GPUX: Tool-chain for porting Legacy applications to GPGPU

A. Zaidi* (alixedi@gmail.com), A. Raza* (engr.aliraza@hotmail.com) and M. Umair* (munairk@live.com)

* NESCOM, Karachi, Pakistan

Abstract—A large number of scientific applications have been shown to accelerate using GPGPU. However, porting legacy scientific applications to GPGPU is a challenging task. An important part of this challenge is to identify vectorizable loops in the program as well as try and vectorize non-vectorizable loops. In this paper we present GPUX: A tool-chain for porting legacy applications to GPGPU by analyzing program traces. GPUX identifies vectorizable loops and diagnose loop-carried dependencies in non-vectorizable loops thereby helping programmers to vectorize them. We demonstrate the workings of GPUX using a representative set of common loop constructs. GPUX correctly identifies all the loops, loop-carried dependencies and their properties including dependence types, dependence distances as well as the source line numbers and variable names.

1 INTRODUCTION

The Nebulae is a supercomputer installed at the National Supercomputing Centre in Shenzhen, China. It is based on a Dawning TC3600 Blade system with Intel X5650 processors and NVIDIA Tesla C2050 GPUs (Graphic Processing Units) [1]. The June 2010 Top500 list of supercomputers rated the second fastest in Linpack performance at 1.27 PFlop/s - The highest rank achieved by a GPU-accelerated system till date [1].

The key to the Nebulae is the NVIDIA Tesla C2050 GPU. It ships with as many as 448 computing cores and a theoretical peak performance of 1 Teraflop [2] which is representative of modern GPUs. The increasing gap between the peak performance of microprocessors and GPUs has fueled research in GPGPU (General-purpose Processing on Graphic Processing Units). While the GPUs were originally designed for rendering realistic 3D graphics for computer games, the GPGPU phenomenon has found applications in areas such as Astro-Physics [4, 5]. Computational Fluid Dynamics [5, 6], Numerical Algorithms [7, 8], Cryptography [9, 10] etc. Several of these applications have been shown to accelerate by as much as 100X using GPGPU [11].

The introduction of GPGPU-specific APIs such as NVIDIA's CUDA (Compute Unified Device Architecture) [12] goes a long way in making the power of GPGPU accessible to Engineers and Scientists. However, programming a GPU to operate at maximum possible throughput remains a challenging task [13]. The task is further complicated when legacy programs are required to be ported to GPGPU. This porting can be broken down into following steps:

1. Identify vectorizable loops in a program. Loops are vectorizable when either they do not have any loop-carried dependency or have a loop-carried dependency that can be easily mitigated

2. Find a partitioning for the set of vectorizable loops that minimizes communication between CPU and GPU while maximizing speed-up.

3. Extract the loops earmarked for GPU execution to extern functions implemented using a GPGPU API like CUDA while inserting code for handling data transfers in what is left of the original program.

4. Compilation and Execution

In this paper we present GPUX: A tool-chain for porting legacy programs to GPGPU by analyzing program traces. GPUX automates the first step of the porting process i.e. identification of vectorizable loops in a program. It also diagnose loop-carried dependencies in non-vectorizable loops thereby helping programmers to vectorize them.

The rest of this paper comprises of five sections. Section II presents a literature review of similar research efforts. Section III explains the working of GPUX. Section IV shows GPUX in action using a representative set of common loop constructs. Section V lists possible improvements to GPUX and summarizes our contributions.

II RELATED WORK

A number of tools have been presented in literature that identify loops and some of their characteristics by using binary instrumentation [14 and 15]. However, none of these tools provide information about loop-carried dependencies. This information is vital for vectorizing the loops which is necessary for accelerated GPU execution. The work by Kim et al. [16] is an exception. They present Prospector: A dynamic binary instrumentation tool that detects loops and loop-carried dependencies. In addition, it also detects several characteristics of loop-carried dependencies such as dependence type, dependence distance as well as line number and variable name in source code which is very helpful to the programmer in vectorizing the loops. Prospector follows what can be termed as a single-pass approach to the problem i.e. producing dynamic program information (using binary instrumentation) as well as consuming it (detect loops, loop-carried dependencies and their characteristics) at run-time.

The single-pass approach has the advantage of not having to instrument non-loop code for finding loop-carried dependencies. However, according to the 90-10 rule [17], for majority of programs, the non-loop code is responsible for no more than 10% of the total program execution time. In addition, the single-pass approach may result in larger instrumentation overheads due increased complexity incurred by run-time analysis of dynamic program information generated by binary instrumentation.
III GPUX

GPUX follows what can be termed as a two-pass approach as shown in Figure 1. The first pass is for producing dynamic program information (using binary instrumentation) which is then compressed and stored in a program trace file. The second pass consumes this information (detect loops, loop-carried dependencies and their characteristics) post-mortem.

This approach has the benefit of reduced instrumentation overhead due to simpler trace-generation code as well as avoiding the overheads of trace-generation every time the trace is to be analyzed. In addition, GPUX offers the following advantages over its peers:

1. **Compilation:** GPUX uses Pin [18]: An instrumentation framework that is easy-to-use, portable, transparent, efficient, and robust while supporting the IA32, EM64T, Itanium, and ARM architectures.

2. **GPUX's post-mortem program analysis is based on Python [19]: A programming language designed for ease of use. It builds on Networkx [20] which is a powerful Python module comprising functions to not only read and write Graphs in several standard formats (Adjlist, GML, Pajek, Graphviz, VCG etc.) but also hundreds of Graph algorithms from literature (Cliques, Clusters, Cycles, Flows etc.) that are used extensively in program analysis.

3. **GPUX generates the program CFG (Control Flow Graph) from parsing objdump [21]: A program that displays information in object files including structural information (Images, Sections and Functions), debug information (Symbol table and Source line numbers) as well as program assembly. Unlike dynamic program CFG generated from program trace, the CFG obtained by parsing objdump can be guaranteed to be complete i.e. include all possible control-flow alternatives.

The GPUX tool-chain comprises five phases as shown in Figure 1. These phases are described below:

1. **Compilation:** GPUX is a compiler-independent tool. However, it is recommended that the program be compiled with the following options:
   a. Optimizations turned off (-O0 for gcc) to ensure that the program analysis output relates to the program source code.
   b. Debug flag turned on (-g for gcc) so that the program analysis output can be linked to the line number and variable names in the program source code.

2. **Instrumentation:** As a part of the GPUX tool-flow, we have developed a Pin tool that dumps all control flow edges and all memory access encountered during program execution to their respective files. We have deliberately kept this tool very simple in order to avoid incurring instrumentation overheads [22].

3. **Trace Compression:** Program Traces are notorious for their large size. GPUX uses TCGen [23] to overcome this bottleneck. TCGen has been shown to have very good numbers for compression speed as well as compression ratio.

4. **Objdump:** GPUX parses the objdump output to generate program CFGs. This approach has many advantages as described earlier. Following objdump options are used to generate requisite information:
   a. -d: Disassemble object file
   b. -l: Annotate output by source line numbers
   c. -t: Append symbol table to output

5. **Flowgraph:** The final phase of GPUX is called Flowgraphx. It comprises the following steps:
   a. Parse objdump and program trace to generate program CFG (Control Flow Graph) annotated with edge-weights for control-flow edges.
   b. Store the program CFG in Networkx DiGraph data-structure so that it can be manipulated by the Networkx API.
   c. Detect loops and loop-nesting using standard program CFG analysis techniques such as dominator analysis.
   d. Detect loop-carried dependencies by using memory access information from the enumerating the iteration-space of loops and annotating it with dependencies and their characteristics including dependence type, dependence distance as well as line number and variable name in source code. Flowgraph also employ heuristics presented by Kim et al. [16] to identify dependencies incurred by induction variables, reduction variables and temporary variables. These dependencies are easily mitigated when vectorizing the loops and are not considered in further inspection and analysis.
   e. Print hierarchical report of loops and their dependencies. Hierarchical report here means that the report expresses not only loops and their dependencies but also shows nesting information.
Table 1 Program containing common loop constructs with variable loop-carried dependence characteristics

<table>
<thead>
<tr>
<th>S. No.</th>
<th>Source Code</th>
<th>Type</th>
<th>Dist.</th>
<th>Line</th>
<th>Sym</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A[__i] = __i;</td>
<td>None</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>A[__i] = A[__i - 1];</td>
<td>RAW</td>
<td>1</td>
<td>10</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>A[__i] = A[__i - 5];</td>
<td>RAW</td>
<td>5</td>
<td>12</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>A[__i] = A[__i + 1];</td>
<td>WAR</td>
<td>1</td>
<td>14</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td>A[__i] = A[__i + 5];</td>
<td>WAR</td>
<td>5</td>
<td>16</td>
<td>A</td>
</tr>
<tr>
<td>6</td>
<td>A[__i] = A[0];</td>
<td>RAR</td>
<td>1</td>
<td>18</td>
<td>A</td>
</tr>
<tr>
<td>7</td>
<td>A[0] = A[__i%5];</td>
<td>RAR</td>
<td>5</td>
<td>20</td>
<td>A</td>
</tr>
<tr>
<td>8</td>
<td>A[0] = A[__i];</td>
<td>WAW</td>
<td>1</td>
<td>22</td>
<td>A</td>
</tr>
<tr>
<td>9</td>
<td>A[__i%5] = A[__i];</td>
<td>WAW</td>
<td>5</td>
<td>24</td>
<td>A</td>
</tr>
</tbody>
</table>

IV RESULTS

In order to demonstrate the capabilities of GPUX, we analyzed a program containing a representative set of common loop constructs with variable dependence type and dependence distance. Table 1 shows the source code of the program in focus. Figure 2 shows the program CFG generated by GPUX. Table 1 shows the output of GPUX.

Figure 2 CFG of program generated by GPUX – Opacity of elements α Frequency of execution
Cursory inspection of Table 1 containing the program in focus would reveal the following:
1. There are a total of nine inner loops executing 50 times each and one outer loop that executes ten times.
2. The outer loop has no instructions that may incur loop-carried dependencies.
3. The first inner loops at Line #7 has no loop-carried dependency.
4. The second inner loop at Line #9 has a RAW (Read-After-Write) dependence incurred on vector A on Line #10 with a distance of 1.
5. The third inner loop at Line #11 has a RAW (Read-After-Write) dependence incurred on vector A on Line #12 with a distance of 5.
6. The fourth inner loop at Line #13 has a WAR (Write-After-Read) dependence incurred on vector A on Line #14 with a distance of 1.
7. The fifth inner loop at Line #15 has a WAR (Write-After-Read) dependence incurred on vector A on Line #16 with a distance of 5.
8. The sixth inner loop at Line #17 has a RAR (Read-After-Read) dependence incurred on vector A on Line #18 with a distance of 1.
9. The seventh inner loop at Line #19 has a RAR (Read-After-Read) dependence incurred on vector A on Line #20 with a distance of 5.
10. The eighth inner loop at Line #21 has a WAW (Write-After-Write) dependence incurred on vector A on Line #22 with a distance of 1.
11. The ninth inner loop at Line #23 has a WAW (Write-After-Write) dependence incurred on vector A on Line #24 with a distance of 5.

Figure 2 presents the program CFG generated by GP UX which depicts the program control-flow information visually. It is important to note that the CFG has been pruned for illustration purpose. As a result, nodes and edges that are seldom executed are not shown. Finally, it is evident from the summary of diagnosis presented in Table 2, GP UX accurately detects all the loops, loop-carried dependencies and their characteristics.

V CONCLUSION AND FUTURE WORK
We have presented GP UX: A tool-chain for porting legacy programs to GPGPU by analyzing program traces. We presented the methodology adapted as well as results compiled using a program containing a representative set of loop constructs. GP UX identified the loops and their associated loop-carried dependencies, their characteristics including dependence type and dependence distance as well as the source line numbers and variable names. We believe this information is very helpful to the programmer who is vectorizing the loops in order to port legacy applications to GPGPU.

In the future, we plan to generalize the GP UX tool-chain to a generic program analysis framework. The framework will be based on the Networkx module which has extensive set routines for handling Graphs. This set will be targeted to analyze program CFGs by including some extra features and algorithms from program analysis literature. The framework will support tools for specific tasks such as those performed by GP UX. Some additional ideas for tools include a program partitioning tool which will partition the set of identified vectorizable loops to reduce the memory transfers to/from the GPU.

REFERENCES
[1] Top 500 supercomputers based on Linpack performance (http://www.top500.org)
[12] NVIDIA CUDA Programming Guide version 1.1
[19] Python Programming Language (http://www.python.org/)