Ontology-based Knowledge Representation for PORPLE

[CSC766 Final Report]

Feifei Wang
North Carolina State University
fwang12@ncsu.edu

Yue Zhao
North Carolina State University
yzhao30@ncsu.edu

Xipeng Shen
North Carolina State University
xshen5@ncsu.edu

ABSTRACT

Data placement is important for the performance of a GPU (Graphic Processing Unit) program. However, where to place the data is a complex decision for a programmer to make. Among the recent techniques in solving data placement problem, PORPLE is a representative one. PORPLE is a portable data placement engine that uses hardware information (memory systems and processors) described by memory specification language (MSL) and software information (data access patterns) gathered from a compiler called PORPLE-C. Because of the two different representations, it is hard to share common understanding of the program, and reuse information. Providing a more general, uniform and reusable representation can make data replacement decisions more efficient, interoperable and reusable.

In this paper, we apply ontology-based techniques to systematically and formally represent both hardware information and software information used by PORPLE hoping to achieve efficiency, interoperability and reusability. Specifically, we transform the information of GPU memory systems and processors, and the data access patterns gathered by PROPLE-C to ontology which can be used by PORPLE for data replacement.

General Terms

Compiler

Keywords

compiler, ontology, data placement

1. INTRODUCTION

Data placement is essential for the performance of a GPU (Graphic Processing Unit) program [?]. However, where to place the data depends on the hardware information of the GPU and software information of the program and its input. The hardware information of the GPU includes its memory systems and processors, while the software information means the data access patterns associated with the input to the programs. The memory systems of GPUs are becoming increasingly complex. For example, there exists more than eight types of memory (including caches) on the Tesla M2075 GPU. These memories have different size limitations, block sizes, access constraints and etc. Also, the suitable placements depend on the program inputs since different inputs to a program may lead to different data access patterns, and thus require different data placement. As a result, data placement problem is difficult but should be solved.

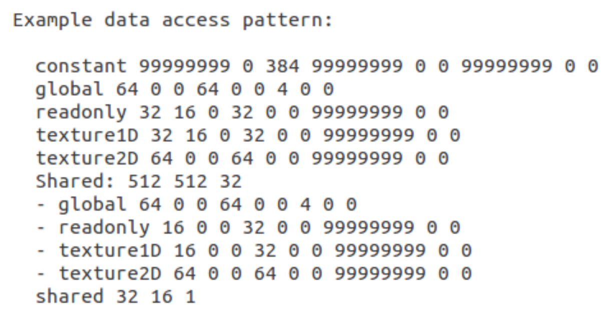
There have been some efforts to address the data placement problem [?, ?, ?, ?]. Among them, PORPLE [?] is a representative one since it considers various types of GPU programs. PORPLE is a portable data placement engine that takes both hardware information (memory systems and processors) and software information (data access patterns) into consideration and use them to make data placement decisions. PORPLE obtains information about memory systems and processors from memory specification language (MSL), and uses the runtime profiling to acquire the data access patterns. In such a sense, PORPLE uses two different types of representations, which makes it hard to share common understanding of the program and reuse information. Providing a more general, uniform and reusable representation can improve the efficiency, interoperability and reusability of PORPLE and potentially some other work.

There exists various techniques to represent knowledge [?]. Recently, ontology-based knowledge bases are becoming increasingly popular. Ontology is a general-purpose modeling for knowledge resources used to define a common vocabulary and a shared understanding explicitly [1, ?]. It has been successfully used to build knowledge bases in many fields [?, ?, ?, ?, ?, ?]. Motivated by their advances, we choose ontology as the representation.

In this paper, we apply ontology-based techniques to systematically and formally represent both hardware information and software information hoping to make PORPLE more efficient, interoperable and reusable. Specifically, we transform the information of GPU memory systems and processors, and the data access patterns gathered to ontology which can be used by PORPLE for data replacement. Note that although our work is applied to PORPLE, it can also be applied to other work.

This paper is organized as follows. In Section 2, we present the motivation of our work. Section 3 illustrates the challenges of the project, our solutions, and lessons we learned. In Section 4 we explain our methodology in detail. Section 5 shows the results. Section 6 concludes this paper and dis-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CSC766 '15 Spring Raleigh, North Carolina USA
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.



```
Example data access pattern:

  constant 99999999 0 384 99999999 0 0 99999999 0 0
  global 64 0 0 64 0 0 4 0 0
  readonly 32 16 0 32 0 0 99999999 0 0
  texture1D 32 16 0 32 0 0 99999999 0 0
  texture2D 64 0 0 64 0 0 99999999 0 0
  Shared: 512 512 32
  - global 64 0 0 64 0 0 4 0 0
  - readonly 16 0 0 32 0 0 99999999 0 0
  - texture1D 16 0 0 32 0 0 99999999 0 0
  - texture2D 64 0 0 64 0 0 99999999 0 0
  shared 32 16 1
```

Figure 2: The data access pattern used by PORPLE

cusses some possible future work.

2. MOTIVATION

This work is mainly motivated by an observation that PORPLE uses two different representations for hardware information and software information. Hardware information is represented by memory specification language (MSL), which is a carefully designed small specification language. It describes the memory systems of a GPU. 1 shows the MSL specification of the Tesla M2075 GPU as an example. The MSL describes the properties of a type of memory and its relations with other pieces of memory in a system. For example, the second line of the file shows the properties of globalMem. The name of the memory is globalMem, its id is 8, it is software manageable, allows read and write accesses, and etc. It also shows the relations between globalMem and L2 is that globalMem has an upper level called L2. Software information, which is data access patterns, is represented by a form defined by Chen et al. [?] 2 presents one data access pattern used by PORPLE. In this file, the second line shows that the total memory access time is 64, L1 cache hit is 0, and L2 cache hit is 0.

Due to the two different representations, it is hard for them to share common understanding of the program and reuse information. In such a sense, we are motivated to choose ontology as a more general, uniform and reusable representation hoping to improve the efficiency, interoperability and reusability of PORPLE and potentially some other work.

This work is also motivated by the success of some previous work of using ontology to systematically represent, reuse, and manipulate software information, hardware information and optimization information. For example, OpenK adapts ontology-based techniques to build open and reusable knowledge bases to do program analysis and optimization in HPC. Sosnovsky et al. [?] and Ganapathi [?] et al. use ontology to teach abstract programming language. Moor et al. [?] and Leenheer et al. [?] focus on community-based evolution of knowledge-intensive systems with ontology. Tang et al. [?] implement a profile compiler that support ontology-based, community-grounded, multilingual, collaborative group decision making by leveraging ontology.

3. CHALLENGE, SOLUTION AND LESSON

Transforming MSL and data access patterns used by PORPLE to ontolgoy is quite straightforward. The main challenge is to understand ontology. In this section, we introduce the challenge to understand ontology, how we overcome them, and the lessons we learned.

3.1 Challenge

Ontologies have three parts: individuals, properties and classes [?]. Individuals are objects in the domain in which we are interested. Properties represent binary relations between two individuals. Objects with similar characteristics are grouped by classes. In this part we use 1 as an example to illustrate the challenges.

The individual part is related easy to understand. We just use

Each object property may have a corresponding inverse property. If some property links individual a to individual b then its inverse property will link individual b to individual a. For example, Figure 4.16 shows the property hasParent and its inverse property hasChild âĂŤ if Matthew hasParent Jean, then because of the inverse property we can infer that Jean hasChild Matthew.

Datatype properties describe relationships between individuals and data values. Object properties describe relationships between two individuals. Datatype properties link an individual to an XML Schema Datatype value or an rdf literal. In other words, they describe relationships between an individual and data values.

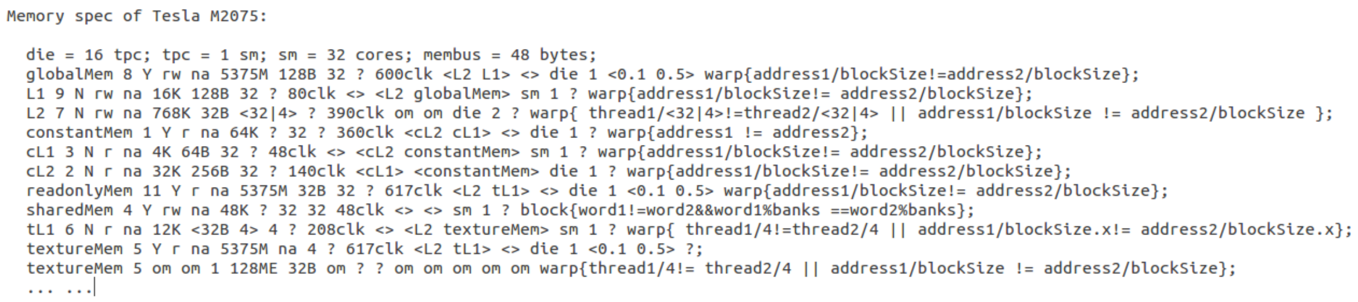
OWL allows us to define individuals and to assert properties about them. Individuals can also be used in class descriptions, namely in hasValue restrictions and enumerated classes which will be explained in section 7.2 and section 7.3 respectively. To create individuals in Prot ÌĄeg ÌĄe 4 the âĂŸIndividuals TabâĂŹ is used. Suppose we wanted to describe the country of origin of various pizza toppings. We would first need to add various âĂŸcountriesâĂŹ to our ontology. Countries, for example, âĂŸEnglandâĂŹ, âĂŸItalyâĂŹ, âĂŸAmericaâĂŹ, are typically thought of as being individuals (it would be incorrect to have a class England for example, as itâĂŹs members would be deemed to be, âĂŸthings that are instances of EnglandâĂŹ). To create this in our Pizza Ontology we

OWL object property characteristics: If a property is functional, for a given individual, there can be at most one individual that is related to the individual via the property. If a property is inverse functional then it means that the inverse property is functional. If a property is transitive, and the property relates individual a to individual b, and also individual b to individual c, then we can infer that individual a is related to individual c via property P. If a property P is symmetric, and the property relates individual a to individual b then individual b is also related to individual a via property P. If a property P is asymmetric, and the property relates individual a to individual b then individual b cannot be related to individual a via property P. Reflexive properties Irreflexive properties

Properties may have a domain and a range specified. Properties link individuals from the domain to individuals from the range.

how to develop an ontology? defining concepts in the domain (classes) arranging the concepts in a hierarchy (subclass-superclass hierarchy) – I don't have this in project Defining which attributes and properties (slots) classes can have and constraints on their values Defining individuals and filling in slot values

what is the domain that the ontology will cover? for what we are going to use the ontology? for what types of ques-



```
Memory spec of Tesla M2075:

  die = 16 tpc; tpc = 1 sm; sm = 32 cores; membus = 48 bytes;
  globalMem 8 Y rw na 5375M 128B 32 ? 600clk <L2 L1> <> die 1 <0.1 0.5> warp{address1/blockSize!=address2/blockSize};
  L1 9 N rw na 16K 128B 32 ? 80clk <> <L2 globalMem> sm 1 ? warp{address1/blockSize!= address2/blockSize};
  L2 7 N rw na 768K 32B <32|4> ? 390clk om om die 2 ? warp{ thread1/<32|4>!=thread2/<32|4> || address1/blockSize != address2/blockSize };
  constantMem 1 Y r na 64K ? 32 ? 360clk <cL2 cL1> <> die 1 ? warp{address1 != address2};
  cL1 3 N r na 4K 64B 32 ? 48clk <> <cL2 constantMem> sm 1 ? warp{address1/blockSize!= address2/blockSize};
  cL2 2 N r na 32K 256B 32 ? 140clk <cL1> <constantMem> die 1 ? warp{address1/blockSize!= address2/blockSize};
  readonlyMem 11 Y r na 5375M 32B 32 ? 617clk <L2 tL1> <> die 1 <0.1 0.5> warp{address1/blockSize!= address2/blockSize};
  sharedMem 4 Y rw na 48K ? 32 32 48clk <> <> sm 1 ? block{word1!=word2&&word1%banks ==word2%banks};
  tL1 6 N r na 12K <32B 4> 4 ? 208clk <> <L2 textureMem> sm 1 ? warp{ thread1/4!=thread2/4 || address1/blockSize.x!= address2/blockSize.x};
  textureMem 5 Y r na 5375M na 4 ? 617clk <L2 tL1> <> die 1 <0.1 0.5> ?;
  textureMem 5 om om 1 128ME 32B om ? ? om om om om om warp{thread1/4!= thread2/4 || address1/blockSize != address2/blockSize};
  ... ...
```

Figure 1: The memory specification of Tesla M2075 in MSL

tions the information in the ontology should provide answers? which characteristics should I consider when choosing ...? what are the terms we need to talk about? what are the properties of these terms? what do we want to say about the terms?

enumerat terms... define classes and the class hierarchy... define properties of classes ... valuetype .. value... The ClassAssertion axiom allows one to state that an individual is an instance of a particular class. The ObjectPropertyAssertion axiom allows one to state that an individual is connected by an object property expression to an individual The DataPropertyAssertion axiom allows one to state that an individual is connected by a data property expression to a literal

Individuals in the OWL 2 syntax represent actual objects from the domain. There are two types of individuals in the syntax of OWL 2. Named individuals are given an explicit name that can be used in any ontology to refer to the same object. Anonymous individuals do not have a global name and are thus local to the ontology they are contained in.

The sentence ”The individual I is an instance of the class C” can be understood as a statement that I is an instance of the UML class Individual, C is an instance of the UML class Class, and there is an instance of the UML class ClassAssertion that connects I with C. This statement can be captured precisely using the structural specification as ClassAssertion(C I).

The individual a:Peter can be used to represent a particular person. It can be used in axioms such as the following one:

ClassAssertion(a:Person a:Peter) Peter is a person.

DataPropertyAssertion(a:hasAge a:Meg ”17”8sd:double) Meg is seventeen years old. The first axiom states that all values of the a:hasAge property must be in the value space of xsd:integer, but the second axiom provides a value for a:hasAge that is equal to the floating-point number 17. Since floating-point numbers are not contained in the value space of xsd:integer, the mentioned ontology is inconsistent.

bjectPropertyAssertion(a:hasBrother a:Chris a:Stewie) Stewie is a brother of Chris.

http://protege.stanford.edu/publications/ontology$_d$*evelopment/ontology*101.*html*

3.2 Solution

3.3 Lessons

4. METHODOLOGY

5. RESULTS

6. CONCLUSION AND FUTURE WORK

Classes can be organized into superclass-subclass hierarchy and they are described or defined by the relationships that individuals participate in.

7. ACKNOWLEDGMENTS

This section is optional; it is a location for you to acknowledge grants, funding, editing assistance and what have you. In the present case, for example, the authors would like to thank Gerald Murray of ACM for his help in codifying this *Author's Guide* and the **.cls** and **.tex** files that it describes.

8. REFERENCES

[1] T. R. Gruber. Toward principles for the design of ontologies used for knowledge sharing? *International journal of human-computer studies*, 43(5):907–928, 1995.

8.1 References

Generated by bibtex from your .bib file. Run latex, then bibtex, then latex twice (to resolve references) to create the .bbl file. Insert that .bbl file into the .tex source file and comment out the command `\thebibliography`.

Table 1: Correctness for All the Questions

Classes and Instances	ClassAssertion()
Class Hierarchies	SubClassOf()
Object Properties	ObjectPropertyAssertion()
Property Hierarchies	SubObjectPropertyOf()
Datatypes	DataPropertyAssertion()