Design Space Explorations for Streaming Accelerators using Streaming Architectural Simulator

Muhammad Shafiq
Centre of Excellence in Science & Advanced Technologies (CESAT)
Islamabad, Pakistan
mshafiqshami@yahoo.com

Miquel Pericàs
JST, CREST
Tokyo Institute of Technology
Tokyo, Japan
pericas.m.aa@m.titech.ac.jp

Nacho Navarro, Eduard Ayguadé
Computer Sciences
Barcelona Supercomputing Center
Barcelona, Spain
{nacho.navarro, eduard.ayguade}@bsc.es

***Abstract*—In the recent years streaming accelerators like GPUs have been pop-up as an effective step towards parallel computing. The wish-list for these devices span from having a support for thousands of small cores to a nature very close to the general purpose computing. This makes the design space very vast for the future accelerators containing thousands of parallel streaming cores. This complicates to exercise a right choice of the architectural configuration for the next generation devices. However, accurate design space exploration tools developed for the massively parallel architectures can ease this task.**

The main objectives of this work are twofold. (i) We present a complete environment of a trace driven simulator named *SArcs*[1] (Streaming Architectural Simulator) for the streaming accelerators. (ii) We use our simulation tool-chain for the design space explorations of the GPU like streaming architectures.

Our design space explorations for different architectural aspects of a GPU like device are with reference to a base line established for NVIDIA's Fermi architecture (GPU Tesla C2050). The explored aspects include the performance effects by the variations in the configurations of *Streaming Multiprocessors* ,*Global Memory Bandwidth*, *Channels between SMs down to Memory Hierarchy* and *Cache Hierarchy*. The explorations are performed using application kernels from Vector Reduction, 2D-Convolution, Matrix-Matrix Multiplication and 3D-Stencil. Results show that the configurations of the computational resources for the current Fermi GPU device can deliver higher performance with further improvement in the global memory bandwidth for the same device.

I. Introduction

In computer architecture research, design space explorations are a key step for proposing new architectures or modifications in an existing architectural configuration. During the last decade, computer architecture research has witnessed a shift from a single core to multicore processors and expectedly the future of computer architecture research will be revolving around the parallel architectures. This has made the design space explorations a great challenge for the computer architects. The designs of new high performance computing (HPC) systems which are sharply converging towards the idea of exploiting massively data-level parallelism on thousands of compute cores has further complicated this challenge. The one way to overcome these challenges is the development of new architectural exploration tools by taking into account the new research trends in computer architecture.

GPUs introduced just a decade back are now considered an effective part of many HPC platforms [2]. GPUs are throughput-oriented devices. A single GPU device can contain hundreds of small processing cores. These use multi-threading to keep a high throughput and hide memory latency by switching between thousands of threads. In general, the architecture of a GPU consists of dual level hierarchy. The first level is made of vector processors, termed as streaming multiprocessors (SMs) for NVIDIA GPUs and SIMD cores for AMD GPUs. Each of the vector processor contains an array of simple processing cores, called streaming processors (SPs). All processing cores inside one vector processor can communicate through an on-chip user-managed memory, termed local memory for AMD GPUs and shared memory for NVIDIA. On a single HPC platform, GPUs and CPUs can run in parallel but execute different types of codes. Generally, the CPUs run the main program, sending compute intensive tasks to the GPU in the form of kernel functions. Multiple kernel functions may be declared in the program but as a common practice only one kernel is executed on one GPU device at a time. Therefore, most of the HPC platforms uses configurations of single CPU with multiple GPUs to run kernels independently and in parallel. However, the performance driving factor remains the basic architecture of the device being used in all the GPUs of the platform.

GPUs are still considered at an early stage of an era of their architectural growth and innovations. As compared to an enormous amount of efforts devoted to application development for GPUs, only a little has been done on supporting tools for performance characterization and the architectural explorations. Only a few years back, GPUs were only an effective choice for the fine-grained data-parallel programs with limited communications. However, these were not so good for programs with irregular data accesses and a lot

[1]The *SArcs* simulator was initially presented by Shafiq et al. in ACM International Conference on Computing Frontiers 2012, Cagliari, Italy for their work on *BSArc* [1]

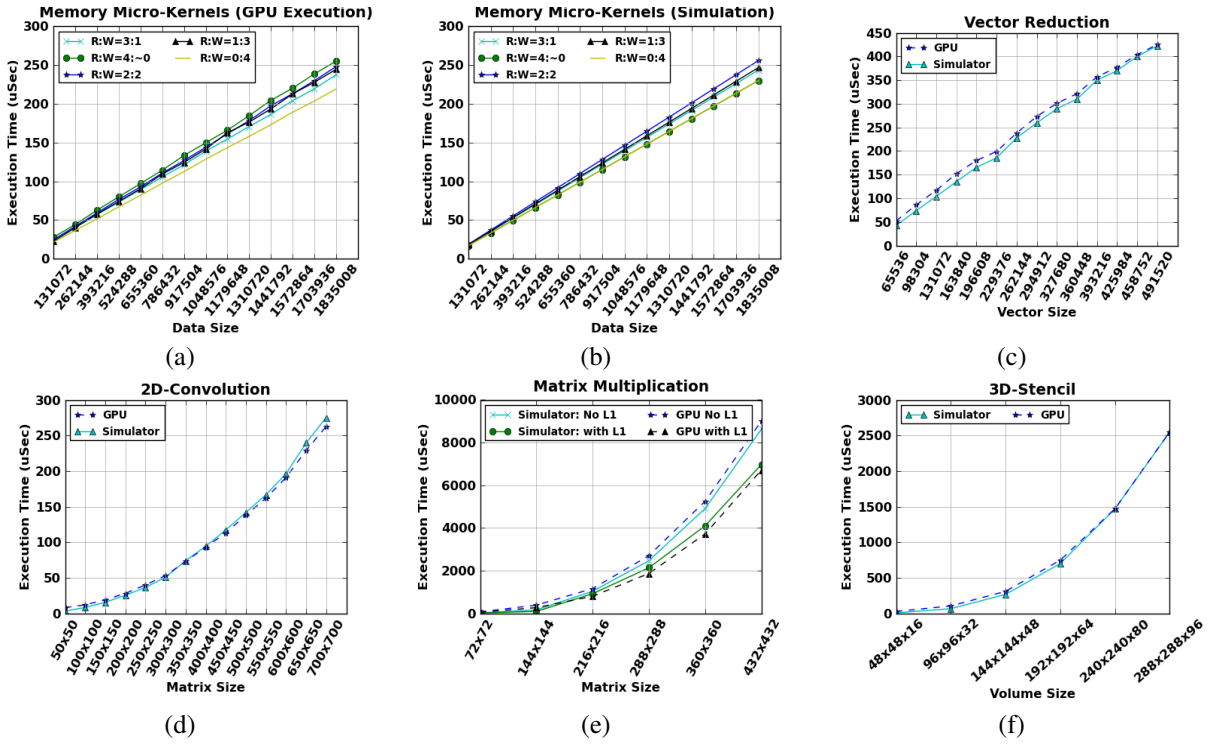


Fig. 1. Establishment of the accuracy of the simulator (SArcs) by performance characterization against the real GPU for the base line architecture (NVIDIA's Tesla C2050) (a) Memory Micro-Kernels (real GPU Executions) (b) Memory Micro-Kernels (Simulated Executions) (c) Vector Reduction using shared Memory (d) 2D-Convolution using shared memory (e) Matrix Multiplication with/without L1 (f) 3D-Stencil Kernel using shared memory

of communication [3], [4]. This is because the original architecture of GPU was designed for graphics processing. In general, these graphical applications perform computations that are embarrassingly parallel. Later, the GPU architecture was improved [5] to be able to run general purpose programs under CUDA [6] and OpenCL [7] like programming models. The general purpose programs with arbitrary data-sets may or may not perform well on the GPU like streaming devices. This motivates the newer generation of the GPUs like the NVIDIA's Fermi architecture to incorporate both the level-1 and the level-2 caches in their memory hierarchy. However, further architectural improvements in these devices can make them most interesting choice for the efficient parallel computing.

The design choices for GPU like streaming architectures are so large and diverse that these architectures are still finding, on one hand, a balance between the available bandwidth and the on-chip computational resources and on the other hand, a balance between generality and speciality of the underlying architecture. This imposes a need for finding an easier but at the same time highly effective way to rapidly explore design spaces for the new GPU like proposals. We – in this work – present architectural explorations and its methodology for GPU like streaming architectures. These explorations are done using a locally developed environment of a trace driven simulator called *SArcs* (**S**treaming **Arc**hitectural **S**imulator). This simulation framework uses CPU code projections for GPU performance modeling on a detailed cycle accurate streaming simulator. This platform independent simulation infrastructure, on the one hand, is very useful for the design space explorations for the future GPU devices and on the other hand, it can be used for performance evaluation of different applications on the existing GPU generation with a high accuracy. The modules of *SArcs* are written in C and C++. These are enveloped inside a python script to run in an automated way which starts by grabbing the application source file and finalizes showing performance results. Some performance characterization results of the *SArcs* are shown in Figure 1 and explained in the next section (section II). However, for high accuracy, it is required to do some optimizations by taking into account the real CUDA compiled code. To the best of our knowledge *SArcs* tool is the first trace based GPU architectural simulator which can also be used independent of the requirements of having any kind of GPU environment.

In our evaluations we explore different architectural aspects of a GPU like device against a base line established for NVIDIA's Fermi architecture (GPU Tesla C2050). The explored aspects include the performance effects by the variations in the configurations of *Streaming Multiprocessors* ,*Global Memory Bandwidth*, *Channels between SMs down to Memory Hierarchy* and *Cache Hierarchy*. The explorations are performed using application kernels from Vector Reduction, 2D-Convolution, Matrix-Matrix Multiplication and 3D-Stencil computations. The results show that the configurations of the computational resources for the current Fermi GPU device can deliver higher performance with further improvement in the global memory bandwidth for the same device.

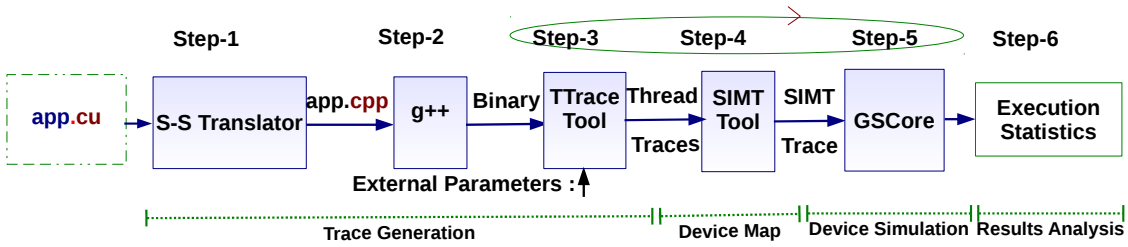


Fig. 2. The Framework of *SArcs*

This paper is organized as follows: we start in section II by giving motivational results for the simulation tool used in the design space explorations of the GPU like streaming devices. The details on the architecture Simulation tool framework are given in section III. This section introduces the process of trace generation, the transformation of the trace to a SIMT (Single Instruction Multi-thread) trace format and the cycle accurate simulator. The details on the design space exploration environment are given in section IV followed by description of evaluated architectural configurations in section IV-D. The results are presented and discussed in section V. After presenting some related contributions in section VI, we conclude in section VII.

II. The Accuracy of the Design Space Exploration Tool

The simulator accuracy is an important factor to be established before that one proceed for design space exploration for a target architecture using that simulator. The proposal on *SArcs* contributes in computer architecture research by providing an automated framework for simulations of streaming architectures like GPUs. *SArcs* can be used either as a standalone system – completely independent of a streaming environment – or it can be connected to other existing simulation related tools. *SArcs* as an independent simulation infrastructure for GPUs does not require to have a physical GPU or any GPU related software tool-chain.

SArcs is a trace driven simulation framework and exploits the fact that an application compiled for any architecture would require to transact the same amount of data with the main memory in the absence of registers or cache hierarchy. Moreover, the computations inside an application can be simulated by the target device latencies. The instruction level dependencies in GPU like architectural philosophy pose least impact on the performance because of zero-overhead switching between the stalled and large number of available threads. However, there could be cases where these dependencies can took longer time but the current version of *SArcs* is not accommodating these corner cases. *SArcs* creates an architectural correlation with the target device by passing the source code through a source to source translator followed by a thread aware trace generation. This trace is used by a device mapping process which transforms the trace into a SIMT trace specific for a GPU architecture. The SIMT trace is passed through a cycle accurate simulator to get the performance and related statistics. The detailed design description of the simulator framework is given in section III.

The simulation results of *SArcs* and the reference results of real GPU (NVIDIA's Tesla C2050) based executions for the performance characterization of different application kernels are shown in the Figure 1 (a) to (f). The Memory Micro-Kernels shown in the Figure 1 (a) & (b) are used for the fine detailed analysis of the simulator targeting the evaluations of the simulator memory behaviour. These memory micro-kernels are based on five different types of memory accesses during single execution of the kernel. We categorize these single kernel accesses in the ratio between consecutive reads (R) and writes (W). These ratios are R:W = 0:4 , 1:3, 2:2, 3:1 and 4:0. In order to avoid *nvcc* compiler from optimizing out the R:W=4:0 case, we use an external flag passed from command prompt to implement the kernel for only a conditional write. This flag always remain *false*. The descriptions of application kernels (Figure 1 (c) to (f)) are given in the section IV-A. It can be observed that in all cases, the *SArcs* simulated results accurately follow the real GPU based executions. The results for matrix-multiplication (MM) kernel also present the real and simulated behavior of L1 cache. Other kernels use shared memory to exploit data locality thus makes only a little use of L1 cache. The simulation framework apply a large set of architectural optimizations including the ones briefly described in section III-B. In our all test cases the average error in the performance prediction using our design space exploration tool chain (*SArcs*) remain less than 10% of the real executions. This high level of accuracy of the architectural exploration tool promises that the predictions for new architectural configuration would also be accurate ones.

III. The Framework of Design Space Exploration Tool

The basic goal of the Streaming Architectural Simulator *SArcs* is to provide a simulation platform for streaming architectures that could be used for applications performance analysis or to experiment around the architectural innovations. These objectives are achieved by working through different stages of the *SArcs* framework. These stages – as shown in Figure 2 – consist of the *Trace Generation*, the *Device Mapping*, the *Device Simulation* and the *Results Analysis*. The Figure 2 also shows that these stages are executed in different

```
1  /* kernel_name dimGrid,dimBlock >>> (a_d,b_d,c_d,iter); */
2        blockDim.x = dimBlock.x;
3        blockDim.y = dimBlock.y;
4  printf("GDim.y , GDim.x , BDim.x , BDim.y , BId.y , BId.x , TId.y , TId.x\n");
5  printf(":>REF:>%p %p %p %p %p %p %p %p<:REF<:\n", &dimGrid.x,&dimGrid.y,&blockDim.x,&blockDim.y,
6         &blockIdx.y,&blockIdx.x,&threadIdx.y, &threadIdx.x);

8  printf("BId.y , BId.x , TId.y , TId.x , GDim.y , GDim.x , BDim.x , BDim.y \n");
9  printf(":>PAR:>%ld %ld %ld %ld %ld %ld %ld %ld<:PAR<:\n",dimGrid.x,  dimGrid.y, blockDim.x, blockDim.y,
10        blockIdx.y, blockIdx.x, threadIdx.y, threadIdx.x );

12    for (blockIdx.y=0; blockIdx.y< dimGrid.y; blockIdx.y++)
13                for(blockIdx.x=0; blockIdx.x< dimGrid.x; blockIdx.x++)
14                        for(threadIdx.y=0; threadIdx.y< dimBlock.y; threadIdx.y++)
15                                for(threadIdx.x=0; threadIdx.x< dimBlock.x; threadIdx.x++) {

17                                  kernel_name (a_d,b_d,c_d,iter);

19                                }
```

Fig. 3. An example code insertion for the replacement of the target gpu kernel call

steps. The steps 3 to 5 can be repeated for the number of device kernels in an application and/or as many times a device kernel requires to run with different inputs. A brief introduction for the different stages of the *SArcs* framework is given in the next sections.

A. Trace Generation

SArcs supports CUDA programming model. The users of *SArcs* are only required to write a plain CUDA program (The *main* and the device *kernel(s)*) for an application. The users can use CUDA specific API's inside the device kernel. However, it is not allowed to call any application specific API's for the standalone version of *SArcs*. The CUDA source file(s) for an application is processed by a source to source translator (*S-S Translator*) before compilation with the *g++* compiler in *step-2* as shown in the Figure 2. After compilation, the generated binary of the application is forwarded to a thread aware tracing tool (*TTrace tool*) to generate the traces. The details on *S-S Translator* and *TTrace tool* are given below:

1) S-S Translator: *S-S Translator* is a source to source translator. It takes in a CUDA program and apply appropriate modifications and additions for two main reasons: (i) Program should be compilable by a GNU *g++* compiler (ii) The added code inside the source forces to output necessary runtime information to support the next stages of the simulator. At first, to make the CUDA code compilable with the GNU compiler, we provide simulator with a modified cuda header file (*mcuda.h*) which satisfies CUDA API calls, internal variables (eg. thread and block IDs) and special identifiers (*__global __shared* etc.).

The *S-S Translator* also inserts additional code at predefined places in the CUDA source file(s) as shown in Figure 3. This code insertion helps the simulator in two ways: (i) To get a detailed trace of target application kernel that needs to be run on the GPU device. (ii) To extract certain information from the code at run-time. The code between lines 2 and 19 – as shown in the Figure 3 – is an example replacement done by the *S-S Translator* for the code in line 1. Line 1 shows a commented CUDA call to a global function (*kernel_name*) that originally has to run on the GPU device. However, the *S-S Translator* commented this call and inserts a code with some assignment statements, *printf* instructions and nested loops. In this example piece of code (Figure 3), the lines 2 and 3 copies values of Block Dimensions to the global variables. Next, the lines 4 to 10 show code inserted to extract some runtime information specific to a code and also specific to a run. The examples of this runtime information are the pointer addresses assigned to the global variables *dimGrid*, *blockDim*, *blockIdx* and *threadIdx*. This information is used during the later steps of the simulation process. The nested loops in the inserted code from lines 12 to 19 calls the target function (*kernel_name*) at the thread granularity (the most inner loop). These nested loops make it possible to generate a complete trace for all the threads (originally CUDA Threads) in a Block (originally CUDA Block) and for all the Blocks in a Grid. It is important to remember that these nested loops work according to the dimensions of a block and the grid dimensions. These dimensions are defined by the user before calling a gpu target function in a CUDA program.

2) TTrace Tool: The modified source code from the *S-S Translator* is compiled with the *g++* compiler at the step-2 (Figure 2) of SArcs framework. The binary of the program is executed with the thread aware trace (*TTrace*) tool. *TTrace* tool uses dynamic instrumentation of the programs in the PIN [8] environment. The target kernel function name (originally the GPU device kernel) can either be given as an external argument or – by default – it is identified by the *S-S Translator* and forwarded to *TTrace tool*. The name of the kernel function allows the tool to only instrument this function. The *TTrace tool* arranges the instruction level trace information in separate thread groups. The main parameters traced by this binary instrumentation tool include the *Instruction Pointers*, *Instruction Ops*, *Memory Addresses*, *Memory Access Sizes* and any calls to the sub-functions from the kernel function e.g the calls to the thread synchronization APIs. In a CPU ISA, the instruction set can be very large. Therefore *TTrace Tool* only identifies common operations and rest of the operations are accommodated under the single identification.

B. Device Map

The *Device Mapping* stage provides an isolation between the user control over the program and the micro-architectural level handling of the program execution by a GPU generation. For example, In the trace generation stage, the user has a

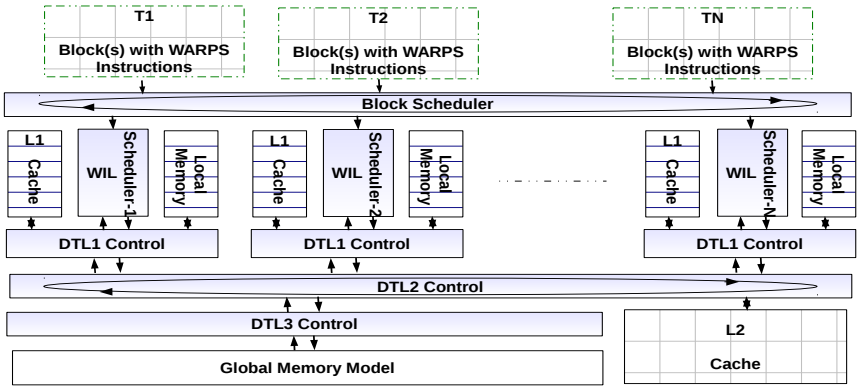


Fig. 4. GPU Simulation Core (GSCore)

control over the CUDA program to adjust the Block and Grid dimensions while the number of threads in a WARP is a micro-architectural feature of a GPU device handled at the *Device Mapping* stage. This stage of *SArcs* framework uses a *SIMT tool* to map a user program trace (the output of *TTrace tool*) for a specific GPU device. The output of the the *SIMT tool* is a SIMT trace which is fed to a *GPU Core Simulator* in the next stage. The *SIMT tool* passes the user program trace through multiple processing phases. Some important phases are given below:

1) Garbage Removal: A real GPU uses some built-in variables represented in CUDA as *dimGrid*, *blockDim*, *blockIdx* and *threadIdx* etc. These variables act as parts of the GPU micro-architecture. However, in our trace generation methodology, these variables acts as global variables with their accesses from the main memory. *SArcs* removes all accesses to these variables from the trace by identifying their address pointers obtained at the execution of program with *TTrace tool*.

2) WARP Instructions Formation:: The user program trace (the output of *TTrace tool*) only groups the instructions traces at thread level granularity. The *SIMT tool* arranges these trace instructions as WARP Instructions and group these WARP Instructions at the Block granularity.

3) Coalescing Effects: The sets of WARP Instructions created in the previous step are further processed by the *SIMT tool* to add the coalescing or un-coalesced effects for the memory access instructions. The *SIMT tool* runs an analysis on the data access pointers for the WARP instructions. A WARP Instruction is split into multiple WARP Instructions if the memory accesses are not coalesced inside the original WARP Instruction. The new WARP Instructions contains accesses which are coalesced.

4) Registers and Shared Memory Handling: In a GPU kernel, the local variables are mapped to the SM (Streaming Multiprocessor) registers. Therefore, the scope of accesses to these local variables inside a GPU remains inside a block allocated to a SM. *SArcs* categorize all stack based accesses inside a kernel either as registered accesses or the shared memory accesses. The shared memory accesses are isolated from the registered accesses based on the base pointer of the shared array and its allocation size. Currently *SArcs* does not handle corner cases like dynamic allocation of shared memory. The shared memory accesses are also organized as WARP Instructions but with separate identifications.

5) Grouping Blocks: We call the new formatted trace generated by the *SIMT tool* as *SIMT Trace*. The *SIMT Trace* is arranged in Blocks. In order to help the *GPU Simulation Core* (Section III-C) to efficiently access the SIMT Trace, *SIMT tool* arranges these blocks in multiple files (called SIMT trace files) which are kept equal to the number of SMs in the target GPU device. This means that if there are *M* number of SMs then the first SIMT file will contain 1^{st}, $M + 1^{th}$, $2 * M + 1^{th}$ and so on SIMT trace Blocks. However, as we will see in the explanations of *GPU Simulation Core* that this arrangement does not create any binding on the choice of SIMT trace Blocks for any SM during the simulation process.

C. Device Simulation

The *Device Simulation* stage models the dynamic effects for various micro-architectural components of a target GPU device. This stage uses *GSCore* (GPU Simulation Core), a cycle accurate simulator specifically developed in-house for simulating the GPU like streaming devices. The functional layout of *GSCore* is shown in the Figure 4. This simulator accepts SIMT Trace files generated by the *SIMT tool*. These SIMT trace files contains Blocks of WARP Instructions as shown at the top of the Figure 4. These Blocks corresponds to the Blocks defined in a Grid for the target application kernel. However, now these Blocks do not contain threads but traces arranged in the form of WARP Instructions. The *GSCore* implements a *Block Scheduler* which is responsible for delivering these Blocks to the SMs – initially – in a round-robin fashion and later based on requests from a SM. SMs are represented as *WIL Schedulers* next to the *GSCore's Block Scheduler* in the Figure 4. The *WIL Scheduler* is named upon its real function which is to schedule the WARPs Instructions & Latency (WIL).

The *WIL Scheduler*, schedules WARPs Instructions from one or more Blocks based on the latencies corresponding to

the operations these WARPs have to do. The latency values for different operations are loaded by the *GSCore* corresponding to a target device from a *GPU Constants File*. This constant parameters file is provided with the *SArcs* frame work. The GPU Constants file keep architectural and micro-architectural parameters for various GPU devices. The latencies due to the instruction level dependencies are normally hidden or unknown in trace driven simulators. However, In case of *GSCore*, the final performance as compared to a real GPU shows almost no effect for these dependencies. This is because of the inherent nature of the real GPU architecture which switches with almost zero-overhead between the WARPs to avoid performance loss due to these dependencies.

The WARP Instructions corresponding to memory transactions are forwarded to the Data transaction Level-1 (DTL-1) control. The memory WARP Instructions are scheduled as first-come first-serve basis or in a round-robin way if multiple requests are available in the same cycle from different WILs (SMs). These memory WARP Instructions goes through the *GScor's* modeled memory hierarchy corresponding to a real GPU. This includes implementation of configurable L1 Cache and Local Scratch Pad memory for each of the WIL Scheduler (i.e for each SM in a real GPU), L-2 Cache and the Global Memory. All levels of *GScor* works in a synchronous way and simulate latencies from going one level to another one. In-case, a memory WARP Instruction is not fulfilled at (DTL-1), it is passed to the DTL-2 – for L2 cache test —. and if required it is forwarded to the DTL-3 level which models a Global memory access. All WARP Instructions which are memory writes are forwarded to the Global memory.

IV. Design Space Exploration Environment

In our explorations for GPU like streaming architectures, we use four application kernels covering one dimensional (1D), 2D and 3D types of data accesses. A brief description of application kernels, the base line GPU configuration and the test platform is given in the following:

A. Application Kernels

In our tests for the various architectural configurations of GPU like device, we use Vector Reduction (VR), 2D-Convolution (CV), Matrix Matrix multiplication (MM), and 3D-Stencil (ST) kernels. The implementations for the two kernels (RD and ST) uses configurations for the shared memory usage. However, the MM and CV kernels do not use shared memory and the performance benefits for these applications only comes from the reuse of data in the standard L1 and L2 caches. The vector reduction kernel uses shared memory along with multiple invocations of the the GPU device during the reduction process of the whole vector to a single value. The convolution kernel uses a constant filter of size 5×5 to be convolve with various sizes of 2D image data sets. The 3D-Stencil kernel implements an odd symmetric stencil of size $8 \times 9 \times 8$. The choice of a kernel implementation is to have diversity in data access patterns and computations from the other kernel.

B. Base Line Architecture

In our design space explorations, *SArcs* simulation infrastructure uses a base line architecture for NVIDIA's GPU of Tesla C2050. This device belongs to Fermi generation [9] of GPUs which is the most recent architecture from NVIDIA. This device has 14 Streaming Multiprocessors (SMs) each contains 32 streaming (scalar) processors. The device is capable of performing 32 fp32 or int32 operations per clock cycle. Moreover, it has 4 Special Function Units (SFUs) to execute transcendental instructions such as sin, cosine, reciprocal, and square root. On the memory hierarchy side the device supports 48 KB / 16 KB Shared memory, 16KB / 48 KB L1 data cache and 768Kbytes of L2 memory.

C. Simulation Platform

The *SArcs* can be compiled for any host machine. The only constraint is that the PIN environment used in *TTrace tool* should have support for that CPU. In our evaluations, we use IBM "x3850 M2" machine. It has 48GBytes of main memory and 4 chips of Intel Xeon E7450, each one with 6 Cores running at 2.40GHz. This machine only helps us to run multiple instances of the simulation in parallel, otherwise a single core machine can be used for running single instance of the simulator. Further, in our case, the host machine uses x86_64-suse-linux and gcc compiler version 4.3.4. The target application kernels are compiled for optimization level 3 (switch -O3). On the GPU side, we use *nvcc* compiler with cuda compilation tool release 4.0, V0.2.1221. We compiled the the CUDA codes using optimization level 3. Further, we use compilation switch *-Xptxas* along with *-dlcm=ca or -dlcm=cg* to enable and disable L1 cache accesses where ever needed.

D. Evaluated Architectural Configurations

Normally, the design space for a processor can be huge one based on the different combinations of the architectural configurations. Therefore, in a realistic way and to give a proof of concept along with some insight for the possible improvements in the current GPU generation, we choose four main architectural components of a GPU device for the experimentations and the explorations. The selection of various test configurations for each component are just based on our intuition and a user of our design space exploration tool can modify these according to one's own requirements.

1) Global Memory Bandwidth: On our base line architecture for the Fermi device, the global memory accesses are processed per warp bases. The maximum bandwidth achievable on the base line configuration is 144 GBytes/second. The memory controllers of the GPU device operates at a bit higher frequency as compared to the SMs operational frequency. This makes it possible that the throughput of the Global memory – in an ideal case – can reach to 128 Bytes/cycle (with respect to the the SM's frequency). The *DTL3* (Data Transaction Level 3) shown in the GPU Simulation Core (Figure 4) is responsible for the bandwidth scaling. In our evaluations, we test the global memory configurations in the ranges from $\times 1$ to $\times 10$ where

the first-one is the base line bandwidth and the later-one is the 10 times of the base bandwidth.

2) Data Channels Between Memory Hierarchy and SMs: The Streaming Multiprocessors at the back-end of a GPU device do data transactions with the front-end memory hierarchy through multiple data channels. The *DTL2-Control* shown in the Figure 4 of *GSCore* handles these channels for the data transactions between the SMs and the memory hierarchy. In the base architecture, there are six channels. In our evaluations we increase and decrease the number of these channels to see their possible effect on the applications performance.

3) Cache Memory: Our base line device uses both L1/L2 cache hierarchy to cache the local and the global memory accesses. However, It is possible that both or anyone of these caches can be turned-on or turned-off at any time. Both caches are fully configurable for any cache size. However, the cache-line size is fixed. the cache line size for L1 cache is 128 bytes and it is 32 Bytes for the L2 cache. Moreover, these caches can be configured for two types of replacement policies: LRU and FIFO.

4) Streaming Multiprocessors: Streaming Multiprocessors (SMs) work as the vector processing units. This is the same as we model SMs in our simulation framework. The SM model in the GPU Simulation Core (GSCore) of our *SArcs* framework consists of *WARP Instruction and Latency (WIL) Scheduler*, Local memory, L1 cache and the *Data Transaction Level-1* control. Our simulator implements the L1 cache and Local memory separately. However, both of these in their functionality exactly behaves like a real NVIDIA's GPU. In order to be concise, we did not go for testing of all the internals of the SM rather than we simply vary the number of SMs in a GPU device to see how these changes effect the execution the WARP instructions and eventually effect the overall performance of an application.

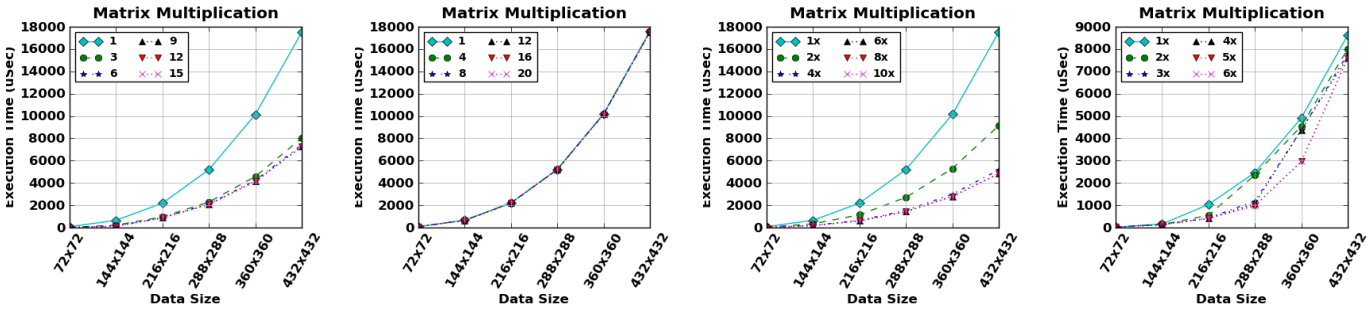
V. Results and Discussion

The results for the evaluated architectural configurations of a GPU like streaming device are shown in in Figures 5 to 8. Here, before that we proceed to discuss the results, we define two terms being used in the discussion. These are the *SM WARP Instructions* and *Global WARP Instructions*. The general descriptions of the *WARP Instructions* formation are given in section-III-B. The *SM WARP Instructions* are the *WARP Instructions* which complete their execution phase inside an SM and the *Global WARP Instructions* consumes cycles inside an SM and as well these are forwarded to the downside memory hierarchy. We are not calling the *Global WARP Instructions* as *Memory WARP Instructions* because if local memory is used inside an SM or there are read hits in the L1 cache then it is quite possible that a number of *Global WARP Instructions* becomes *SM WARP Instructions*. All writes to the global memory are always categorized as part of *Global WARP Instructions*.

The effects of various channel configurations on the application kernels are shown in Figures 5(a), 6(a), 7(a) and the 8 (a). The usage of multiple channels from SMs on the top of a GPU are beneficial in two ways: (i) To keep busy the memory sub-system by forwarding data requests from various SMs (ii) To increase the Bandwidth of the system at L2 cache level. The results show that vector reduction kernel (Figure 8(a)) does not show any significant performance effect due channel variations. The basic reason for this behavior is that the reduction kernel uses local memory for the reduction process. In this case only the reduction result for two values is reused with the next one and this process of reuse remain inside the shared memory. Ultimately only a single value is written back to the main memory for a single call to the device. Therefore the overall data required to transact with the global memory for this kernel is also very small. This means that the application kernel dominates with the *SM WARP Instructions* and does not show any effect with the channel variations. The same reason is true for the behavior of the reduction kernel for the corresponding results of the Memory Bandwidth and L2 cache shown respectively in the Figures 8(c) and (d). However, the reduction kernel shows performance improvements for the increase in the number of SMs as shown in Figure 8(b). This makes sense because the kernel is dominated by the *SM WARP Instructions* and increasing the number of SMs increase the parallelism in the execution. However, this performance due to parallelism with more number of SMs is saturated for 16 SMs because of the fixed channel configuration (6 in the base case) and the ultimate limit of the global memory bandwidth. On the other extreme, it can be seen that the matrix multiplication kernel does not show any effect for the Number of SMs as shown in the Figure 5(b). The MM kernel does not use local memory therefore this kernel dominates with the *Global WARP Instructions*. In this case the requests generated by a single SM saturates the memory sub-system (L2 and L1 are disabled in the test). Therefore, increasing the number of SMs does not show any significant variation in the results for the kernel.

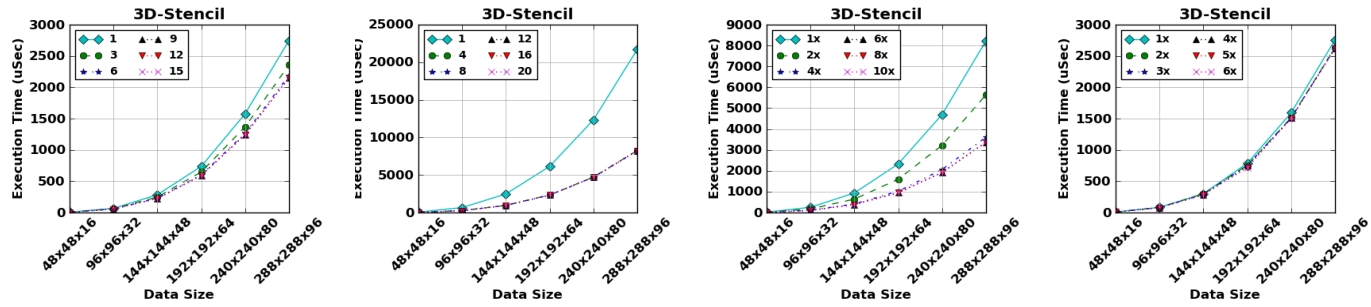
The effects of various Global memory bandwidth configurations on the test kernels are shown in Figures 5(c), 6(c), 7(c) and the 8 (c). All the kernels except the reduction kernel respond to the increase in the memory bandwidth. The reason about the behavior of reduction kernel is already explained in the last paragraph. The effect of the bandwidth is saturated because of the limited number of channels used to transfer memory requests. The Figures 5(d), 6(d), 7(d) and the 8(d) shows the effects of L2 cache configurations. The 2D-convolution kernel only show negligible effect of L2 cache same as the reduction kernel. But here, the reason for this behavior of convolution kernel is that it uses only a small filter matrix (5×5) which gives only a little reuse as compared to the data set size.

The rest of the results follow almost the same or the similar reasoning for their performance behavior as explained in the above two paragraphs. During our evaluations, we also tested L1 cache and the replacement policies. However, only the usage of L1 cache gives some performance benefits and in some cases shows even a little degradation in the results.



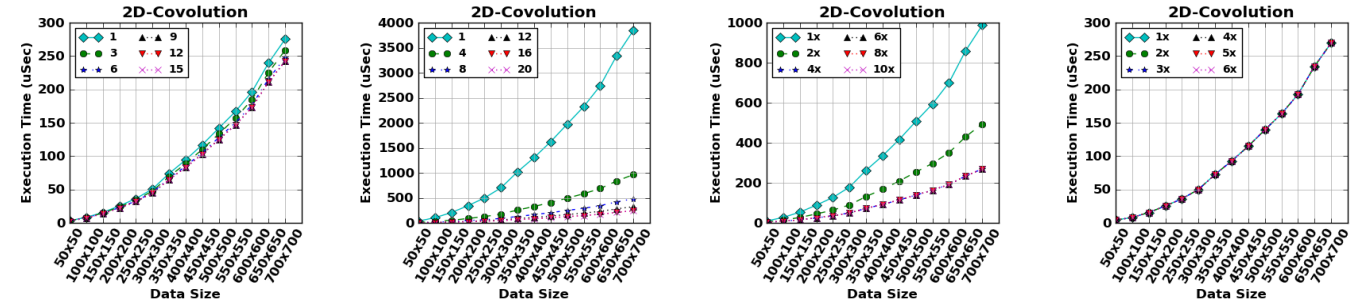
(a) Channel Configuration (b) Number of SM (c) Memory Bandwidth (d) L2 Cache

Fig. 5. Matrix multiplication Kernel (No shared memory)



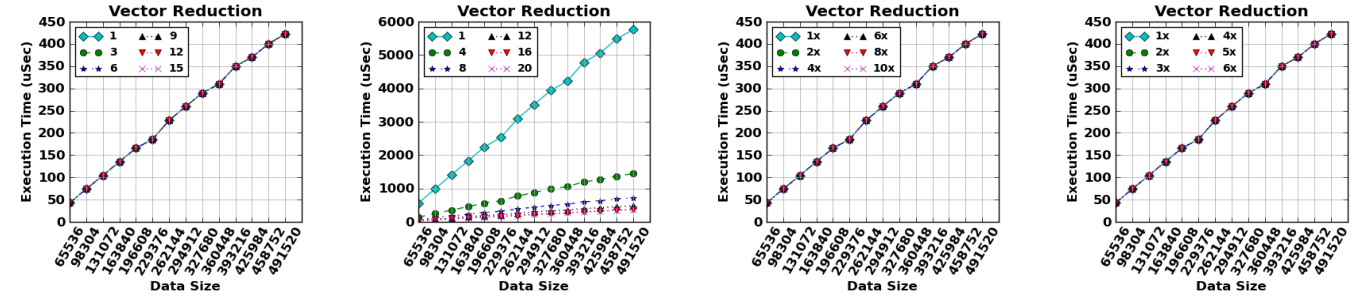
(a) Channel Configuration (b) Number of SM (c) Memory Bandwidth (d) L2 Cache

Fig. 6. 3D-Stencil Kernel using shared memory



(a) Channel Configuration (b) Number of SM (c) Memory Bandwidth (d) L2 Cache

Fig. 7. 2D-Convolution Kernel



(a) Channel Configuration (b) Number of SM (c) Memory Bandwidth (d) L2 Cache

Fig. 8. Vector Reduction using shared memory and multiple Invocations of the device

VI. Related Work

During the last four decades an enormous size of research has been carried-out in the design space explorations for computer architectures. However, most of these research efforts remain focused toward enhancing the the general purpose computing architectures. These efforts enjoyed success sto-

ries in the form of pipelining, branch predictions, hierarchical caches, out-of-order-executions, architectural support for multi-threading etc. A large number of design exploration environments like SimpleScalar [10], Simics [11], PTLsim [12], M5 [13], TaskSim & Cyclesim [14] etc, are available for research on general purpose processor architectures. However, the streaming architectures like GPUs are lacking of similar level of support from the simulation infrastructures. No-doubt, there exist some good efforts in the development of GPU simulation environments like Barra [15] and GpuOcelot [16] but still most of the efforts are either limited in their applicability for a specific GPU architecture or require the presence of a physical GPU and its related software toolchains. Our proposed framework *SArcs* gives an opportunity to researchers in computer architecture to be able to explore various possibilities to improve on top of current GPU designs.

The GPU architectural history is not very old as compared to the CPU generations. The first GPU developed by NVIDIA was just dated back in 1999. However, the base architecture for the current GPU design was incorporated in G80 series in 2006 and the first GPU (GT200) with CUDA cores was introduced just three years back in 2008 [9]. It is understandable that in this short period of time the developers might be only trying to stabilize their GPU products and the researchers to start understanding and exploring the hidden micro-architectural details of GPUs. Therefore, it makes sense that the research and development tools, specially, the simulation infrastructures for these devices are countable on the fingertips.

The previous contributions related to GPU simulation and performance analysis mostly adopt the analytical methods but there are also examples of application methods. In analytical methods, two interesting contributions are from Hong et al. Initially they proposed a GPU performance model [17] and later extended it as integrated performance and power model for GPUs [18]. *CuMAPz* is a CUDA program analysis tool proposed by Y. Kim and A. Shrivastava [19]. This tool analyze memory access patterns in CUDA. The proposal helps in tuning the GPGPU applications for better performance by allowing its users to compare the memory (shared and global memories) performance for an application designed with various versions of memory access patterns. Since the *CuMAPz* approach is compile-time analysis . Therefore, It can not Handle any information that can only be determined during run-time, such as dynamically allocated shared memory, indirect array accesses, etc. In 2009, A. Bakhoda et al. proposed a detailed GPU simulator [20] for analysing the CUDA Workloads. The simulator runs NVIDIA's parallel thread execution (PTX) virtual instruction set for CUDA compiled applications. The simulator design framework sounds interesting. However, we did not find any follow-up work to this one.

A GPU adaptive performance modelling tool [21] presented by Baghsorkhi et al. This tool uses a compiler-based approach to run a static analysis called *symbolic evaluations* on the program structure. This analysis determines the effects of the structural conditions and complex memory access expressions on the performance of a GPU kernel. Moreover, this tool provides a mechanism to adjust latencies according to the kernel inputs and/or data access patterns.

GROPHECY [22] takes as input a modified CPU code called *Code Skeleton* from the user to tune it for a GPU based implementation. In *Code Skeletonization* the user abstracts the CPU code's parallelism, computational intensity, and data accesses. The *GROPHECY* tool apply different set of parameters and optimizations in an automated way to propose one of the best code structure for GPU based implementation.

GpuOcelot [16] is an interesting compilation framework for heterogeneous systems. *Ocelot* provides various back-end targets for CUDA programs and analysis modules for the PTX instruction set. We, in addition to the current standalone framework of *SArcs*, plan to use *GpuOcelot* at the front-end of *SArcs* to enable a provision to also generate traces directly from the PTX code.

MacSim [23] is a trace driven simulation tool chain for heterogeneous architectures. However, we were able to find only a little information related to this tool for the GPU specific support. This tool intends to use Ocelot [16] for handling the PTX based trace generation to be used in their simulation framework. MacSim idea is to convert the program trace to RISC style *uops* and those *uops* will be simulated. However, the design of *SArcs* controls the trace generation process. The generated trace is either from a CPU code or a PTX based GPU code, *SArcs* can provide capability to directly map and simulate the real trace for a GPU generation.

VII. Conclusions

The new architectural explorations are not possible without accurate design space exploration tools. GPUs – even now a part of many supercomputing systems – still lack of non-commercial architectural simulation infrastructures. In this work, we show that the architectural model of GPU like streaming devices can be effectively transformed to a simulator infrastructure under our proposed *SArcs* framework. *SArcs* framework provides an automated interface to simulate different target architectural configurations for a GPU based proposal.

We show the potential of our proposed framework with example explorations for the design of future GPU devices. Results show that the configurations of the computational resources for the current Fermi GPU device would still be enough for the newer designs. The current generation of GPUs can deliver higher performance with further improvements in the design of GPU's global memory for higher bandwidth and efficiency. The results motivates for further research and explorations in this direction. As a future work, we consider that the GPU like streaming architectures can be improved for their performance, efficiency and lesser pressure on the requirements of external bandwidth by using a GPU front-end to accommodate more efficient data organizations as compared to the standard cache hierarchy.

REFERENCES

[1] M. Shafiq, M. Pericas, N. Navarro, and E. Ayguade, "BSArc: Blacksmith Streaming Architecture for HPC Accelerators," *ACM International Conference on Computing Frontiers*, May 2012.

[2] "Top 500 Supercomputer Sites," June 2011. [Online]. Available: http://top500.org/lists/2011/11

[3] G. Caragea, F. Keceli, A. Tzannes, and U. Vishkin, "General-Purpose vs. GPU: Comparison of Many-Cores on Irregular Workloads," *HotPar, Berkeley, CA*, June 2010. [Online]. Available: http://www.usenix.org/event/hotpar10/final_posters/Caragea.pdf

[4] S. Asano, T. Maruyama, and Y. Yamaguchi, "Performance Comparison of FPGA, GPU and CPU in Image processing," *IEEE FPL*, September 2009.

[5] D. B. Kirk and W. mei W. Hwu, "Programming Massively Parallel Processors: A Hands-on Approach (Chapter-2)," *Published by Elsevier Inc*, 2010.

[6] "CUDA Programming Model." [Online]. Available: http://developer.nvidia.com/category/zone/cuda-zone

[7] "Open Computing Language (OpenCL)." [Online]. Available: http://developer.nvidia.com/opencl

[8] "Pin - A Dynamic Binary Instrumentation Tool." [Online]. Available: http://www.pintool.org/

[9] NVIDIA, "Whitepaper : NVIDIAs Next Generation CUDA Compute Architecture," 2009.

[10] "SimpleScalar:." [Online]. Available: http://pages.cs.wisc.edu/ mscalar/simplescalar.html

[11] "simics: ." [Online]. Available: https://www.simics.net/

[12] "PTLsim: ." [Online]. Available: http://www.ptlsim.org/

[13] "M5: ." [Online]. Available: http://www.m5sim.org/Main_Page

[14] "TaskSim and Cyclesim: ." [Online]. Available: http://pcsostres.ac.upc.edu/cyclesim/doku.php/tasksim:start

[15] "Barra - NVIDIA G80 GPU Functional Simulator ." [Online]. Available: http://gpgpu.univ-perp.fr/index.php/Barra

[16] "GpuOcelot: A dynamic compilation framework for PTX." [Online]. Available: http://code.google.com/p/gpuocelot/

[17] S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," *SIGARCH Comput. Archit. News*, June 2009.

[18] Sunpyo Hong and Hyesoon Kim, "An integrated GPU power and performance model," *ACM ISCA 10*, June 2010.

[19] Y. Kim and A. Shrivastava, "CuMAPz: A tool to analyze memory access patterns in CUDA," *ACM/IEEE DAC 2011*, June 2011.

[20] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," *IEEE ISPASS 09*, April 2009.

[21] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W. mei W. Hwu, "An Adaptive Performance Modeling Tool for GPU Architectures," *ACM PPoPP10*, January 2010.

[22] J. Meng, V. A. Morozov, K. Kumaran, V. Vishwanath, and T. D. Uram, "GROPHECY: GPU Performance Projection from CPU Code Skeletons," *ACM/IEEE SC11*, November 2011.

[23] H. Kim, "GPU Architecture Research with MacSim ," 2010. [Online]. Available: http://comparch.gatech.edu/hparch/nvidia_kickoff_2010_kim.pdf