D5.5.3– Design and implementation of the SIMD-MIMD GPU architecture

Document Information

Contract Number	288653
Project Website	lpgpu.org
Contractual Deadline	31-08-2013
Nature	Report
Author	Jan Lucas (TUB), Sohan Lal (TUB), Mauricio Alvarez-Mesa (TUB), Ahmed Alhossini (TUB) and Ben Juurlink (TUB)
Contributors	Paul Keir (Codeplay), Iakovos Stamoulis (ThinkS), Alex J. Champandard (AiGD)
Reviewers	Georgios Keramidas (ThinkS)

Notices:

The research leading to these results has received funding from the European Community's Seventh Framework Programme ([FP7/2007-2013] under grant agreement n° 288653.

Contents

1 Description of Task

Based on the research done in the project this task was slightly changed. We will first provide the original description of the task and explain the changes after this.

1.1 Original Description of the Task

H.264 as well as other applications are difficult to efficiently execute on GPUs because they exhibit divergent branching and memory access patterns while typical GPUs rely on wide SIMD processing to achieve high performance. Similarly, SIMD-based GPUs are not so suitable for many of the task-level parallel jobs currently performed by the SPU on the Cell BE processor, such as game AI, creating lists of triangles to render, doing animation calculations, or doing physics simulation. In this task we will investigate what the most power-efficient way is to execute such algorithms and applications, either by relaxing the requirement that the instruction paths have to be synchronized (as in SIMD), or should a GPU consists of SIMD and MIMD processors? There are intermediate architectural choices between full SIMD and MIMD, such as a technique called Warp Reforming. This task will also involve making power and area estimations of SIMD and MIMD processor cores. This task is a joint effort of TUB, Codeplay, and ThinkS. TUB will focus on a design consisting of a few MIMD cores and many SIMD cores, while Codeplay will focus on Warp Reforming techniques. ThinkS will contribute by providing power and area estimations of the proposed design and by providing feedback concerning the area overheads and the hardware complexity introduced by the proposed approach (combination of SIMD and MIMD processor cores).

1.2 Modifications to the Task

TU-Berlin research done on this task has resulted in a novel technique for efficient execution of applications with divergent branching on GPUs. This technique, however, does not need support from the compiler but is implemented in the hardware. This way it is more flexible than static compiler techniques such as Warp Reforming because it is able to deal with dynamic, divergent, data-dependent branch patterns. Optimizing the execution of applications with these patterns is out of the reach of static compiler-based techniques because the needed information is not available at compile time. In addition to the efficient handling of divergent branches, the proposed architecture also enables efficient hardware support for scalarization. Scalarization is a technique for removing redundant computations and storage from GPU kernels. Scalarization also relies on compiler techniques for identifying registers and instruction that are suitable for scalarization. However, since this technique was not implemented within a conventional compiler but within the GPUPowSim simulator this was developed at TU Berlin. Codeplay provided their expertise on compiler techniques. In addition to the kernels from Task 3.5, we also used kernels provided by AiGameDev and Geomerics.

2

Relevance of the Task to the Project

In this task we designed and implemented a SIMD-MIMD GPU architecture. This task will be further evaluated and optimized in Task 7.3 using all benchmarks developed in WP3. The relevance of this task is indicated by the possibility of developing an IP block if the results are promising. This IP block will be integrated in the final hardware demonstrator. This task provides the input for Task 7.3: Evaluation and Optimization of the SIMD-MIMD Architecture, where the architecture will be further optimized.

2.1 Collaboration

This task is a joint effort of TUB, Codeplay, and ThinkS. We had technical discussions with ThinkS on new proposed architecture called DART. AiGameDev and Geomerics contributed AI and Enlightening benchmarks respectively. We could successfully simulate one benchmark from AiGameDev and used it to evaluate the performance of DART. We only managed to run the Geomerics Enlightening benchmark on a real GPU in the power measurement testbed, but were not able to simulate them to evaluate the performance of DART due to simulation issues related to the CUDA drivers API. We will try to simulate Enlightening benchmark in the third year of the project.

3 Work Performed

3.1 Introduction

During the last years GPUs have evolved from 3D graphic accelerators to devices able to do general purpose computing at very high throughputs at a high energy and area efficiency. GPUs capable of general purpose GPU computing (GPGPUs) are now becoming common even in SoCs intended for mobile devices. Many factors influence the power and performance of applications running on GPUs. For this reason we characterized different GPU workload to gain further insights into their behavior. We also analyzed the relationship between power and the characteristics of the workloads. We used characterization results to provide additional guidance to our work on the SIMD-MIMD architecture.

But two problems were clear from the beginning: The problem of branch divergence and redundancy. We will now first provide an introduction into these topics and then explain the work done on workload characterization.

3.1.1 Branch Divergence

GPUs are programmed using a single program multiple data (SPMD) programming model. While this programming model gives the programmer the illusion of thousands of independent threads, the GPU internally groups together threads into groups (called warps in NVIDIA termionology) and executes them on a SIMD unit. But at the same time this imposes the requirement of running all threads within one warp in lockstep and executing all branch directions and masking out all threads that do not follow the currently executing direction. During the execution the masked out functional units (FUs) can not be used.

The grouping of threads into warps allows the GPU to fetch instructions once per warp instead of once per thread. This SIMD execution provides high efficiency and high peak performance to GPUs but at the same time it also makes execution of code with divergent branches less efficient. Efficient execution of branch divergent code in GPUs has been an active research area [2] [9]. However these techniques only work well with specific branch patterns. We propose to deal with this problem using Temporal SIMD (TSIMD). TSIMD is a new and potentially more general technique for more efficient execution of SPMD code. While it has already been mentioned in some publications, the performance characteristics and architectural changes required for TSIMD have not been analysed in detail. This report describes DART, a GPU architecture build around TSIMD execution units and enables a look at the performance characteristics of such an architecture. DART stands for Decoupled ARchitecture for TSIMT. In DART the execution units and register files together with some additional logic form lanes that execute the instructions of the threads in a self-managed way, decoupling the execution of instructions from the fetching and decoding. With this decoupling it

becomes possible to adjust the performance of the fetch and decode frontend and the execution lanes to the application and the level of divergence present in the application.

3.1.2 Redundancy

Kernels for GPUs are written using a single program multiple data (SPMD) programming paradigm. To reach their high efficiency they use thousands of parallel threads instead of the usual hand full of threads used in multi- and manycore parallel programs. To reach this level of thread level parallelism (TLP) algorithms running on the GPU must be split into many small parts that can execute mostly without requiring communication or synchronization between the threads.

For this reason values that are equal in all threads are often calculated separately in every single thread. Exchanging the values between the threads would be much more expensive, both in time as in energy, than recalculating them. An energy efficient GPU architecture should be able to reduce the number of redundant instructions and provide an efficient way of reusing calculations in different threads.

However providing a way of exchanging data between the threads and reusing calculations is not enough. It is hard and time consuming for a programmer to correctly identify the opportunities for a reuse of calculations in a different thread. For this reason this step must be done automatically by the compiler. This way all existing code can be reused.

On a GPU the warps are executed on single instruction multiple data units. These instructions get vectors of data as inputs and produce vectors of data as a output. When all used elements of these vectors are identical, we can reduce this operation to single scalar operation. We call the process of changing suitable instructions from vector to scalar operations Scalarization. The proposed architecture provides hardware support for scalarization and is thus able to remove this redundancy from the SPMD programs.

3.2 Workload Characteristics

While general purpose computing on GPUs continues to enjoy higher computing performance with every new generation, the high power consumption of GPUs is an increasingly important concern. To meet future GPGPU computing performance and power requirements, it is important to develop new architectural and applications techniques to create power efficient GPUs. To identify such techniques, it is necessary to study power, not only at the level of the entire GPU chip or an entire application, but to investigate the power consumption for individual architectural components and application kernels. Therefore, in this section we study GPGPU workload characteristics and its correlation with GPU component power consumption.

3.2.1 GPGPU Workload Metrics

We use several simple and well known workload metrics such as IPC (instructions per cycle), SIMD Utilization to understand the performance characteristics of various workloads. Below is the descrip-

tion of metrics that we consider for our work. Each metric is calculated per kernel:

1. IPC: IPC is instructions per cycle.
2. Total Ins (TI): Total number of executed instructions.
3. Arithmetic Ins (AI): It is the ratio of arithmetic instructions to total instructions. Arithmetic instructions is the sum of integer, floating point and special function unit (SFU) instructions.
4. Branch Ins (BI): It is the ratio of branch instructions to total instructions. It is important to note that a branch instruction does not necessarily mean that it will cause branch.
5. Memory Ins (MI): It is the ratio of memory instructions to total instructions.
6. Coalescing Efficiency (CE): It is the ratio of number of global memory instructions and global memory transactions. The metric capture the coalescing behavior of kernels.
7. SIMD Utilization (SU): The average utilization of SIMT core for issued cycles.

$$\text{SIMD Utilization} = \sum_{i=1}^{n} W_i * i / \sum_{i=1}^{n} W_i \quad (1)$$

   where, n is the warp size,
   W_i = Number of cycles when a warp with *i* active threads is scheduled into the pipeline.

8. Pipeline Stalled (PS): It is the fraction of total cycles for which pipeline is stalled and could not do useful work.
9. Active Warps (AW): Number of active warps per Streaming Multiprocessor(SM).

3.2.2 Performance Characteristics

We use GPUSimPow developed in Task 4.2 to simulate different benchmarks. We configure GPUSimPow to simulate architecture similar to NVIDIA GTX580. Theoretical peak IPC and SIMD utilization of simulated GTX 580 are 1024 and 32 respectively. The benchmarks selection include benchmarks from popular Rodinia benchmark suite [**?**], CUDA SDK [**?**]. We also include some benchmarks developed in Task 3.1 and Task 3.5.

Table 1 shows metrics values for different kernels. Each row contains a kernel name, values of different metrics mentioned in Section 3.2.1 including execution time. These metrics help to understand the performance characteristics of kernels. For example, SIMD utilization (SU) metric shows the average utilization of execution units by the kernel. In this case, a SIMD utilization of 32 means 100% utilization of the execution units and a value lower than 32 means there are some execution units which are left unused. Basically, current generation of GPUs does grouping of independent scalar threads and execute these group of threads in lockstep on the underlying SIMD hardware. This technique is used to increase the programmability and hardware efficiency of the GPUs. It works well for highly parallel workloads but if the workload has control divergence the SIMD hardware could be underutilized. Table 1 shows that there are number of benchmarks which do not utilize the execution units fully. To increase the utilization of execution units for control divergent workloads, we propose a new micro-architecture in Section 3.3 which can handle the control divergence efficiently than the current generation of GPUs.

Kernel	IPC	TI (m)	AI	BI	MI	CE	SU	PS	AW	ET (ms)
binomialOptions	823	13474.8	0.53	0.16	0.31	0.65	31.99	0.12	19.35	19.91
blackScholes	387	356.9	0.74	0.01	0.08	1.00	32.00	0.62	14.93	1.12
convolutionSep1	339	496.1	0.39	0.01	0.44	0.50	32.00	0.67	7.98	1.78
convolutionSep2	339	476.2	0.48	0.01	0.46	0.50	32.00	0.67	15.82	1.70
fastWalshTrans1	98	101.1	0.58	0.08	0.33	1.00	32.00	0.90	18.10	1.25
fastWalshTrans2	902	710.9	0.74	0.06	0.20	1.00	32.00	0.09	23.63	0.96
fastWalshTrans3	264	114.2	0.68	0.02	0.20	1.00	32.00	0.74	20.88	0.53
matrixMul	435	4.7	0.44	0.04	0.51	0.50	32.00	0.50	12.48	0.01
mergeSort1	784	4418.7	0.83	0.07	0.10	1.00	32.00	0.21	23.91	6.85
mergeSort2	19	8.8	0.82	0.06	0.12	0.07	32.00	0.98	13.11	0.56
mergeSort3	206	6.0	0.75	0.06	0.08	0.69	32.00	0.39	15.11	0.04
mergeSort4	448	710.5	0.71	0.07	0.22	0.50	26.46	0.45	15.36	1.93
MonteCarlo1	502	8.5	0.84	0.09	0.07	1.00	32.00	0.46	15.20	0.02
MonteCarlo2	795	1212.1	0.83	0.06	0.11	1.00	32.00	0.16	22.62	1.85
scalarProd	182	22.1	0.70	0.14	0.15	1.00	31.22	0.74	23.31	0.15
SimScore1	149	0.2	0.71	0.08	0.19	0.12	23.68	0.74	5.04	0.002
SimScore2	127	0.7	0.71	0.19	0.10	0.97	29.76	0.76	3.95	0.01
SimScore3	13	0.1	0.70	0.20	0.10	0.93	29.73	0.76	3.89	0.01
vectorAdd	171	11.0	0.59	0.09	0.32	1.00	32.00	0.83	20.70	0.08
backprop1	606	117.7	0.74	0.15	0.11	0.35	24.91	0.22	23.60	0.24
backprop2	517	72.4	0.75	0.04	0.20	0.64	32.00	0.49	19.51	0.17
bfs1	21	23.3	0.62	0.11	0.26	0.49	9.69	0.92	15.21	1.30
bfs2	153	16.5	0.57	0.12	0.22	0.73	26.57	0.65	19.73	0.13
heartwall	407	7047.3	0.67	0.11	0.22	0.48	25.47	0.38	15.49	21.04
hotspot	493	110.1	0.82	0.04	0.14	0.33	28.02	0.40	11.62	0.27
kmeans1	468	1570.6	0.78	0.06	0.11	1.00	32.00	0.54	22.98	4.08
kmeans2	11	130.9	0.59	0.13	0.28	0.06	32.00	0.99	16.32	14.47
lavaMD	142	21056.4	0.77	0.03	0.20	0.13	25.02	0.82	11.91	179.49
leukocyte1	695	5055.8	0.83	0.02	0.11	0.48	29.16	0.17	17.08	8.85
leukocyte2	237	12638.5	0.95	0.01	0.04	0.51	31.91	0.69	5.00	64.75
leukocyte3	720	2810.0	0.82	0.04	0.12	0.03	28.56	0.08	23.31	4.75
pathfinder	822	129.8	0.79	0.06	0.16	0.48	30.58	0.08	23.30	0.19
H264 IDCT 4x4	503	5.6	0.77	0.06	0.18	0.75	32.00	0.44	14.23	0.01
H264 IDCT 8x8	532	6.5	0.83	0.02	0.15	0.50	32.00	0.43	14.23	0.01
viscache_borders	26	69.4	0.77	0.09	0.15	0.05	2.83	0.54	3.91	3.24
H264 MC Chroma	447	136.6	0.76	0.04	0.20	0.88	31.81	0.55	7.83	0.37
H264 MC Luma	366	405.3	0.71	0.04	0.24	0.49	31.27	0.62	7.96	1.35

Table 1: Metrics values for GTX580

Motion Compensation Kernel Analysis In this section we analyze the motion compensation kernel of H.264 in detail. The motion compensation kernel was developed in Task 3.5. The motion compensation has two sub kernels namely Chroma and Luma. Table 2 shows different synthetic modes that we analyze. We synthesize videos with block sizes of 16x16, 8x8, and 4x4 with either branch divergence or memory divergence or both enabled. We also compared synthetic modes with the real modes.

Execution Mode	Branch Divergence	Memory Divergence
16x16BrchNoMemNo	n	n
8x8BrchNoMemYes	n	y
8x8BrchYesMemNo	y	n
8x8BrchYesMemYes	y	y
4x4BrchNoMemYes	n	y
4x4BrchYesMemNo	y	n
4x4BrchYesMemYes	y	y
AllPFrameRef1	y	y
AllBFrameRef16	y	y

Table 2: Simulated modes of motion compensation kernel for GTX580

IPC and SIMD Width Utilization Table 3 shows IPC and SIMD width utilization of different execution modes for both Chroma and Luma kernels. The table shows that SIMD utilization is less than 32 for some modes especially for Luma kernel. For example, the execution mode 4x4 block with both branch and memory divergence enabled ($4x4BrchYesMemYes$) has only 22.6 SIMD utilization. The low SIMD utilization is due to the branch divergence present in those kernels. In Section 3.3 we propose a new micro-architecture which can handle branch divergence in a better way than conventional GPUs. The average IPC value of Chroma and Luma kernels is only 44.2% and 40.5% of peak IPC. We do resource usage analysis in paragraph 2 of the Section 3.2.2 to find the reasons for low IPC.

Mode	Chroma		Luma	
	IPC	SIMD Utilization	IPC	SIMD Utilization
16x16BrchNoMemNo	455	32.0	469	32.0
8x8BrchNoMemYes	455	32.0	469	32.0
8x8BrchYesMemNo	454	32.0	418	27.1
8x8BrchYesMemYes	455	32.0	418	27.3
4x4BrchNoMemYes	453	32.0	462	32.0
4x4BrchYesMemNo	455	32.0	358	22.7
4x4BrchYesMemYes	455	32.0	337	22.6
AllPFrameRef1	448	32.0	430	31.0
AllBFrameRef16	447	31.8	366	31.3

Table 3: IPC and SIMD utilization of different modes for GTX580

Resource Usage Chroma and Luma Kernels on GTX580 Typically, GPUs require lot of parallel threads to exploit thread level parallelism to hide long latency memory operations. However, actual number of threads executing in parallel is limited by the amount of resource usage such as number of registers, shared memory, maximum limit on Cooperative Thread Array (CTA) etc. Table 4 shows the maximum value of different resources on a single Streaming Multiprocessor (SM) of GTX580 and their usage by chroma and luma kernels. The table shows that each SM only has 512 threads (33%) and this much number of threads might not be sufficient to hide the long latency of memory operations. The number of threads in chroma is limited by the maximum limit on CTA and in luma it is limited by both CTA and shared memory. In paragraph 3 of the Section 3.2.2 we increase the maximum limit of CTA as well as shared memory and present their results.

Resource	Max. Value	Chroma	Luma	Utilization(%)
CTA	8	8	8	100
Threads/CTA	1024	64	64	6.25
Max threads	1536	512	512	33
Shared memory/CTA	48K	0	48K	0/100
Registers/thread	63	20	20	33
IPC	1024	452.8	414.2	44.2/ 40.5
Pipeline stalled	-	54.5%	51.6%	–

Table 4: Resource usage per SM on GTX580

Architectural Enhancements Table 5 shows four new configurations for GTX580. In the four new configurations we increase the maximum limit on CTA and shared memory. The others parameters are similar to GTX580.

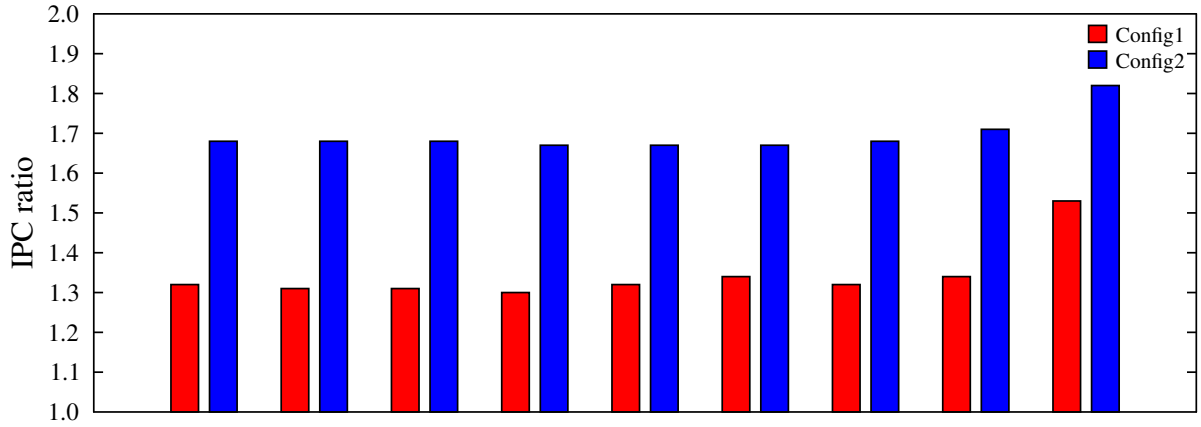
Configuration	Max CTA	Shared Memory
Base	8	48K
Config1	16	48K
Config2	24	48K
Config3	16	96K
Config4	24	144K

Table 5: Configurations

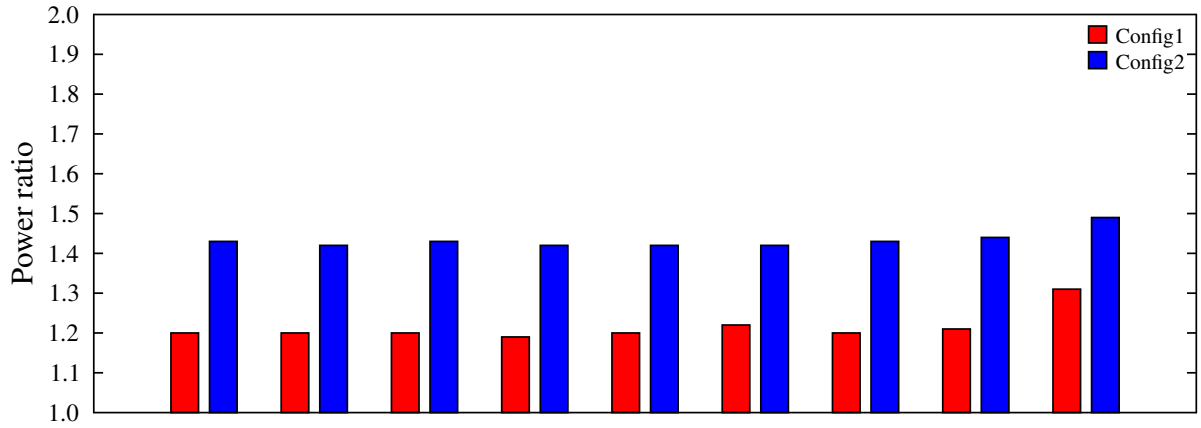
In config1 and config2 we only increase the maximum number of CTA per SM and keep all other parameters same as in base GTX580 configuration. Since chroma kernel is only limited by CTA limit, we expect chroma to gain from this change. However, the luma does not benefit from config1 and config2 because it is limited by both CTA and shared memory. Figure 1 shows the normalized IPC, power, and energy of chroma for config1 and config2. The IPC, power, and energy ratios are normalized to base configuration.

In config3 and config4 we increase both the maximum number of CTA and shared memory per SM. Since luma is limited by both CTA limit and shared memory, we now expect both chroma and luma to gain from this change. However, since the chroma does not use shared memory, the change in chroma performance is same as in config1 and config2. Figure 2 shows the normalized IPC, power, and energy of luma for config3 and config4.

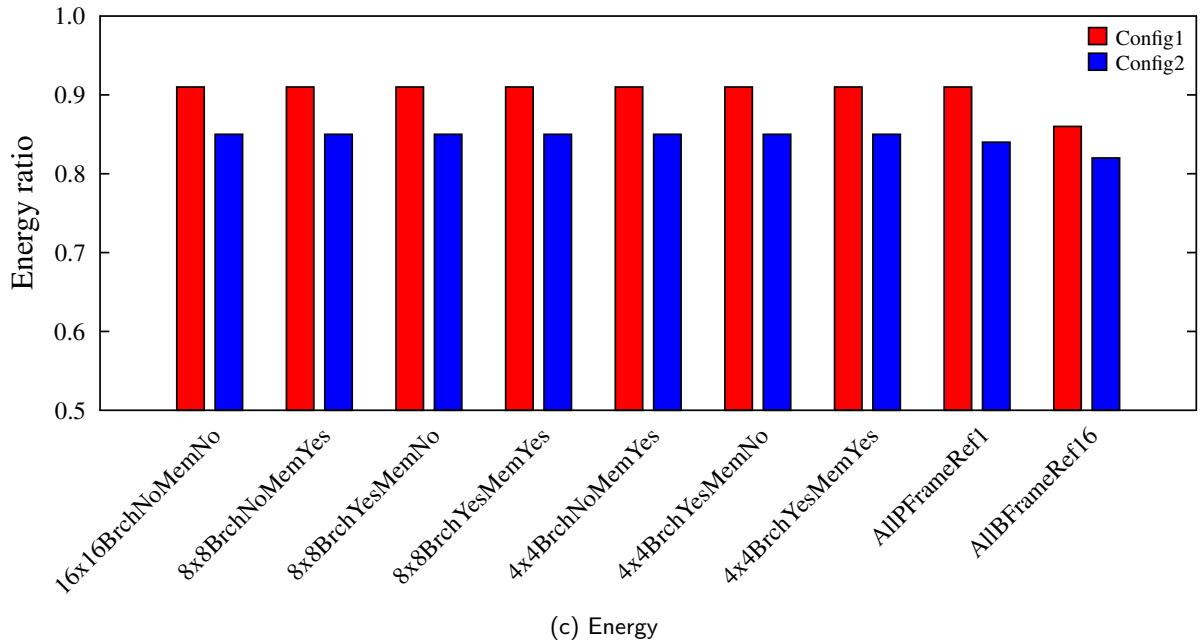
Figure 1a and Figure 2a show that IPC increases on an average by 35% and 34% compared to base for config1 and config3 for chroma and luma respectively. The IPC further increases on an average by 70% for config2 and 45% for config4. Figure 1b show that power consumption also increases on an average by 22% for config1 and 43% for config2 for chroma. Figure 2b show 19% and 26% average increase in power consumption for config3 and config4 for luma. However, Figure 1c and Figure 2c show that we can save on an average 10% energy for config1 and 16% energy for config2 for chroma. For luma we save 11% energy for config3 and 13% energy for config4. The one limitation of our work is that the GPU power model currently does not include power overhead of increasing CTA limit. The hardware support for increasing the CTA limit will require more space for storing CTA ids, shared memory offsets and barriers. We think that this overhead is negligible. However, more work is required to exactly quantify these overheads.



(a) IPC

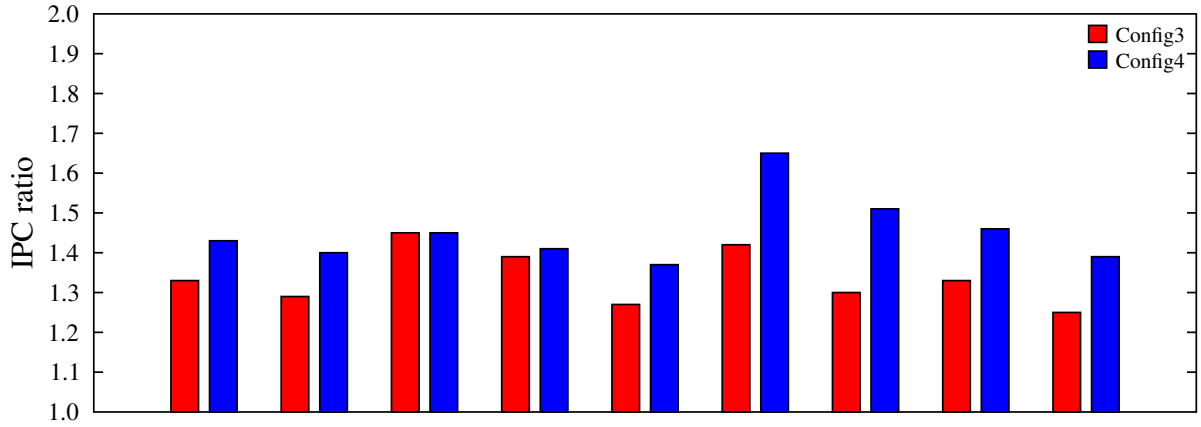


(b) Power

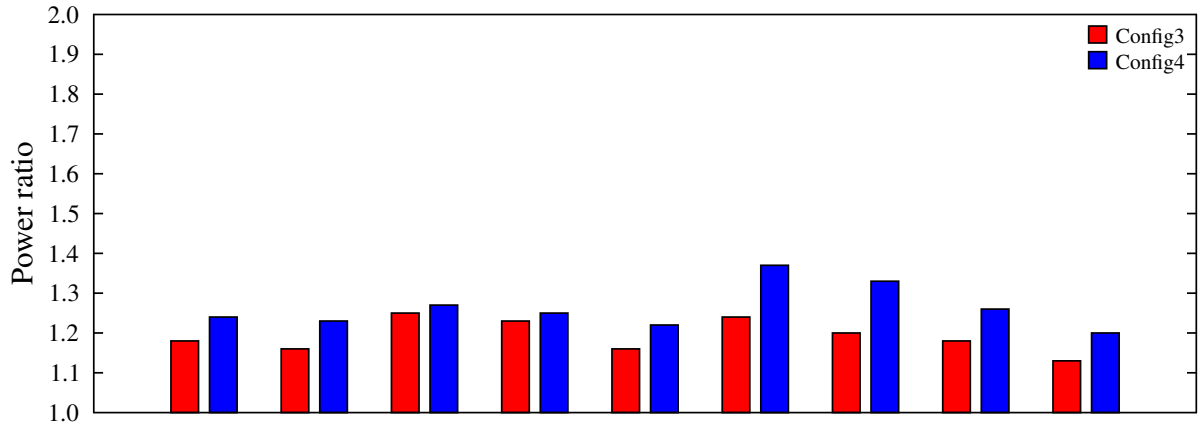


(c) Energy

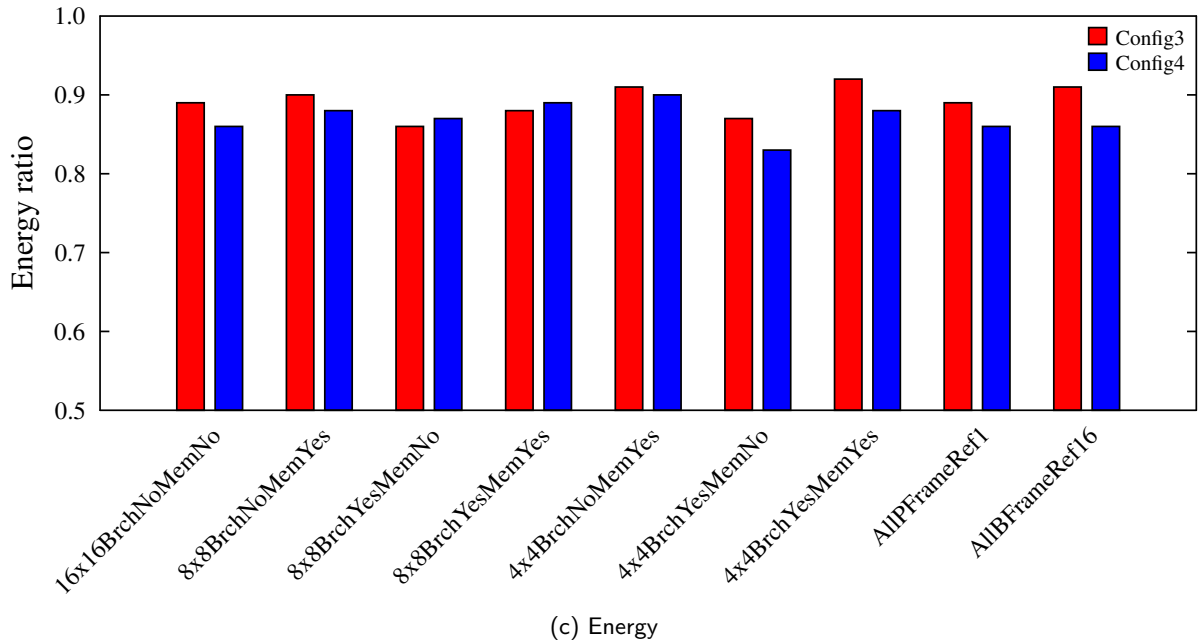
Figure 1: Normalized IPC, power and energy of chroma for config1 and config2



(a) IPC



(b) Power



(c) Energy

Figure 2: Normalized IPC, power and energy of luma for config3 and config4

3.2.3 GPU Components Evaluated for Power Consumption

Before performing any power optimizations, it is necessary to accurately estimate the power consumption of various components and the change in component power with the change in workload characteristics. This is a necessary step to identify the most power consuming components for different workload characteristics. Table 6 shows a description of GPU components evaluated for power consumption and correlation.

Component	Description
Execution units (EU)	Integer, floating point, and special functional units
Register file (RF)	SRAM single ported banks
Warp control unit (WCU)	Warp status table, re-convergence stack, scoreboard, instruction buffer and decoder
Load store unit (LDSTU)	Coalescer, Bank conflict checker, Shared memory, L2 cache, Constant cache,Texture cache
Base power (BP)	Streaming Multiprocessor activation power
Clusters power (CP)	Cluster activation power
Network on chip (NOC)	-
Memory controller (MC)	-
Global Memory (GM)	Either DDR3 SDRAM or GDDR5 SGRAM
Total Power (TP)	Power consumed by all GPU components

Table 6: Summary of evaluated GPU components for power consumption.

3.2.4 GPU Component Power Consumption

Table 7 shows the dynamic power consumption by individual components on GTX580 GPU for different kernels. The table shows a significant change in component power consumption across different workloads. It would be interesting to study the workload characteristics that cause this change in component power consumption and classify the kernels into different categories depending on their power characteristics. The kernels with similar power characteristics can be grouped into one category. This could be a useful information for architects and programmers for power optimizations as most power consuming components change from one category of workload to another and they can choose the components depending upon workload. To achieve this, we study correlation between workload metrics and GPU components power consumption in Section 3.2.5.

3.2.5 Correlation between Workload Metrics and GPU Component Power Consumption

We calculate the Pearson correlation coefficient between the workload metrics and GPU components. It is a measure of the linear correlation (dependence) between two variables. Table 8 shows the correlation between the workload metrics described in Section 3.2.1 and GPU components described in Section 3.2.3. The correlation coefficient between workload metrics and component power vary between -1 and 1. Higher absolute value of correlation coefficient means strong correlation exist between the metric and the corresponding component. The negative value means there is an inverse correlation between the metric and the component.

Benchmark Name	RF	EU	WCU	BP	LDSTU	CP	NOC	MC	GM	TP
binomialOptions	12.9	18.6	15.4	3.6	1.7	12.6	1.0	2.8	5.9	74.5
blackScholes	6.7	16.2	9.7	3.5	1.6	12.1	4.2	11.0	21.5	86.4
convolutionSep1	6.2	8.1	6.6	3.6	1.8	12.3	4.4	7.8	15.1	65.8
convolutionSep2	6.6	8.7	8.1	3.8	1.9	13.2	5.5	8.2	15.3	71.4
fastWalshTrans1	1.6	2.3	6.5	3.8	1.6	13.1	4.4	12.2	21.9	67.4
fastWalshTrans2	15.3	24.8	17.1	3.9	2.2	13.5	3.8	7.4	13.8	101.9
fastWalshTrans3	4.2	7.2	8.9	3.7	2.7	12.8	7.3	11.7	21.7	80.2
matrixMul	8.1	9.0	9.0	3.3	2.1	11.5	3.1	2.9	6.0	55.1
mergeSort1	13.7	20.2	15.2	3.9	0.6	13.5	0.5	3.9	7.6	79.2
mergeSort2	0.5	0.5	4.0	3.8	0.3	13.2	3.7	11.7	21.0	58.7
mergeSort3	9.2	14.2	10.9	3.5	2.4	12.0	2.4	4.9	9.5	69.0
mergeSort4	8.4	10.0	12.9	3.9	1.2	13.6	3.3	7.0	12.8	73.0
MonteCarlo1	8.2	22.4	10.9	3.7	1.0	12.8	2.8	8.5	15.4	85.6
MonteCarlo2	14.1	30.3	15.7	3.7	2.7	12.6	1.4	2.9	6.0	89.5
scalarProd	2.9	4.5	6.4	2.8	1.2	9.8	3.1	9.5	17.3	57.5
SimScore1	3.7	4.2	6.3	3.1	1.2	10.8	6.6	2.8	5.8	44.5
SimScore2	2.0	2.9	2.8	2.3	0.2	7.8	0.4	3.2	6.5	28.1
SimScore3	0.2	0.3	0.3	0.2	0.0	0.9	0.0	2.8	5.9	10.7
vectorAdd	2.2	3.5	6.6	3.9	1.5	13.5	4.2	9.8	17.7	63.0
backprop1	11.5	16.6	18.7	3.9	1.1	13.6	4.2	5.4	10.2	85.2
backprop2	7.6	16.9	12.1	3.9	4.4	13.5	9.2	9.8	17.8	95.4
bfs1	4.4	6.2	9.7	3.8	1.5	13.2	3.7	8.6	15.7	66.7
bfs2	5.6	9.3	11.4	3.6	2.2	12.6	3.6	6.1	11.8	66.1
heartwall	7.7	9.8	12.3	3.2	3.1	11.2	3.6	5.5	10.5	67.0
hotspot	9.1	14.5	16.7	3.9	0.6	13.5	2.9	3.9	7.7	72.9
kmeans1	8.6	13.7	12.3	3.9	0.3	13.4	8.2	12.7	23.0	96.1
kmeans2	0.5	0.2	5.0	3.9	0.6	13.5	8.5	12.4	22.2	66.8
lavaMD	3.5	5.7	5.9	3.9	1.4	13.4	14.9	2.8	5.8	57.4
leukocyte1	14.5	23.1	16.2	3.5	0.3	12.1	8.4	2.8	5.8	86.9
leukocyte2	4.1	12.0	4.7	3.0	0.1	10.3	0.3	2.8	5.8	43.0
leukocyte3	13.4	21.5	18.9	3.6	0.1	12.3	3.6	2.8	5.9	82.0
pathfinder	14.0	20.5	18.2	3.8	1.5	13.1	3.2	6.0	11.3	91.6
H264 IDCT 4x4	7.7	11.5	9.8	3.5	1.5	12.0	4.1	6.7	12.5	69.2
H264 IDCT 8x8	9.2	13.6	10.6	3.6	1.4	12.4	4.2	6.4	11.9	73.4
viscache_borders	4.5	0.7	8.9	2.6	0.4	9.1	4.7	6.2	11.7	48.7
H264 MC Chroma	6.6	10.9	8.9	3.9	3.0	13.5	3.9	3.0	6.2	59.7
H264 MC Luma	6.1	8.5	7.8	3.9	1.3	13.5	3.1	3.3	6.6	54.2

Table 7: Component power (Watts) consumed by NVIDIA GTX580

We summarize our observations of correlation between workload metrics and individual component power as follows:

- IPC: IPC has strong correlation with Register file (RF), Execution units (EU), Warp control unit (WCU), and Total power (TP).
- Total Ins (TI): Total number of instructions executed do not correlate well to any component.

	RF	EU	WCU	BP	LDSTU	CP	NOC	MC	GM	TP
IPC	0.90	0.89	0.89	0.41	0.11	0.41	-0.15	-0.19	-0.19	0.71
TI	0.16	0.12	0.08	0.06	0.06	0.06	0.25	-0.21	-0.21	0.06
AI	0.37	0.45	0.28	0.11	-0.04	0.11	-0.15	-0.04	-0.07	0.29
BI	-0.20	-0.24	-0.17	-0.46	-0.20	-0.46	-0.29	-0.08	-0.09	-0.34
MI	-0.12	-0.25	-0.10	0.17	0.17	0.17	0.02	0.12	0.12	-0.05
CE	0.22	0.31	0.13	0.08	0.34	0.08	-0.18	0.27	0.28	0.30
SU	0.21	0.32	0.22	0.41	0.23	0.41	0.16	0.15	0.15	0.38
PS	-0.94	-0.86	-0.91	-0.28	-0.09	-0.28	0.23	0.26	0.26	-0.65
AW	0.46	0.48	0.67	0.53	0.24	0.53	0.21	0.30	0.30	0.68

Table 8: Pearson correlation between workload metrics and GPU component power consumption for NVIDIA GTX580

- Arithmetic Ins (AI): Arithmetic Ins does not have strong correlation with any of the components, but shows some expected trends. For example, Arithmetic Ins has positive correlation with RF, EU, WCU, but it has negative correlation with NOC, Memory controller (MC), and Global memory (GM).

- Branch Ins (BI): Interestingly, Branch Ins has negative correlation with all the components.

- Memory Ins (MI): Memory Ins has negative correlation with RF, EU, WCU and TP.

- Coalescing efficiency (CE): Coalescing efficiency has positive correlation to all components except NOC. For benchmarks with higher coalescing efficiency, relatively less number of memory transactions are done and thus, lower number of packets are injected into NOC.

- SIMD Utilization (SU): This metric has positive correlation with all components.

- Pipeline Stalled (PS): Table 8 shows that the Pipeline stalled metric is almost inverse of IPC for all components. This mean we can choose just one of these two metrics.

- Active Warps (AW): Active warps per SM has positive correlation with all components. The number of active warps shows occupancy of the SM. Active warps correlates very well to RF, EU, WCU, BP, CP and TP.

3.2.6 Categorization of Workloads

In Section 3.2.5, we found that IPC has strongest correlation, as compared to other metrics, with GPU components power consumption. Thus, we choose IPC to further explore the power characteristics of various workloads. We classify the kernels into *high IPC*, *medium IPC*, and *low IPC* category. Peak IPC is the maximum number of instructions a GPU can execute per cycle. The *high IPC* category contains all kernels whose IPC is equal to or higher than 60% of peak IPC. The *medium IPC* category contains all kernels whose IPC is equal to or greater than 40% and less than 60% of peak IPC. The kernels with IPC less than 40% belong to third category of *low IPC*. Blem et al. [1] defines benchmarks with IPC less than 40% of peak IPC as challenging benchmarks as these benchmarks under-utilize the GPU computing resources. We simulated 83 kernels in total. Table 9 shows number of kernels and average IPC for each category of kernels. The table shows that relatively

Category	Peak	High		Medium		Low	
	IPC	#Kernels	Avg IPC	#Kernels	Avg IPC	#Kernels	Avg IPC
GTX580	1024	18	765.7	15	495.6	50	172.7

Table 9: Number of kernels and average IPC for each category of kernels for GTX580.

large number of kernels are in the low IPC category. The reason for large number of kernels in low IPC category includes low SIMD Utilization, high Pipeline Stalled fraction as shown in Table 1.

We also studied the power distribution for different category of kernels. Table 10 shows the power distribution for three categories of kernels. We see a significant change in components power consumption across the three categories of kernels. The Execution units (EU), Warp control unit (WCU), and Register file (RF) are the three most power consuming components for kernels belonging to *high IPC* category. For the medium IPC category kernels, the components EU, WCU, and RF consume relatively less power as compared to *high IPC*, but still consume significant amount of power. Global memory(GM) power consumption increases from 9.9% to 15.9% for *medium IPC* category of kernels. It is interesting to note that power consumed by RF, WCU, and EU is far less for *low IPC* category kernels as compared to other two categories. The largest fraction of power is consumed by global memory in the *low IPC* category. The Cluster power (CP) and the Base power (BP) consumption increase from *high IPC* to *low IPC* because the overall usage of cores decreases, but activation power is still consumed. In short, we see that most power consuming components change across the three categories of workloads. This could be a useful information for targeting power optimizations at the component level for different categories of workloads as the most power consuming components change with the change in workload characteristics. We observed similar trends for GT240 and we did not include these results for reasons of space.

	RF	EU	WCU	BP	LDSTU	CP	NOC	MC	GM
High	14.1%	26.7%	19.3%	4.8%	1.0%	16.5%	2.8%	5.1%	9.9%
Medium	11.4%	17.5%	16.4%	5.1%	2.0%	17.6%	5.8%	8.5%	15.9%
Low	6.1%	9.5%	11.7%	6.2%	2.1%	21.5%	7.5%	12.2%	23.2%

Table 10: Component power consumption for high, medium, and low IPC category of kernels for GTX580.

3.2.7 Summary

We extracted performance metrics for different workloads using GPUSimPow and studied the power consumption of workloads at the component level. We analyzed the performance bottlenecks of motion compensation kernel and evaluated simple architectural enhancements that show significant increase in performance and decrease in energy. Then we studied the correlation between workload metrics and component power consumption which is useful to understand the power consumption of different workloads. We further classified kernels into high, medium, and low IPC category to study the power consumption behavior of each category. The results show a significant change in components power consumption across the three categories of kernels. This could be a vital information for the computer architects and application programmers to prioritize the components for power and performance optimizations.

3.3 Proposed Architecture

In this section we will explain the proposed new architecture. Based on our findings from the characterization of GPU workloads we propose a GPU architecture that is able to deal with one of the main reasons for low performance on a GPU: branch divergence. We also found that register file use a significant amount of energy in GPUs. We also found that the SPMD programming model creates a lot of redundant operations. We will first explain how the proposed architecture enables efficient execution of branch divergent code and how it can remove some redundancy by scalarization. After that we will provide an overview over the new architecture.

3.3.1 Architectural Support for Efficient Execution of Branch-Divergent Code

When programs branch and not all threads of a warp are following the same control flow path this is called branch-divergence. The GPU executes both directions of the branch after each other while disabling all functional units that are following a different path than the currently executed path. The disabled units can not be used and still consume static power. Also the instruction words needs to be send to all functional units, even those that are disabled. To allow a efficient execution of branch divergent code, we must have an architecture that is able to reduce idle functional units, even during the execution of branch-divergent code.

We present here an architecture based on the temporal execution of SIMD instructions. With the temporal execution of SIMD instructions, non-active threads can be skipped. The equivalent of the skipping of non-active threads in the spatial SIMD unit would be the removal of functional units, which can not be done in a flexible and dynamic way. But with temporal SIMD we are able to dynamically skip over unused slots and not waste time and energy on them.

3.3.2 Architectural Support for Scalarization

To enable efficient scalarization vector operations need to have access to the results of the scalar operations. AMD GCN GPUs [10] use a separate functional unit and register file for scalar operations and a broadcast network for distribution of scalar values to the vector functional units. This has multiple drawbacks: 1. Transferring values over long distances to all ALUs of the vector functional units requires serious amounts of energy. 2. Often either the scalar units or the vector functional units have to stay idle or execute scalar operations on the vector unit, because the kernel is only executing operations of one type. 3. This design also requires separate registers files for scalar and vector files. Because of that the ratio of scalar and vector registers can not be dynamically adjusted for the requirements of each kernel.
In the DART architecture, proposed in this report, both scalar and vector values are stored in the same register file and processed by the same functional units. Scalar operations are executed on the functional unit exactly like a warp with just one active thread. Scalar registers are stored in the same register file as vector values. Scalar values are just using a different way of indexing the register file. 32 Scalar registers occupy the same space as a single vector register and all threads access the same set of registers instead of separate per-thread registers. When vector operations use scalar input registers they do not need a broadcast network to access the value. The scalar value can even

be fetched just once from the SRAM banks of the register file and can be stored at the operand collector. This way even vector operations can save some energy by scalarization. Vector operations that use scalar operands will produce a smaller number of SRAM accesses and data transfers over the crossbar connecting the register file banks and the operand collector.

3.3.3 Overview of the Proposed Architecture

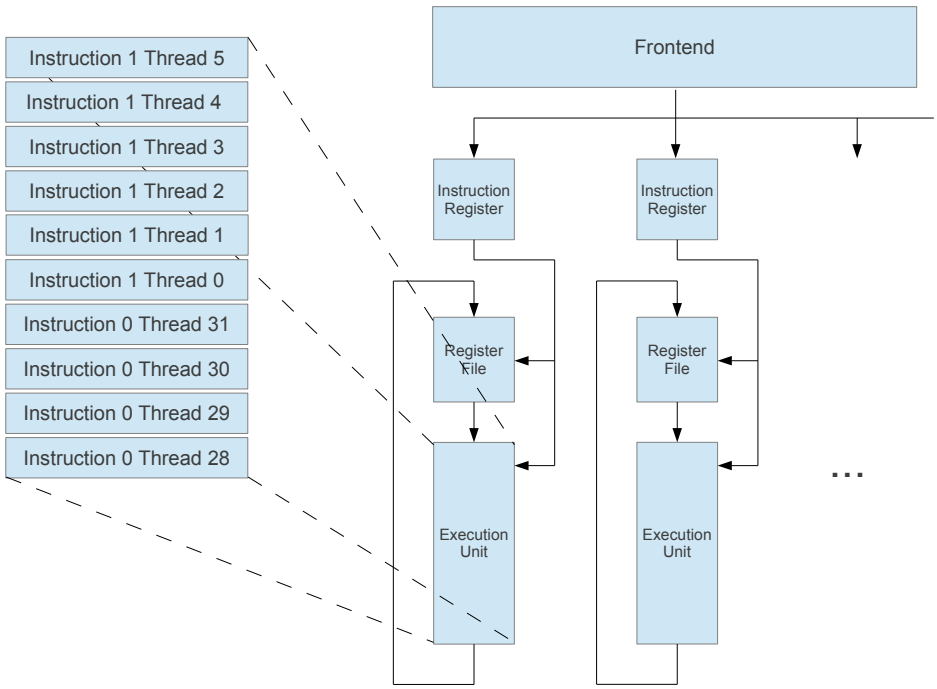


Figure 3: Diagram that shows the basic architectural idea of a TSIMD GPU core with multiple execution units

Figure 3 shows the basic idea of the proposed architecture. The execution is handled by lanes formed from a register file, an execution unit and an instruction register. The instruction register stores the control vector at the execution unit and automatically cycles through all active threads within the warp. After all threads of the instruction have been sent to the execution unit the next instruction can be received from the frontend.
However instructions are still fetched, scheduled and decoded in the front end on a per-warp level. While one of these lanes is busy with the self-managed execution of the SIMD instruction, the frontend is able to issue instructions to one of the other execution units. If all threads in all warps are active, the frontend needs to supply one instruction per 32 executed operations. If a smaller number of threads in the warp is active, due to divergence, the lane will sooner be able to accept a new instruction from the frontend. DART's capability to handle divergent workloads stems from the flexibility of the TSIMD lanes and the more powerful frontend's ability to supply instructions at a higher rate if needed.
Hardware support for scalarization is implemented by adding another way of register address calculation to the register file. While normally every threads maps its registers to a different slot, scalar registers only need one slot per warp. When scalar operations are executed the active mask

is replaced by an active mask with only a single thread active. This way the DART lane only has to execute operations once per warp instead of once per active thread.

3.3.4 Compiler Support for Scalarization

To enable efficient use of scalarization not only the hardware must be able to execute scalar instructions efficiently but they must also be used where possible without placing additional workload on the programmer. For this reason the hardware must also be supported by a new kind of compiler that is able to mark scalar operations and scalar operands in regular SPMD GPU kernels. We developed a new algorithm that is able to extract a large number of scalar operations directly from the PTX code that is produced by the already available compilers for GPU kernels. Figure 5 shows the proposed scalarization algorithm. To do scalarization, it first optimistically tags all instructions and all operands as scalar. Only the instructions that read from the built-in threadid.x register are marked as non-scalar. All operations that read results from operations that use non-scalar input values are then also marked non-scalar until all scalar operations have only scalar inputs. But these dataflow dependencies to the threadid.x register are not the only possible way a register can be non-scalar. A register can also be non-scalar because of the control flow. A less effective way of dealing with this control flow dependency is to only scalarize instructions where the control flow can be shown to be always convergent. The control flow is convergent if all threads are following the same control flow. This, however, prevents many possible opportunities for scalarization. We developed a code analysis technique on the PTX code that is able to recognize many scalar instructions.

Figure 4 shows a simplified example of CUDA source code. The calculation of e and f clearly has a control flow dependency on the threadId.x register, however because e and f are not live after the warp reconvergeces they can still be scalarized by our code analysis. d however can not be scalarized because while d has no data flow dependency on threadId.x, it is used after the warp reconvergences and different threads have different values of d that depend on the control flow of the thread. We developed an algorithm to analyze the data and control flow and live time of the variable within the kernel to detect which registers can be scalarized and which registers can not be scalarized.

```
__global__ void example(float a, float b, float* c)
{
        float d;
        if (threadIdx.x > 16) {
                float e = a+b;
                d=e*e*e;
        } else {
                float f = a+b;
                d=f*a;
        }
        c[threadIdx.y*a+threadIdx.x] = d;
}
```

Figure 4: Scalarization example

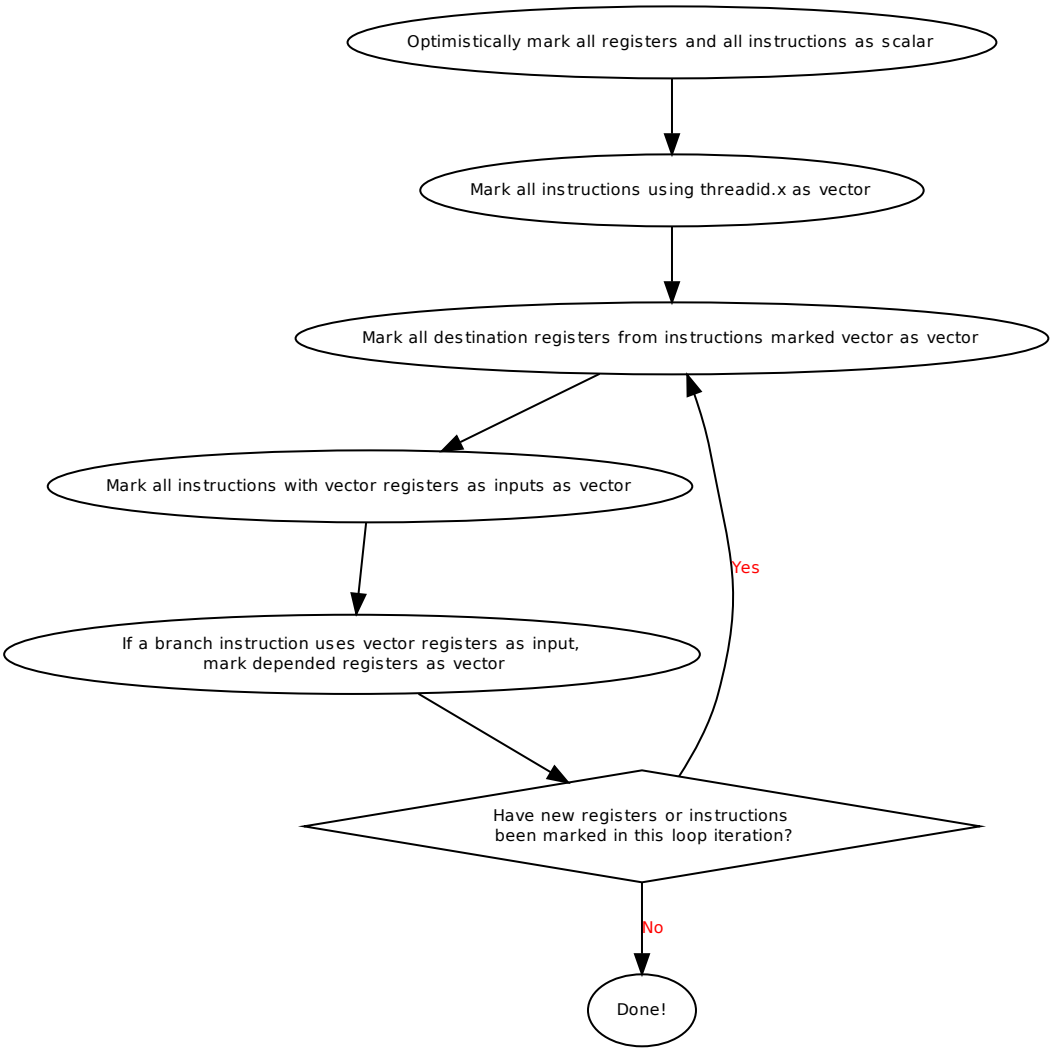


Figure 5: Scalarization algorithm

3.3.5 Energy Efficiency

The DART architecture should be more energy efficient than conventional GPUs for many different reasons. It enables higher performance without idle units with only small hardware changes. The number of operations needed to execute a given kernel can almost always be reduced by scalarization. Scalarization also makes the register file more efficient. A smaller register file can provide storage for the same number of threads or a register file with the same size can store the values for more threads. Non-Scalar operations that use scalar inputs only need to read the scalar values once from the register file and after that they can store the value directly at the execution unit. With DART the frontend does not need to drive its control vector to all ALUs on each issued SIMD instruction but instead the control vector only needs to be transfered to the TSIMD unit where it is executed. The control vector is then stored locally and used unchanged for multiple cycles. Another property

of DART that should help the energy efficiency of this architecture is that the same operation for many different threads is executed consecutively on the same functional unit. In many applications the values processed by one thread are similar or even identical. This reduces the energy needed per operation because it lowers the average amount of switching within the functional unit per processed operation.

Based on the discussions with ThinkSilicon we believe that DART has a negligible area overhead compared to a conventional GPU. Accurately quantifying this small overhead however requires a full design.

4 Experimental Evaluation

4.1 Experimental Setup

We modified GPUSimPow to simulate DART GPUs. The simulator was modified to model the proposed architecture including the hardware support for the scalarization and the code analysis to detect scalar instructions and registers. GPUSimPow simulates the hardware and the driver of a GPU. The DART simulation support was added to the hardware simulation part of the simulator, while the code analysis is part of the simulated driver.

4.1.1 Synthetic Benchmarks

We developed a microbenchmark for testing the performance at different levels of divergence. To test different levels of divergence we can configure the microbenchmark's control flow to branch into 1 to 32 divergent flows per warp. As a baseline we configured a GPU with a single core with NVidia GT200 based parameters. GT200 based GPUs use SIMD units with 8 execution units. The parameters of our simulated TSIMD are also based on a single GT200 core, but this time with 8 TSIMD lanes. Both GPUs have the same theoretical peak performance.

4.1.2 Benchmarks

To test the DART architecture in real world application we used the kernels developed in Task 3.5 and described in Report D3.3.4, we also used kernels from Task 3.2, which are described in Report D3.3.6. We also employed the matrixmul kernel from the NVidia CUDA SDK.

4.2 Results

4.2.1 Synthetic Benchmarks

Figure 6 shows the result of our microbenchmark runs. With all threads on the same control flow conventional GPUs and DART are showing similar performance. But even with just two to three different control streams conventional GPU performance shows a steep drop while DART is still able to archive almost peak IPC. On highly divergent workloads the DART GPUs is approximately 4x faster than a conventional GPU. This reflects the increased frontend performance.

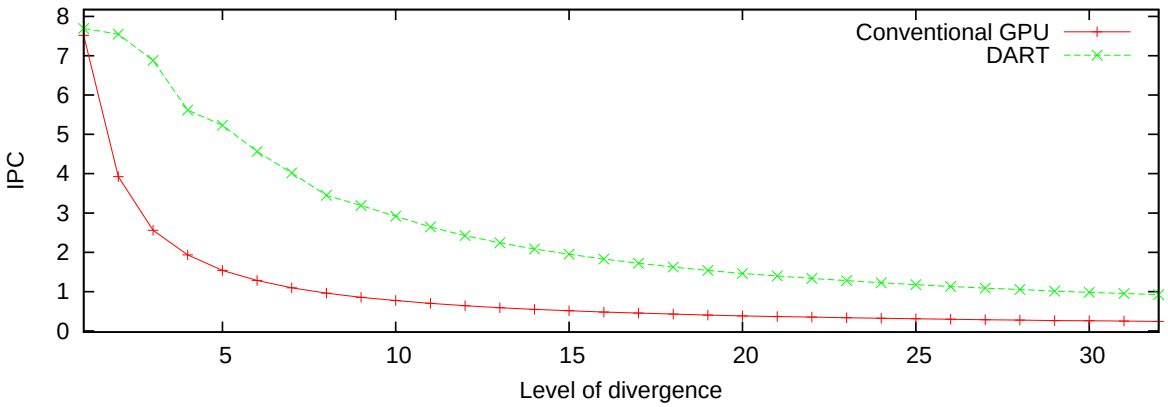


Figure 6: DART IPC vs. Conventional GPU IPC for different levels of divergence

4.2.2 Real Benchmarks

In Figure 7 we display the results for running several kernels on simulated GPUs. We normalized all runtimes to the runtime of the kernel on a conventional GPU and show the speedup over this GPU. The highly divergent benchmarks viscache_borders and h264 MC chroma show the largest improvements. Scalarization often provides an additional improvement over DART. In matrixmul it is critical for good performance. While matrixmul on DART without Scalarization shows a decreased performance, scalarization allows running more warps using the same register file size and leads to slightly better performance, unfortunately it is still slower than using a conventional GPU. The reason for the reduced performance in DART for matrixmul still needs to be identified. We, however suspect memory instruction latency to be the reason for this behaviour. GPUs perform a step called memory coalescing to bundle together the memory transfers from multiple threads into a smaller number of transactions. This step needs more time in DART, because not all addresses are available at the same time. We already have an idea how this coalescing step can be optimized for the DART architecture. The IDCT kernels contain no divergence and only small amounts of redundancy and thus only shows small benefits from DART.

4.2.3 Power & Energy

In Figure 8 and Figure 9 we show preliminary power and energy estimations for DART and DART with Scalarization. These results, however, do not yet model some of the proposed energy benefits by DART. The power used by DART can often be slightly higher than the power of a conventional GPU but this is only because the DART GPU is able to utilize its executions units more efficiently. The energy needed for executing the kernels on DART is lower for all benchmarks that show a speedup. Only the matrixMul kernel requires more energy because it also executes slower.

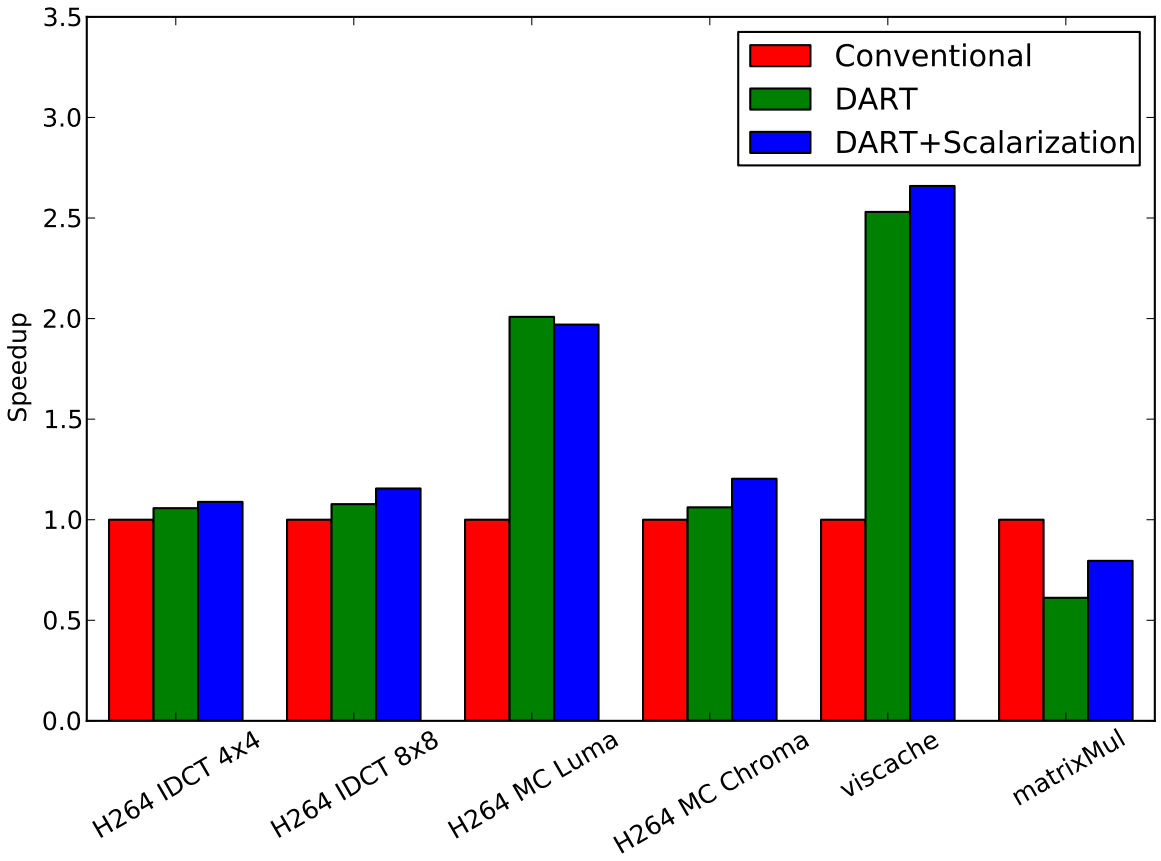


Figure 7: Speedup of DART and DART+Scalarization over conventional GPU

4.2.4 Scalarization

In this section we analyze the results of the scalarization even further. In figure 10 we show how many scalar and vector registers are needed. In each kernel a significant number of registers was identified as scalar. An average of 41.5% of the registers has been identified as scalar. This results in a big improvement of register file efficiency. Figure 11 shows how many 32-bit register slots are needed per warp. Each vector register needs 32 register slots per warp, while scalar register only need one register slot per warp and register. With scalarization warps reduce their storage requirements in the register file by more than 19%. This can be used to either reduce the register file and save area and power or can be used to improve performance by enabling more warps to run at the same time and thus increasing the tolerance of the GPU for memory latency. This can be seen in the performance results of the matrixmul benchmark.

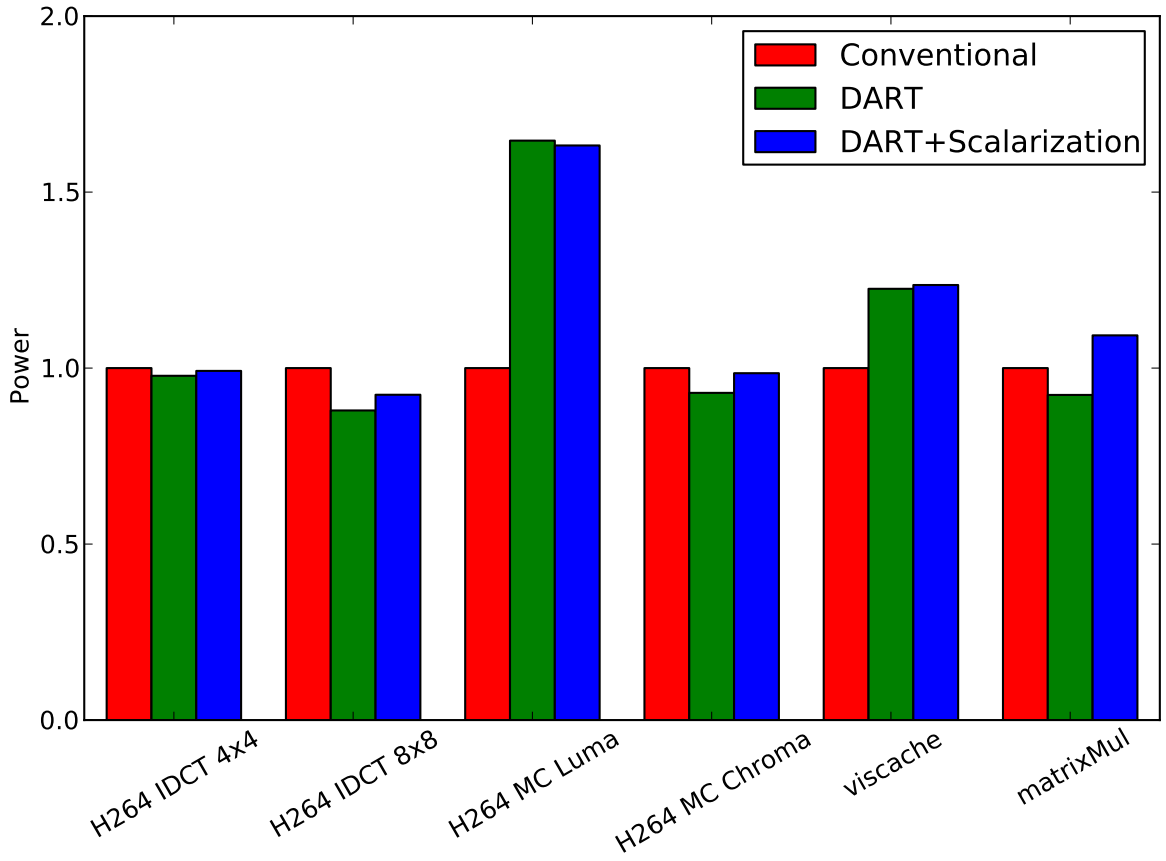


Figure 8: Normalized Power for DART and DART+Scalarization over contentional GPU

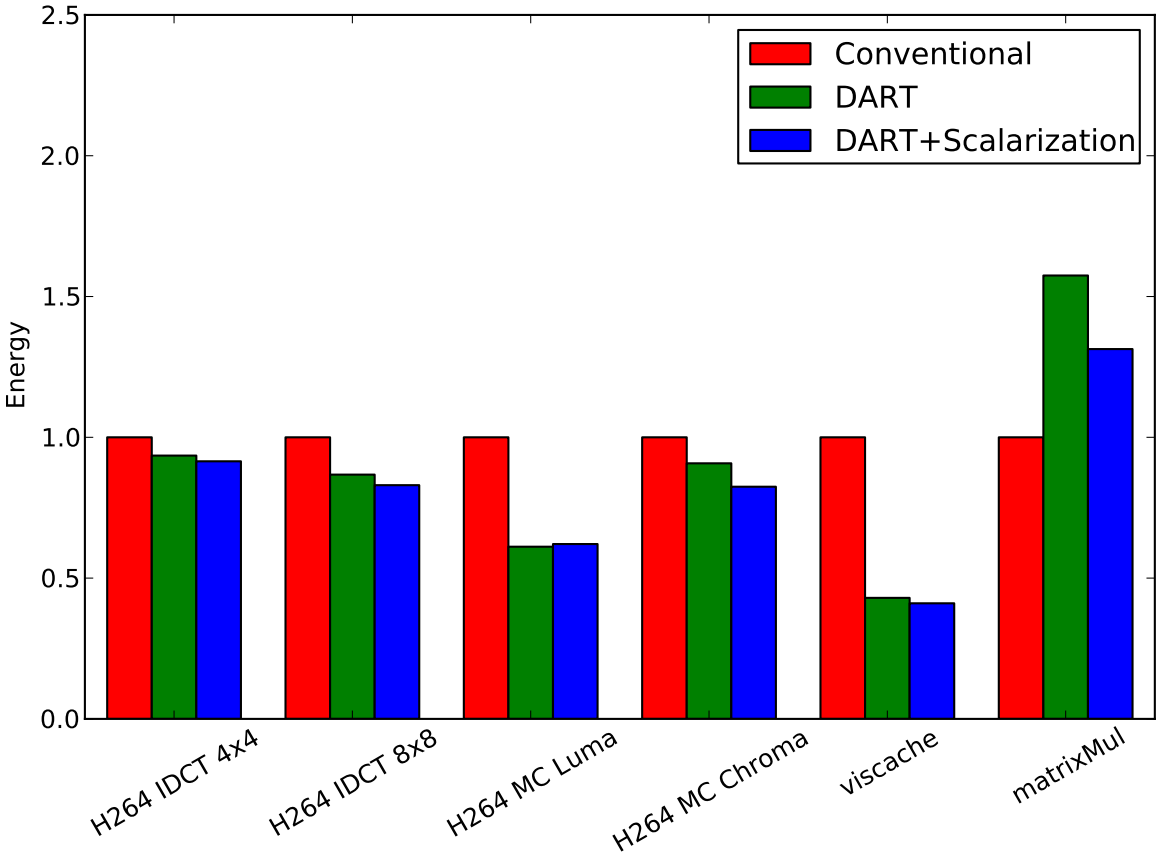


Figure 9: Normalized Energy for DART and DART+Scalarization over contentional GPU

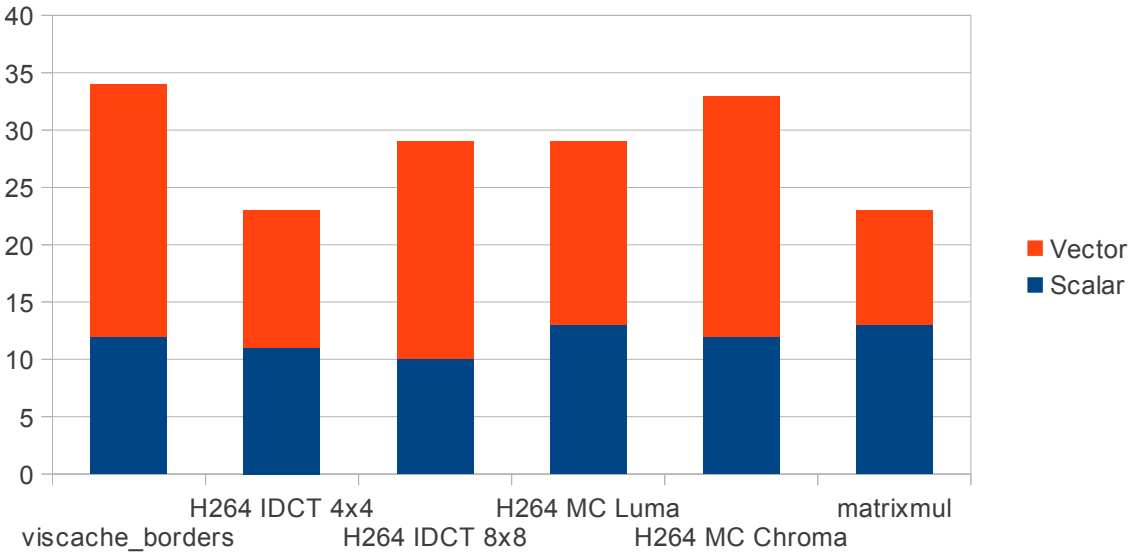


Figure 10: Scalar and Vector Registers per Kernel

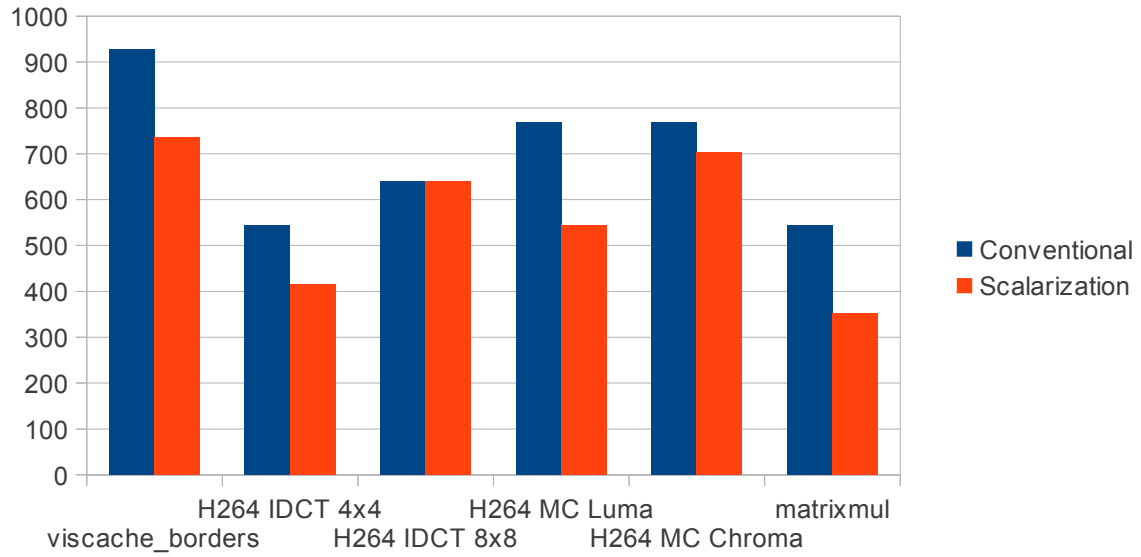


Figure 11: 32-Bit Register slots per warp

5 Status

For this task the following criteria were defined to evaluate the results:

Minimum results: Architectural enhancements are conceived so that at least one of the kernels developed in T3.5 can be executed efficiently on a GPU with these architectural enhancements.

Expected results: Architectural enhancements are conceived so that at least two of the kernels developed in T3.5 can be efficiently executed on a GPU with these architectural enhancements.

Exceeding expectations: All kernels developed in T3.5 can be efficiently executed on a GPU with these architectural enhancement.

Based on our research the new architecture should be able to provide significant performance enhancements and energy savings for four kernels developed in Task 3.5 and one kernel developed by AiGameDev as well. For this reason we evaluate the outcome of this task as **expected results**, but very close to exceeding expectations because of the very promising results for the AiGameDev kernel.

6

Advances over State of the Art

Workload characterization to understand the behavior of workloads is not new. Kerr et al. [6] proposed a set of metrics for GPU workloads and use these metrics to analyze the behavior of GPU workloads. Che et al. [3] presented characterization of the Rodinia benchmarks suite and also presented performance results for the NVIDIA GeForce GTX480. In this sense, we also use workload characteristics to identify the performance bottlenecks. In addition to this, we also use workload characteristics to study their correlation with GPU component power consumption. Based on the correlation results we categorize the workload into three different categories and show that there is a significant change in components power consumption across the three categories. We see that the most power consuming components are different between the selected categories. This information can be used to prioritize GPU components for power optimization according to the category.

A few people tried make execution of branch divergent code more efficient [2] [9]. These papers however did not focus on power and provide techniques that are only efficient with specific branch patterns. When we started working on the SIMD-MIMD architecture, temporal SIMD was only mentioned by Keckler et al. [5]. However during the last year several publications about TSIMD appeared. NVidia patented a TSIMD-like technique. This patent [7] was published in February 2013. It however does not provide any performance details or scalarization techniques. Lee et al [8] provide a scalarization technique and also mention implementation using TSIMD. Their work, however, provides only compiler level statistics without detailed architectural level simulation. Their scalarization algorithm also scalerized only instructions where convergence can be proven. This unnecessarily strict criterion limits the opportunities for scalarization. Coutinho et al. provide a good theoretical foundation of scalarization [4], but they do not apply their algorithm to GPUs with hardware support for scalarization. They also require a change of the representation of the kernels while our algorithm is able to work directly on PTX or similar representations.
Our scalarization algorithm works on PTX without a change of representation and can scalarize many instructions at places where control flow is not convergent. We combine temporal SIMD with scalarization and provide detailed architectural simulation. With the extended GPUSimPow the design space of TSIMD based architectures can be explored. In DART multiple lanes share the same frontend and shared memory. While some causes of stalls in conventional GPU disappear, e.g.: intra-warp shared memory bank conflicts, new sources of stalls e.g.: inter-warp shared memory bank conflicts or register file skipping stalls appear. With our extended simulator the TSIMD concept can be validated and optimized.

7 Conclusions

To develop a new SIMD-MIMD architecture we first characterized GPGPU workloads using simple and well known workload metrics to identify the performance bottlenecks. We found that the benchmarks with branch divergence do not utilize the SIMD width optimally on conventional GPUs. We also studied the performance bottlenecks of motion compensation kernel developed in Task 3.2 and showed that increasing the maximum limit on CTA and shared memory can significantly increase in performance and save energy. We also studied the correlation between workload characteristics and GPU component power consumption. In addition we categorize the workload into high, medium, and low IPC category to study the power consumption behavior of each category. The results show a significant change in components power consumption across the three categories of kernels. We believe this is a vital information for computer architects and application programmers to prioritize the components for power and performance optimizations. Guided by this information we proposed a new architecture which can handle branch divergence efficiently.

The proposed SIMD-MIMD architecture shows encouraging results. It shows very good speedups for highly divergent workloads and still provides benefits for some workloads with low divergence. This makes it a promising architecture to enable more usage of GPUs in less regular computations that are not a good fit for current conventional GPUs, while still keeping the high energy efficiency characteristics of GPUs. It also enables the efficient inclusion of scalarization. Scalarization can be used to reduce the size of the register file and save both area and energy. At the same time it removes many redundant calculations, one of the key inefficiencies of the SPMD programming model of current GPUs.

As a future work more benchmarks need to be tested in the SIMD-MIMD architecture to optimize it even further. We are not yet able to model some of the expected energy benefits of the architecture. Additional work in the simulator and in the power model is needed to quantify the size of these effects. We expect an additional reduction of the power consumption due to these effects.

References

[1] Emily Blem, Matthew Sinclair, and Karthikeyan Sankaralingam. Challenge Benchmarks that must be conquered to sustain the GPU revolution. In *Proceedings of the 4th Workshop on Emerging Applications for Manycore Architecture*, June 2011.

[2] N. Brunie, S. Collange, and G. Diamos. Simultaneous Branch and Warp Interweaving for Sustained GPU Performance. In *Proc. of 39th Annual Int. Symp. on Computer Architecture*, pages 49–60, 2012.

[3] Shuai Che, J.W. Sheaffer, M. Boyer, L.G. Szafaryn, Liang Wang, and K. Skadron. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, IISWC, 2010.

[4] B. Coutinho, D. Sampaio, F.M.Q. Pereira, and W. Meira. Divergence Analysis and Optimizations. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 320–329, 2011.

[5] S.W. Keckler, W.J. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the Future of Parallel Computing. *Micro, IEEE*, 31(5):7–17, 2011.

[6] A. Kerr, G. Diamos, and S. Yalamanchili. A characterization and analysis of PTX kernels. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, IISWC, 2009.

[7] R. M. Krashinsky. Temporal SIMT Execution Optimization, 2011. Patent, US 20130042090 A1.

[8] Yunsup Lee, Ronny Krashinsky, Vinod Grover, Stephen W. Keckler, and Krste Asanovi. Convergence and scalarization for data-parallel architectures. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2013.

[9] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt. Improving GPU Performance via Large Warps and Two-level Warp Scheduling. In *Proc. of the 44th Annual IEEE/ACM Int. Symp. on Microarchitecture*, pages 308–317, 2011.

[10] AMD Radeon Graphics Technology. AMD Graphics Cores Next (GCN) Architecture White Paper, 2012.