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 [9]. 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 depends 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 [9, 11, 15, 1]. Among them, PORPLE [1] is a representative one because 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 uses 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 [14]. Recently, ontology-based knowledge bases are becoming increasingly popular. Ontology is a general-purpose modeling for knowledge resources. It is used to define a common vocabulary and a shared understanding explicitly [6, 7]. It has been successfully used to build knowledge bases in many fields [3, 2, 13, 10, 12, 4]. 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 designed for 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.

```
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

In Section 4 we explain our implementation in detail. Section 5 shows the results. Section 6 concludes the paper and discusses 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. Figure 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.` [1] Figure 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.` [12] and `Ganapathi et al.` [4] use ontology to teach abstract programming language. `Moor et al.` [3] and `Leenheer et al.` [2] focus on community-based evolution of knowledge-intensive systems with ontology. `Tang et al.` [13] implement a profile compiler that support ontology-based, community-grounded, multilingual, collaborative group decision making by leveraging ontology.

3. CHALLENGES, SOLUTIONS AND LESSONS

Transforming MSL and data access patterns used by PORPLE to ontolgoy is quite straightforward. We have not met many challenges. The main challenge is to understand ontology. In this section, we introduce the challenges we met to understand ontology, how we overcame them, and the lessons we learned.

3.1 Challenges

Ontologies have three parts: individuals, properties and classes [8]. 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 Figure 1 as an example to illustrate the challenges we had to understand ontology.

For the individual part, there are mainly two challenges. The first challenge is to use unique `namedIndividual` for each type of memory system. For example, in the MSL for `Tesla M2075`, although there are four `textureMem` with the same name, we cannot use the same name `textureMem` for all of them. We need to make sure that every `namedIndividual` is unique. The second challenge is that we cannot only create the memory types as `namedIndividual`, e.g. `L1` and `L2`. We must also consider the GPU vendors and types, e.g., `M2075`, `M2075GlobalMemory` and `M2075ConstantMemory`.

The challenge in understanding the property is that we need to consider *inverse property*. Inverse property means that if a property links individual A to individual B then its inverse property should also link individual B to individual A [8]. For example, if `globalMem` has an upper level that is `L2`, then `L2` should have a lower level that is `globalMem`.

For the class part, the challenge is to consider some characteristics that are not so obvious. For example, `L1` is a sub-class of `Cache` is obvious. However, we need to pay attention that `M2075` is a sub-class of `Tesla`, `M2075Processor` is a sub-class of `Processor`, `M2075GlobalMemory` is a sub-class of `GlobalMemory` and `etc.`

3.2 Solutions

Our solution is to use Protégé [5] to generate ontology files, and then learn from those generated files. By learning from Protégé, we learn to define individuals and arrange them into a hierarchy (sub-class hierarchy). We take `Tesla`, `M2075`, `Processor`, `scope`, `GlobalMemory`, `ConstantMemory`, `TextureMemory`, `Cache` and `etc.` into consideration. We also learn to define the properties that should be assigned to each class and give them values.

3.3 Lessons

The lessons we learned are that when creating ontology, we must think comprehensively. For example, we cannot just use the memory types as `namedIndividual`, e.g. `L1` and `L2`. We must also consider the GPU vendors and types, e.g., `M2075`, `M2075GlobalMemory` and `M2075ConstantMemory`. Similarly, when constructing the sub-class hierarchy, we cannot only consider the obvious ones such as `L1` is a sub-class of `Cache`. We must also consider that `M2075` is a sub-class of `Tesla`, and `M2075GlobalMemory` is also a sub-class of `GlobalMemory`. Also when assigning properties to `namedIndividual`, we have to think carefully about whether these properties have some special attributes such as inverse property. For example, if we give `globalMem` an upper level that is `L2`, then we should also give `L2` a lower level that is `globalMem`.

4. IMPLEMENTATION

```
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

Keywords:
address1, address2, index1, index 2, banks, blockSize, warp, block, grid, sm, core, tpc, die, clk, ns, ms, sec, na, om, ?;
// na: not applicable; om: omitted; ?: unknown;
// om and ? can be used in all fields

Operators:
C-like arithmetic and relational operators, and a scope operator {};

Syntax:
- specList ::= processorSpec memSpec*

processorSpec ::= die=Integer tpc; tpc=Integer sm; sm=Integer core; membus=Integer bytes; end-of-line
memSpec ::= name id swmng rw dim size blocksize threadsGrouped banks latency upperlevels lowerLevels shareScope pieces concurrencyFactor serialCondition; end-of-line

- name ::= String
- id ::= Integer
- swmng ::= Y | N *// software manageable or not*
- rw ::= R|W|RW *// allow read or write accesses*
- dim ::= na | Integer *// special for arrays of a particular dimensionality*
- sz ::= Integer[K|M|G|T|ε][E|B] *// E for data elements*
- size ::= sz | <sz sz> | <sz sz sz>
- blockSize ::= sz | <sz sz> | <sz sz sz>
- lat ::= Integer[clk|ns|ms|sec] *// clk for clocks*
- latency ::= lat | <lat lat>
- upperLevels ::= <[id | name]*>
- lowerLevels ::= <id*>
- shareScope ::= core | sm | tpc | die
- concurrencyFactor ::= < Number Number>
- serialCondition ::= scope{RelationalExpr}
- scope ::= warp | block | grid

Figure 3: Syntax of MSL

We have designed two implementations for MSL and data access patterns. We first introduce the implementation for MSL in detail, and then describe the implementation for data access patterns.

4.1 MSL

When transforming MSL to ontology, we first create individuals, then enumerate properties about them, and assert class descriptions about them in the end.

To create individuals, we first need to add various types of memory systems to our ontology. For example, `globalMem`, `constantMem`, `sharedMem`, are regarded as individuals. Note that when we add them, we must make sure that these memory systems is associated with `M2075`. As a result, we make them new names by adding `M2075` as a prefix and make the names become `M2075GlobalMemory`, `M2075ConstantMemory`, `M2075SharedMemory` and `etc.` We create `namedIndividual` for each of the memory systems since they are given an explicit name that can be used in any ontology to refer to the same object. After that, we must also add keywords such as `block`, `core`, `wrap` as namedIndividual.

After that, we enumerate the properties of individuals. We consider the properties as shown by Figure 3, which is used by PORPLE [1]. By parsing the input file, we can get the information we need to create properties. We use two kinds of properties:

- `ObjectPropertyAssertion`. ObjectPropertyAssertion allows one to state that an individual is connected by an object property expression to an individual. In MSL, we can get information about ObjectPropertyAssertion such as `hasBlockSize`, `hasLatency`, `hasLowerLevel` and `etc.`
- `DataPropertyAssertion`. DataPropertyAssertion allows one to state that an individual is connected by a data property expression to a literal. In MSL, we can get information about DataPropertyAssertion such as `CoresPerSM`, `numberofCoresValue`, `threadsPerBlock` and `etc.`

In the end, we define sub-class hierarchy to state that an individual is an instance of a particular class. For example, `M2075ConstantMemory` is a sub-class of `ConstantMemory`, `M2075GlobalMemory` is a sub-class of `GlobalMemory`, `cL1` is a sub-class of `L1_constant` and `etc.`

4.2 Data Access Patterns

Transforming data access patterns to ontology is quite easy since the syntax and semantics of data access patterns is simple. For example, Figure 2 shows an example input file. For this input file, we only need to create `namedIndividual` for the memory types. For properties, we group the numbers following the names into groups of three numbers as designed by `Chen et al.` [1] We only need to consider four properties here:

- *Total memory access time.* Total memory access time is the first number in the group.
- *L1 cache hit.* L1 cache hit is the second number in the group.
- *L2 cache hit.* L2 cache hit is the third number in the group.
- *Off-chip memory access time.* Off-chip memory access time is total memory access time minus L1 cache hit and minus L2 cache hit.

Note that when parsing the input file, we neglect numbers begin with `9` since they are considered not applicable by PORPLE [1].

Also, for data access patterns, they do not have the sub-class hierarchy.

5. RESULTS

Figure 4 shows the results of transforming MSL to ontology, and Figure 5 shows the results of transforming data access patterns to ontology. In this section, we first explain Figure 4, and then explain Figure 5.

5.1 Ontology representation for MSL

In Figure 4, the output result consists of three parts:

- namedIndividual.
- propertyAssertion.
- classAssertion.

For example, `namedIndividual('M2075globalMem')` means that `M2075globalMem` is a particular kind of memory system. It can be used in `propertyAssertion` and `classAssertion`.

There are a lot of `propertyAssertion`. We list a few properties here as some examples of how to illustrate the ouput.

- `propertyAssertion('consistsOf', 'M2075', 'M2075globalMem')` means that `M2075globalMem` is consisted of by `M2075`.
- `propertyAssertion('hasID', 'M2075globalMem', literal(type('http://www.w3.org/2001/XMLSchema#integer', '8')))` means that `M2075globalMem` has an ID which is `8`.
- `propertyAssertion('softwareManageable', 'M2075globalMem', literal(type('http://www.w3.org/2001/XMLSchema#boolean', 'true')))` means that `M2075globalMem` is software manageable.
- `propertyAssertion('accessible', 'M2075globalMem', literal(rw))` means that `M2075globalMem` has read and write access.
- `propertyAssertion('hasSizeValue', 'M2075globalMem', literal('5375M'))` has a size of `128B`.
- `propertyAssertion('hasBlockSizeValue', 'M2075globalMem', literal('128B'))` means that the block size of `M2075globalMem` is `128B`.
- `propertyAssertion('threadsGroup', 'M2075globalMem', literal(type('http://www.w3.org/2001/XMLSchema#integer', '32')))` means that `M2075globalMem` has threadGroup of `32`.
- `propertyAssertion('hasLatencyValue', 'M2075globalMem', literal('600clk'))` means the latency value of `M2075globalMem` is `600clk`.
- `propertyAssertion('hasUpperLevel', 'M2075globalMem', 'L2')` means that `M2075globalMem` has an upper level that is `L2`.

```
...
namedIndividual('M2075globalMem').
...
propertyAssertion('consistsOf','M2075','M2075globalMem').
propertyAssertion('hasID','M2075globalMem',literal(type('http://www.w3.org/2001/XMLSchema#integer','8'))).
propertyAssertion('softwareManageable','M2075globalMem',literal(type('http://www.w3.org/2001/XMLSchema#boolean','tr
propertyAssertion('accessible','M2075globalMem',literal(rw)).
propertyAssertion('hasSizeValue','M2075globalMem',literal('5375M')).
propertyAssertion('hasBlockSizeValue','M2075globalMem',literal('128B')).
propertyAssertion('threadsGroup','M2075globalMem',literal(type('http://www.w3.org/2001/XMLSchema#integer','32'))).
propertyAssertion('hasLatencyValue','M2075globalMem',literal('600clk')).
propertyAssertion('hasUpperLevel','M2075globalMem','L2').
propertyAssertion('shareScope','M2075globalMem','die').
propertyAssertion('pieces','M2075globalMem',literal(type('http://www.w3.org/2001/XMLSchema#integer','1'))).
propertyAssertion('concurrencyFactor','M2075globalMem',literal('<0.1 0.5>')).
...
propertyAssertion('hasLowerLevel','L2','M2075globalMem').
classAssertion('GlobalMemory','M2075globalMem').
...
```

Figure 4: Example ontology representation for MSL

```
...
namedIndividual('global').
...
propertyAssertion('MemoryAccessTime','global',literal(type('http://www.w3.org/2001/XMLSchema#integer','64'))).
propertyAssertion('L1CacheHit','global',literal(type('http://www.w3.org/2001/XMLSchema#integer','0'))).
propertyAssertion('L2CacheHit','global',literal(type('http://www.w3.org/2001/XMLSchema#integer','0'))).
propertyAssertion('OffChipMemoryAccessTime','global',literal(type('http://www.w3.org/2001/XMLSchema#integer','64'))).
propertyAssertion('MemoryAccessTime','global',literal(type('http://www.w3.org/2001/XMLSchema#integer','64'))).
propertyAssertion('L1CacheHit','global',literal(type('http://www.w3.org/2001/XMLSchema#integer','0'))).
propertyAssertion('L2CacheHit','global',literal(type('http://www.w3.org/2001/XMLSchema#integer','0'))).
propertyAssertion('OffChipMemoryAccessTime','global',literal(type('http://www.w3.org/2001/XMLSchema#integer','64'))).
propertyAssertion('MemoryAccessTime','global',literal(type('http://www.w3.org/2001/XMLSchema#integer','4'))).
propertyAssertion('L1CacheHit','global',literal(type('http://www.w3.org/2001/XMLSchema#integer','0'))).
propertyAssertion('L2CacheHit','global',literal(type('http://www.w3.org/2001/XMLSchema#integer','0'))).
propertyAssertion('OffChipMemoryAccessTime','global',literal(type('http://www.w3.org/2001/XMLSchema#integer','4'))).
...
```

Figure 5: Example ontology representation for data access patterns

Since `M2075globalMem` is a kind of GlobalMemory, we need to construct a sub-class hierarchy that by specifying `classAssertion('GlobalMemory','M2075globalMem')`.

5.2 Ontology representation for Data Access Patterns

In Figure 5, there are only two parts:

- namedIndividual.
- propertyAssertion.

For example, `namedIndividual('global')` means that this access pattern is for global memory system. It is also used in `propertyAssertion` and `classAssertion`.

There are only four kinds of properties for data access patterns.

- `propertyAssertion('MemoryAccessTime', 'global', literal(type('http://www.w3.org/2001/XMLSchema#integer', '64')))` means that global memory access time is `64`.
- `propertyAssertion('L1CacheHit', 'global', literal(type('http://www.w3.org/2001/XMLSchema#integer', '0')))` means that L1 cache hit is `0`.
- `propertyAssertion('L2CacheHit', 'global', literal(type('http://www.w3.org/2001/XMLSchema#integer', '0')))` means that L2 cache hit is `0` too.
- `propertyAssertion('OffChipMemoryAccessTime','global', literal(type('http://www.w3.org/2001/XMLSchema#integer', '64')))` means that the off-chip memory access time is `64`.

6. CONCLUSION AND FUTURE WORK

Data placement has a great influence on GPU programs, which makes data placement problem an important issue for GPU program performance. Current techniques to solve data placement problem use different representations of information, e.g., PORPLE, which is hard to share common understanding of the program, and reuse information.

We provide a more general, uniform and reusable representation by using ontology-based techniques to make data replacement decisions more efficient, interoperable and reusable. In our work, 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.

Although our work is designed for PORPLE, it can be applied to other work as well. In the future, we may want to transform more memory related representatiions to ontology to achieve a more general information sharing and reuse.

7. ACKNOWLEDGMENTS

Dr. Xipeng Shen helped me to choose the topic for the final project of CSC766. Yue Zhao and Guoyang Chen offered me a lot of help to get start with this project. The OpenK project developed by Chunhua Liao et al. helped enhance the presentation of the project.

8. REFERENCES

[1] G. Chen, B. Wu, D. Li, and X. Shen. Porple: An extensible optimizer for portable data placement on gpu. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 88–100. IEEE, 2014.

[2] P. De Leenheer and R. Meersman. Towards community-based evolution of knowledge-intensive systems. In *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS*, pages 989–1006. Springer, 2007.

[3] A. De Moor, P. De Leenheer, and R. Meersman. Dogma-mess: A meaning evolution support system for interorganizational ontology engineering. In *Conceptual structures: Inspiration and application*, pages 189–202. Springer, 2006.

[4] G. Ganapathi, R. Lourdusamy, and V. Rajaram. Towards ontology development for teaching programming language. In *World Congress on Engineering*, 2011.

[5] J. H. Gennari, M. A. Musen, R. W. Fergerson, W. E. Grosso, M. Crubézy, H. Eriksson, N. F. Noy, and S. W. Tu. The evolution of protégé: an environment for knowledge-based systems development. *International Journal of Human-computer studies*, 58(1):89–123, 2003.

[6] 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.

[7] N. Guarino. Formal ontology, conceptual analysis and knowledge representation. *International journal of human-computer studies*, 43(5):625–640, 1995.

[8] M. Horridge, H. Knublauch, A. Rector, R. Stevens, and C. Wroe. A practical guide to building owl ontologies using the protégé-owl plugin and co-ode tools edition 1.0. *University of Manchester*, 2004.

[9] B. Jang, D. Schaa, P. Mistry, and D. Kaeli. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *Parallel and Distributed Systems, IEEE Transactions on*, 22(1):105–118, 2011.

[10] A. S. Kleshchev. How can ontologies contribute to software development? In *Knowledge Processing and Data Analysis*, pages 121–135. Springer, 2011.

[11] W. Ma and G. Agrawal. An integer programming framework for optimizing shared memory use on gpus. In *High Performance Computing (HiPC), 2010 International Conference on*, pages 1–10. IEEE, 2010.

[12] S. Sosnovsky and T. Gavrilova. Development of educational ontology for c-programming. 2006.

[13] Y. Tang, S. Christiaens, K. Kerremans, and R. Meersman. Profile compiler: Ontology-based, community-grounded, multilingual online services to support collaborative decision making. In *Research Challenges in Information Science, 2008. RCIS 2008. Second International Conference on*, pages 279–288. IEEE, 2008.

[14] F. Van Harmelen, V. Lifschitz, and B. Porter. *Handbook of knowledge representation*, volume 1. Elsevier, 2008.

[15] B. Wang, B. Wu, D. Li, X. Shen, W. Yu, Y. Jiao, and

J. S. Vetter. Exploring hybrid memory for gpu energy efficiency through software-hardware co-design. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 93–102. IEEE Press, 2013.