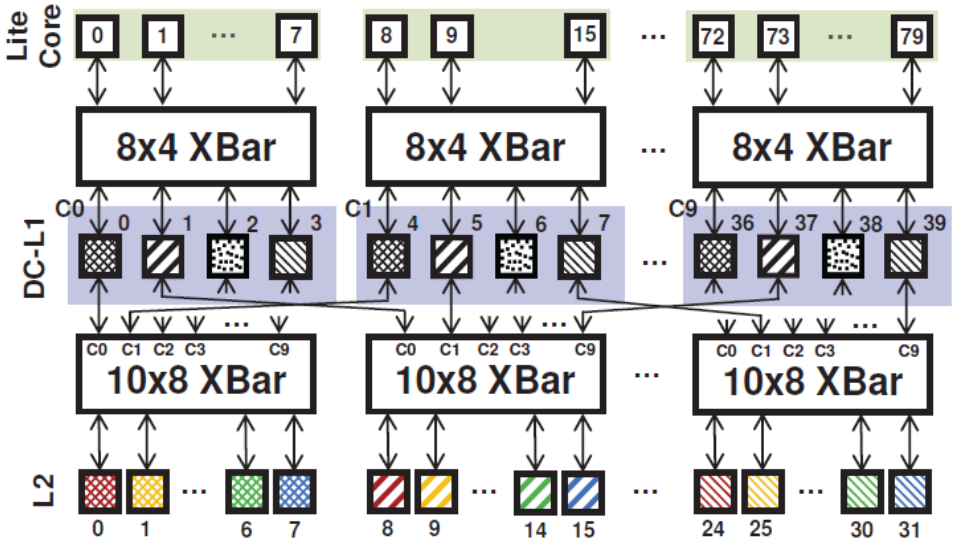

Figure 1.4: Sh40+C10 design

Experimental Results

- **Pr40:** Private DC-L1 cache design reduces the number of Dc-L1 nodes while maintaining the total L1 capacity but the data replication is still quite similar to baseline private L1 cache.
- **Sh40:** A fully Shared DC-L1 cache design eliminates the data replication but with the cost of more NoC area and power dissipation.
- **Sh40+C10:** Cluster-based DC-L1 cache organization eliminate data replication within the cluster and comparatively reduced NoC area and power consumption than Sh40.

1.1.3 Conclusion

- Introduced an L1 cache decoupled from the GPU core i.e. Decoupled L1 (DC-L1) cache.
- High bandwidth utilization of the L1s overcoming cache line replication across the L1 caches.
- A clustered-based DC-L1 cache organization was introduced , where a cluster of GPU cores access a cluster of shared DC-L1s.

1.2 PEPSC: A Power-Efficient Processor for Scientific Computing

1.2.1 Introduction

Earlier periods, scientists have greatly relied on large-scale supercomputers to deliver the computational power to solve the scientific problems. In current structure, GPU provides a more attractive options in terms of cost and bulkiness compared to supercomputers. GPU also present a better performance , with speed ups up to 300X over conventional CPUs for applications like medical imaging, electronic design automation, physics simulations, and stock pricing models.

Why a New Architecture

CPUs are more focused on scalar program performance and do not have sufficient raw floating-point computation resources. Though GPUs have the resources to compute floating-point computations but suffer from two major problems that limit scalability as well as the ability to deliver the promised throughputs for a wide range of applications: high power consumption, long memory access latencies and generalizing for any scientific computations.

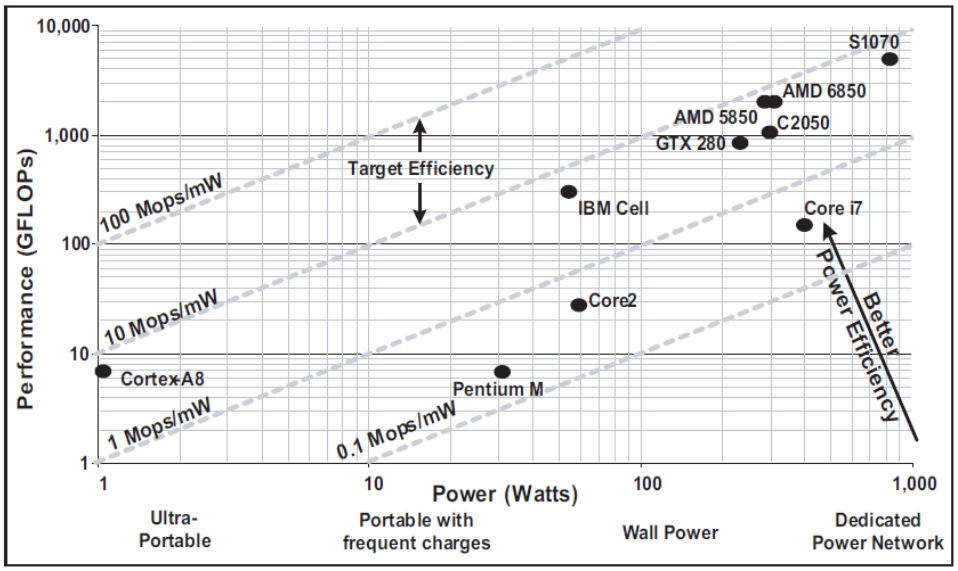

Figure 1.5: Peak Performance/Power for various CPUs and GPUs

- the Intel Xeon X7560, has a peak performance of 144 GFLOPs with a total power dissipation of 130 Watts.
- The AMD Radeon 6870 graphics processing unit (GPU) can deliver a peak performance of nearly 2 TFLOPs with a total power dissipation of 256 Watts.

From figure 1.5 we can deduce that as the performance increases the power also increases drastically. So even performance increases the power efficiency still remains low.

1.2.2 Literature Survey

Contribution of the paper:

1. Analysis of the efficiency bottlenecks of current GPUs on dense matrix scientific applications.
2. A michroarchitectural mechanisms to overcome GPU inefficiencies due to data execution, memory stalls and control divergence i.e PEPSC.
3. An analysis of the performance and power-efficiency of the PEPSC architecture across a range of scientific applications.

GPU Utilization for Different Benchmarks

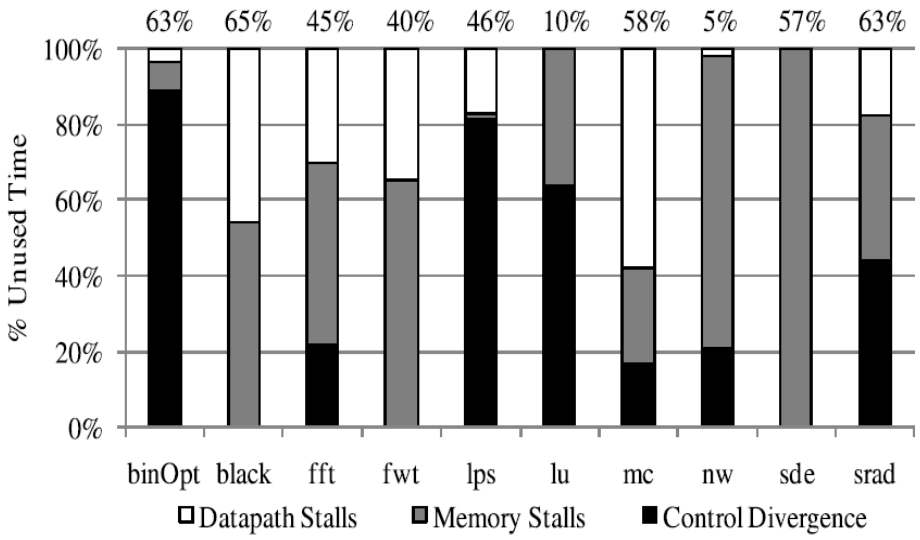

Figure 1.6: Benchmark Utilization on an Nvidia GTX 285 model

- The benchmarks' behavior on GPUs using the GPGPU-SIM simulator. The configuration is very similar to the GTX 285 GPU, the most recent GPU that uses the FX5800's microarchitecture.
- Figure 1.6 illustrates the performance of each of the benchmarks and the sources of underutilization. "Utilization" is the percentage of the theoretical peak performance of the simulated architecture actually achieved by each of these benchmarks.
- On average, these applications make use of around 45% of the GPU compute power. The extent to which specific causes lead to underutilization also varies from one application to another.

The mains reasons of underutilization of the GPU by these benchmarks:

1. **Datapath Stalls:**
   (a) Datapath stalled for various reasons such as read-after-write hazards.
   (b) Big concern in very deep floating-point pipelines.
2. **Memory Stalls**:

- Numerous thread contexts hides latency by issuing a different warp when one warp is waiting on memory.
- The amount of time fetching the data for some benchmarks is still high.

3. **Control Divergence:**
   - Thread divergence breaks the concurrent execution of single instruction to back-to-back execution.
   - Reduces the overall utilization of the processor.

The PEPSC Architecture

The PEPSC architecture is presented in Figure 1.7.

1. A wide SIMD machine to effectively exploit data-level parallelism.
2. A scalar pipeline for non-SIMD operations such as non kernel code and incrementing loop-induction variables.
3. A dedicated address generation unit (AGU).
4. Special function units for math library functions such as sine, cosine and divide.

It also provide some additional techniques to improve performance.

- A two-dimensional design that extracts power efficiency from both the width and the depth of a SIMD datapath.
- Fine-grain control of the SIMD datapath to mitigate the cost of control divergence.
- A dynamically adjusting prefetcher to mitigate memory latency.
- An integrated reduction floating-point adder tree for fast, parallel accumulation with low hardware overhead.

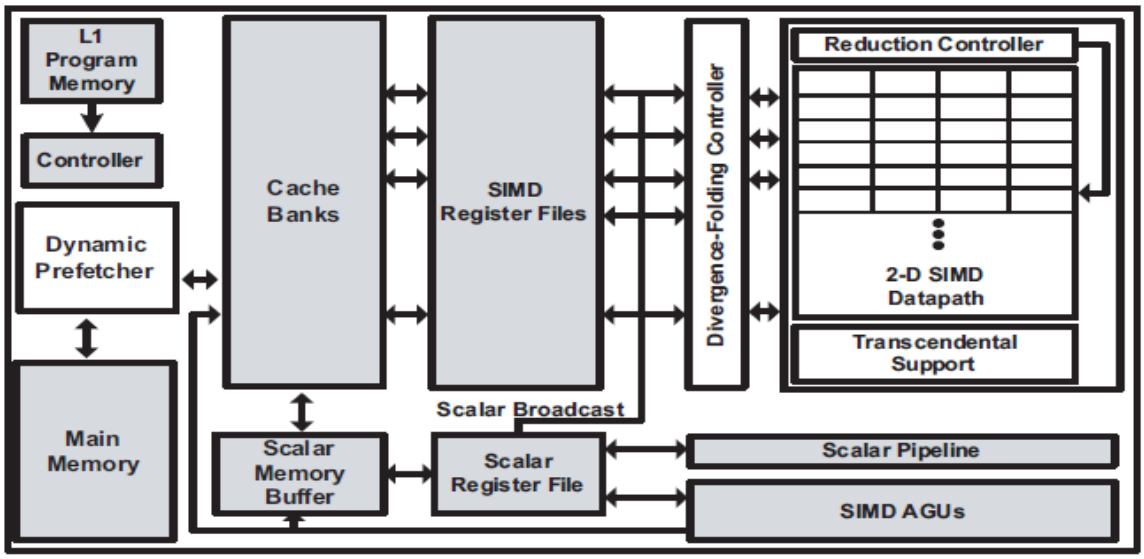

Figure 1.7: PEPSC architecture

Two Dimensional SIMD Datapath

Conventional FPU Structure

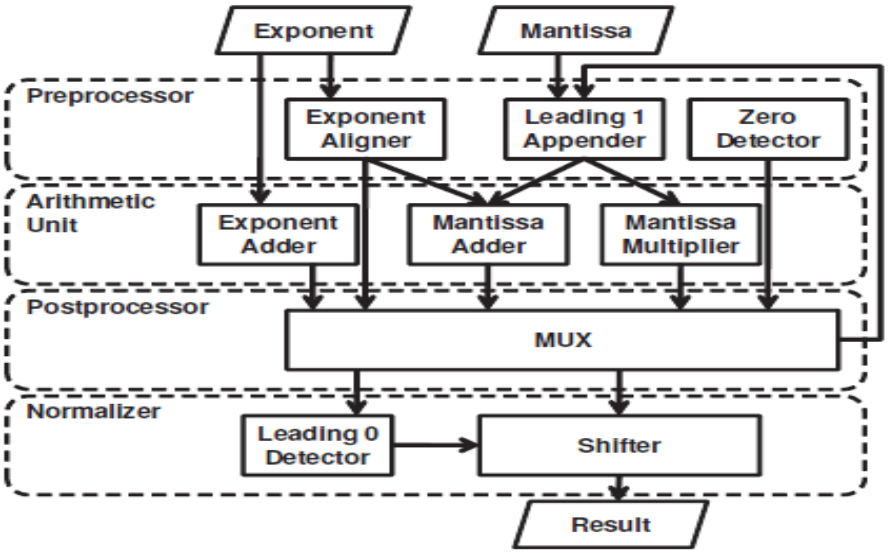

Figure 1.8: FPU internal Architecture

- A typical FP adder consists of a zero detector, an exponent aligner, a mantissa adder, a normalizer, and an overflow/underflow detector.
- A FP multiplier generally consists of a zero detector, an exponent adder, a mantissa multiplier, a normalizer, and an overflow/underflow detector.

Chained FPU Architecture

- The normalizer after each FPU unit is removed as its is chained all the operation can execute one by one and finally use a normalizer to convert the value in IEEE 754 format.
- The normalizer consumes a significant amount of computation time nearly 30% ,so its removal results in marked performance improvements.
- Extra logic that is necessary to make chaining work properly includes the shifter placed in the postprocessor to process varying result widths, resolving temporary overflows, and detecting the true location of leading ones so that they may be correctly appended for the next stage of the computation.
- **Reducing FPU latency:**
  - Scientific applications are typically 3 to 5 back-to-back FP computations.
  - Chaining of FPUs allow for back-to-back execution of FP operations.
  - Thus improve performance and power efficiency.
  - *Chain coalescence:* FPU design have multiple independent 2- or 3-long FPU chains to execute in parallel.
- **Hiding FPU latency:**
  - Chaining depth increases the pipeline depth and hence increased latency.
  - Compiler-directed approach for hiding latency by software pipelining the innermost loops and overlapping independent successive iterations.

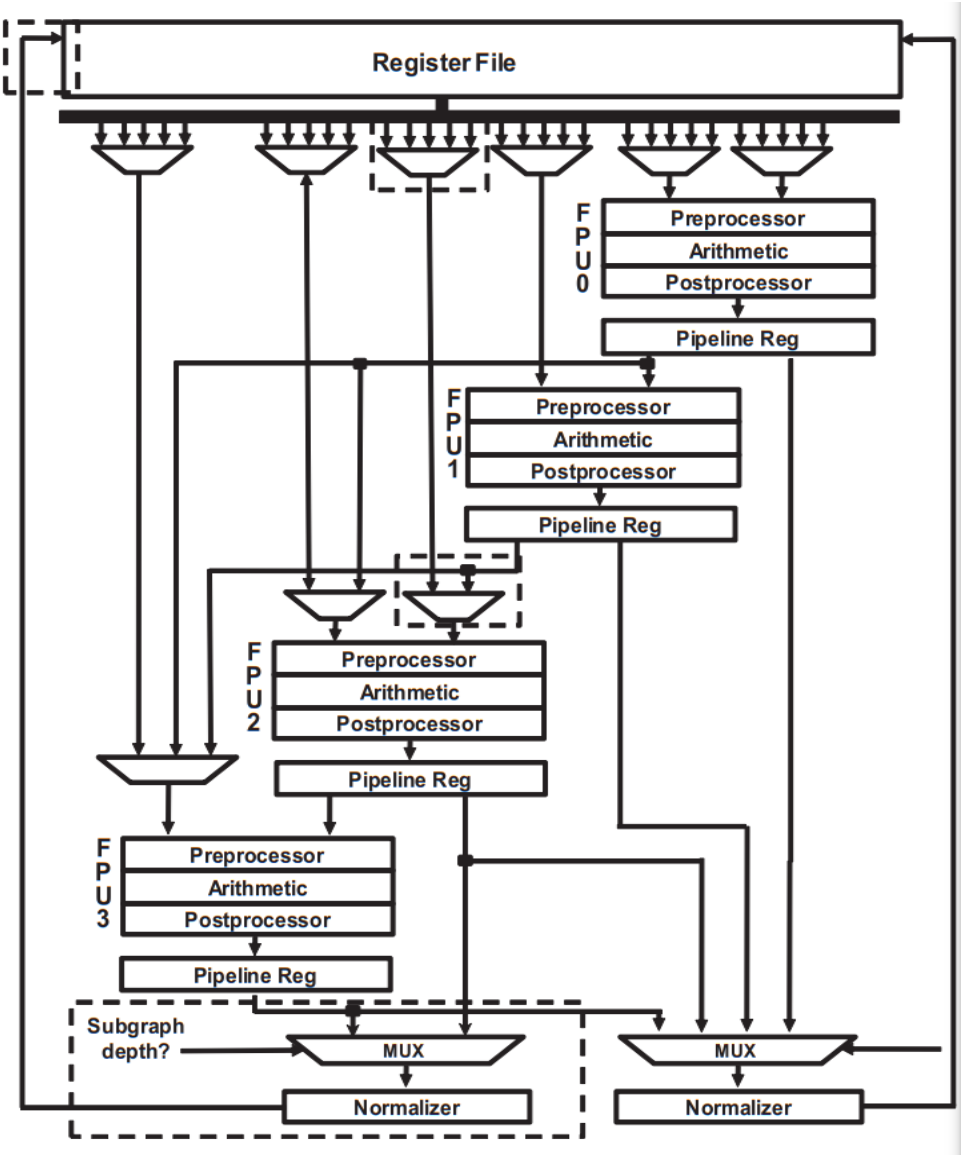

Figure 1.9: Chained FPU design

Prefetcher Techniques

- **Stride Prefetcher:** A conventional stride prefetcher consists
  1. Prefetch table: Store the miss address of a load instruction.
  2. Confidence of prefetching.
  3. Access stride
  4. PC of load instruction to use as the unique identifier.

- **Dynamic Degree Prefetcher:**
  1. Degree: How early the data should be prefetched.
  2. Cyclic code: Difference between the current loop iteration number and the iteration number for which data is being prefetched.

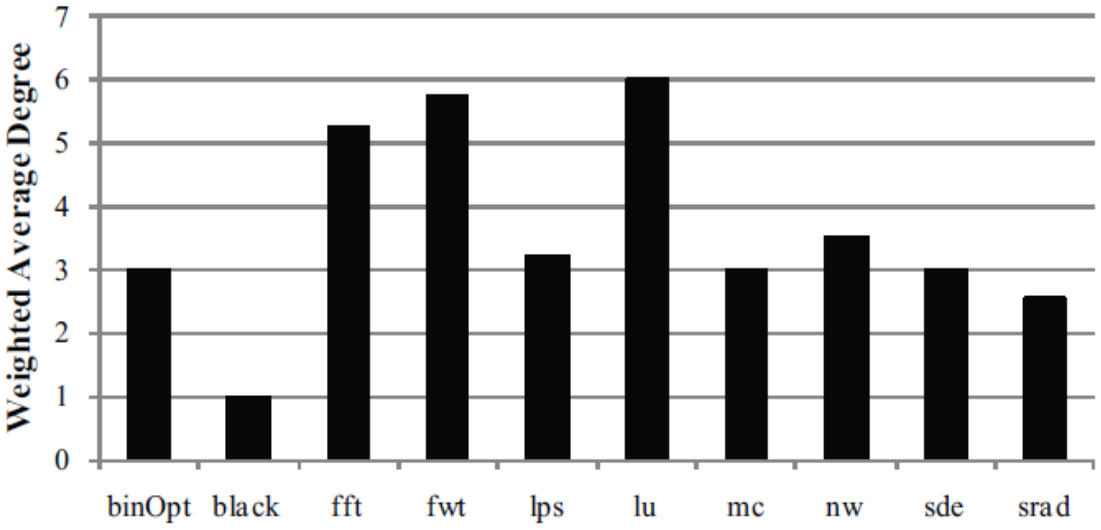

Figure 1.10: Varying weighted degrees for different benchmarks

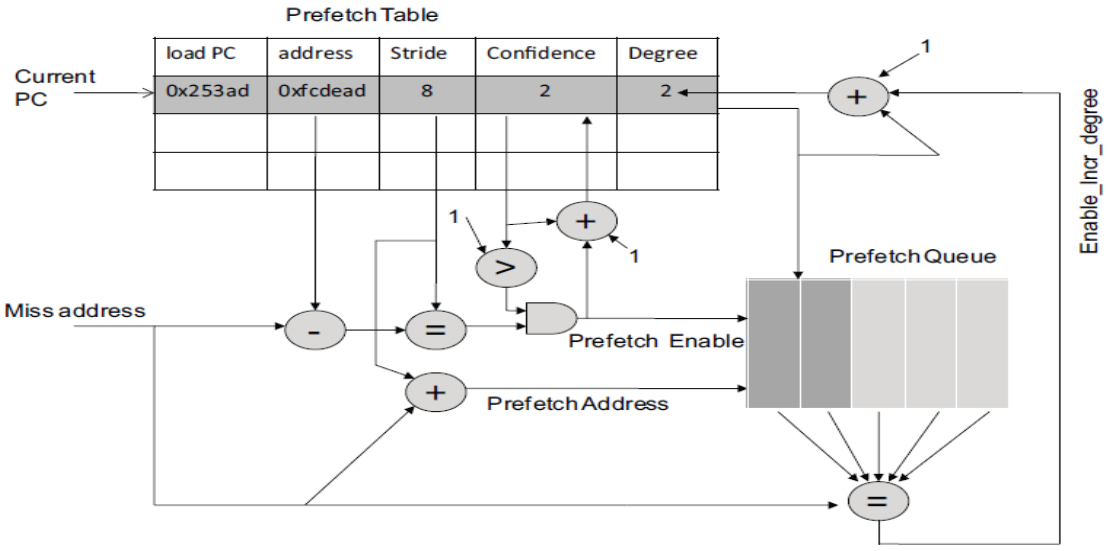

Figure 1.11: Dynamic degree prefetcher

Working:

- The initial degree of prefetching in DDP is set to 1.
- When an address is requested either by a load instruction or by a prefetch request, the prefetch queue is first checked to see if a prefetch request for data at that address has already been made.
- If request is already made , it relies that the prefetching for that load instruction is short-sighted, so the degree for the corresponding PC is incremented.
- By increasing the degree of the prefetcher, data is then prefetched from further ahead and, hence, by the time later iterations are executed, the required data will be present in the cache.
- The degree is not fixed but dependent on the characteristics of the loop itself and different loops settle at different degrees.

- *Lower degree:* Performance penalty as the processor would have to stall for data.
- *Large degree:* More data will be fetched than is necessary, which leads to an over-utilization of bandwidth.
- Higher degrees Assigned to cyclic code with shorter iteration lengths
- Lower degrees Assigned to code with longer iteration lengths.
- The degree computations is done at run-time, based on the miss patterns of the application.

Reducing Control Divergence and Serialization

- SIMD architectures employs "write mask" for updating the values.
- The CUDA environment allows programmer to write scalar codes and have hardware assemble threads into SIMD code.
- *Immediete post-dominator reconvergence:* Leads to reduction of utilization of the datapath.

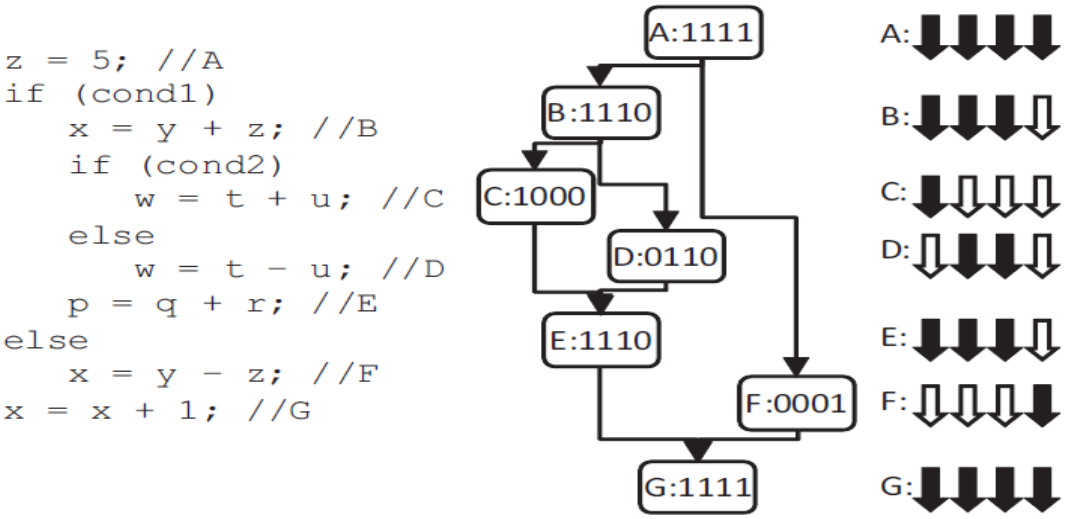

Figure 1.12: Immediete post-dominator reconvergence

- **Divergence Folding:**
  - Complementary predicates executes parallely with some additional modifications.
  - *Modification 1:* Compiler is modified to allow FPU operations guarded by complementary predicates to be executed on the FPU datapath at the same time.
  - *Modification 2:* The control logic is modified to select which of the two concurrently executing subgraphs should write their values to the RF based on a given SIMD lane's predicate mask bit simultaneously.
- **Reduction Tree:**

– *Reduction operations:* Accumulation of previously computed values.

– *Adder tree:* Sum up all the values in logarithmic time.

– Due to FPU chaining, adding some extra interconnect logic can create a FPU tree.

1.2.3 Results

Datapath Optimizations

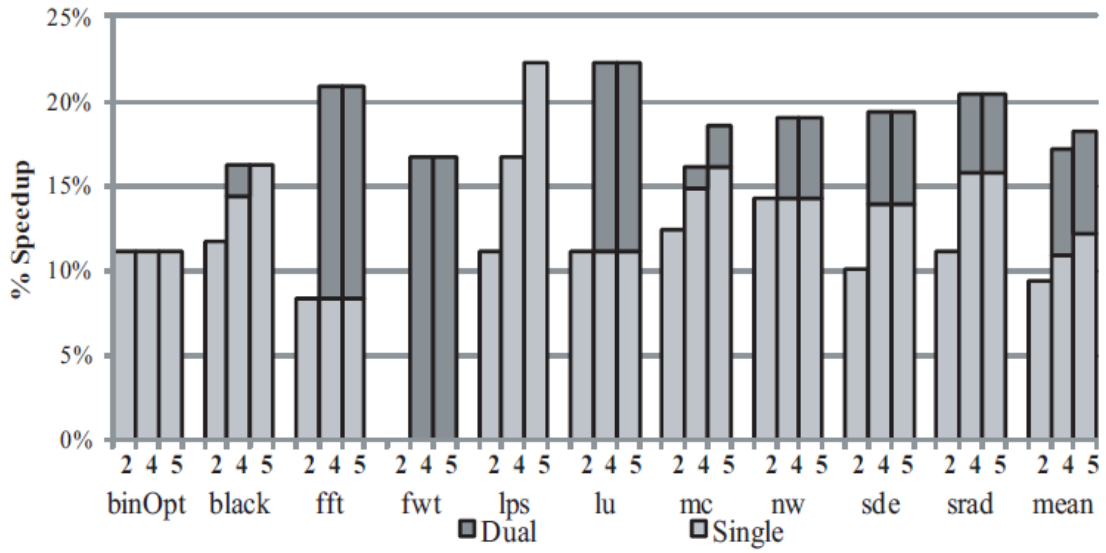

Figure 1.13: Speedup with increasing chain length

- The baseline 3-cycle FPU takes 3 cycles for each operation.
- Latency increases by 3 cycles for every added operation.
- Optimized FPU chain results a savings of 4 cycles when 5 FPUs are connected back-to-back.
- On average, the speedup for a 5-long chain increased from 12% to 18%.
- Speedup saturates at 5 operations and adding a sixth operation in the chain only reduces the overall utilization of the FPU.

Prefetcher Techniques

- The baseline degree-1 provides, on average, a speedup of 2.3X, resultant from a 62.2% reduction of average D-cache miss latency.
- DDP provides a speedup of 3.4X on average, due to an additional 53.1% reduction in average Dcache miss latency over the degree-1 prefetcher. This amounts to a speedup of 1.5X speedup over the traditional degree-1 prefetcher.

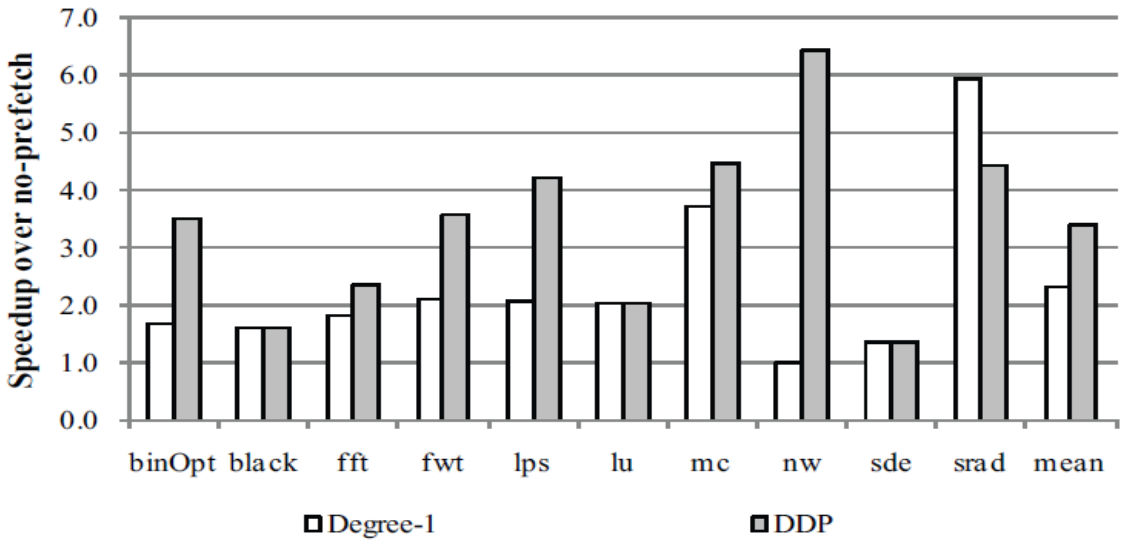

Figure 1.14: Comparison of DDP and degree-1 prefetching

Divergence Folding Technique

- The benchmark lps, lu and srad are control-flow intensive, performing similar, short computation on either side of a divergence.
- Due to divergence folding technique, these benchmarks saw reductions of 38%, 30% and 50% in control-flow overhead, respectively.

Application of PEPSC architecture to GPU

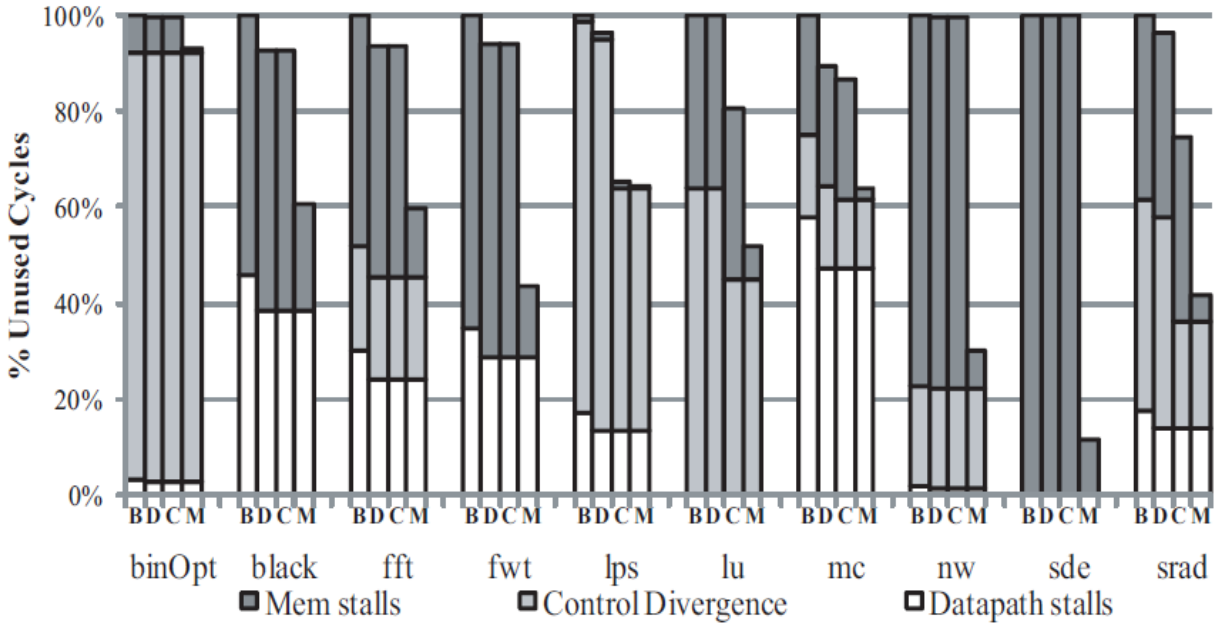

Figure 1.15: Reduction in GPU overheads after cumulatively applying various techniques. "B" is the baseline bar, "D" is after adding the chained FPU datapath, "C" is after adding divergence-folding to mitigate control overhead and "M" is after adding the prefetcher to reduce memory latency.

- Augmenting existing GPUs with the hardware PEPSC introduced here .
- The estimates for the impact this would have are illustrated in Figure 1.15.
- The first bar for each benchmark is the baseline underutilization breakdown from and each subsequent bar is the revised breakdown with cumulatively adding the FPU chaining, divergence folding and the dynamic-degree prefetcher.
- This average utilization increases to around 73% ,around more than 50% improvement over the 45% utilization baseline.

PEPSC Specifications

Frequency	750 MHz
SIMD Lanes	32
Peak Performance	120 GFLOPs
Peak Total Power	2.10W
Efficiency	56.9 Mops/mW
Technology Node	45nm

Component	**Power**
5-op 32-bit FPU with reduction + divergence-folding	1.18W (~37mW/ln)
16-element 32-bit RF	9.28 mW/ln.
Memory system	580mW
Etc. datapath (scalar pipe, AGU, control)	59 mW

Figure 1.16: PEPSC specifications

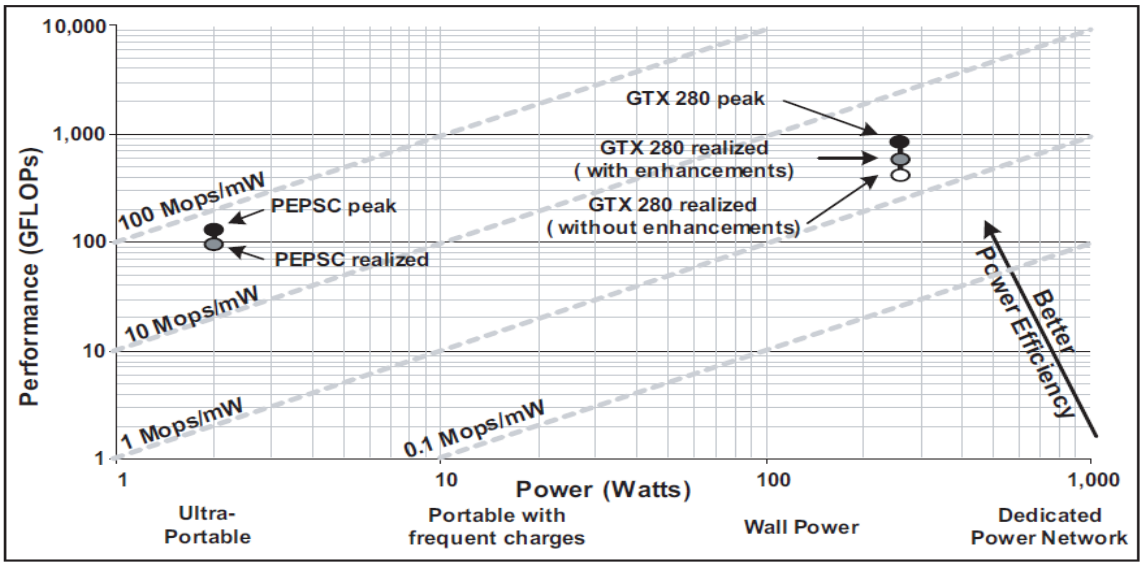

Figure 1.17: Power-efficiency of PEPSC. Black points are peak performances, and the gray and white points represent different points of underutilization.

1.2.4 Conclusions:

PEPSC architecture is

- Power efficient computing architecture.
- Efficient chained-operation datapath.
- Reduces register file accesses.
- Reduces computation latency.
- Significant improvement over 10X over current GPUs.
- Extra hardware components used while in some of the hardware in some benchmarks being underutilized.
- Even register file accesses decreases , port to access those register increases due to chaining
- Compiler is changed to tackle divergence control problem.

Chapter 2

Future Works

2.1 Conference Papers to be Follow Through:

- W. W. L. Fung, I. Sham, G. Yuan and T. M. Aamodt, "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007), 2007, pp. 407-420, doi: 10.1109/MICRO.2007.30.
- Adwait Jog, Onur Kayiran, Nachiappan Chidambaram, ASit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravi R. Iyer and Chita R. Das, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance, ASPLOS 2013
- A. Sethia, D. A. Jamshidi and S. Mahlke, "Mascar: Speeding up GPU warps by reducing memory pitstops," 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), 2015, pp. 174-185, doi: 10.1109/HPCA.2015.7056031.

Chapter 3

References

1. M. A. Ibrahim, O. Kayiran, Y. Eckert, G. H. Loh and A. Jog, "Analyzing and Leveraging Decoupled L1 Caches in GPUs," 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA), 2021, pp. 467-478, doi: 10.1109/HPCA51647.2021.00047.

2. G. Dasika, A. Sethia, T. Mudge and S. Mahlke, "PEPSC: A Power-Efficient Processor for Scientific Computing," 2011 International Conference on Parallel Architectures and Compilation Techniques, 2011, pp. 101-110, doi: 10.1109/PACT.2011.16.