International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 1 Issue 8, October - 2012

Aspect Oriented Programming On GPGPU

V.Srivani
Dept. of CSE,
Vasavi College of Engineering, Hyderabad,
AndhraPradesh-500031, INDIA

B.Syamala
Dept. of CSE,
Vasavi College of Engineering, Hyderabad,
AndhraPradesh-500031, INDIA

Abstract

The move to multi-core CPUs which are accelerated with heterogeneous processing units (e.g., GPU) at every workstation is significantly impacting software programmers to have their programs to be properly parallelized, so that they can utilize the advantage as well as full power of these processing architectures. In comparison with previous processing units, this multi-core heterogeneous parallel device requires a radical change in the way that software is develop and maintained. Portability is required at code and performance level. Power-conscious computations gain is also of stronger importance. Due to parallel processing of languages at source code level certain domain-specific issues and issues related to efficient execution on specific platforms occurs which lead to problems like difficulty to adapt the application to different platforms and/or running conditions, low code reuse across platforms, an invasive approach to parallelize code, and no independent development. Here we propose an invasive approach to parallelize code by developing methodologies and tools to promote a better separation of concerns (i.e., by separating domain specific from platform specific concerns). This approach is based on Aspect-Oriented Programming, to un-clutter the code and to improve programmability of the heterogeneous parallel systems. An aspect-weaving compiler is then used to combine the core OO program with these aspects to generate parallelized programs. This approach modularizes concerns that are hard to manage using conventional programming frameworks as a result it simplifies the programming of accelerator-based heterogeneous parallel systems.

1. Introduction

1.1 AOP

Many programming objectives in both Object oriented programming and procedural techniques are unable to implement the important design decisions. Here the design decisions are been forced to be scattered all over the code, that results in "tangled "code which makes the development and maintenance difficult. The properties of design decisions are addressed as aspects. Aspects are hard to capture because the cross-cut the systems basic functionality. Aspect Oriented Programming is a new technology which is developed for separation of cross cutting concerns that are usually hard to do in object oriented programming.

1.2 Cross-cutting concerns

To build a complex systems component-based development is a widely used approach. Most of the businesses today cannot run without the help of software systems to conduct business operations. But ,software systems are complex and while designing such complex systems ,it is not possible to consider everything and solve everything at once. So the natural way to solve the problem is to break the problem into smaller parts and solve them one by one. This is why we have components. Each component has a specific role to play, has specific responsibilities and purposes. Later these components of various kinds are assembled to form the complete system.

Components are useful and important because they represent the static structure of a complex system. They are very useful for understanding, designing, implementing, distributing, testing, and configuring the system. The most important part of these components are reusability. They also contain things that change together. They are the most important asset for reuse in practice. Components contain things that change together. They keep concerns about a kind of object or an abstraction of real-world phenomena separate.

The type of logging required in our example is a cross-cutting concern. It cannot be isolated or encapsulated in one or two specific classes; the logging involves changes in many places across the system. Our desire is to keep the system maintainable, and therefore we want our logging approach to keep the code as clean and simple as possible. We would also like to avoid significant structural changes to the system's architecture. So how do we implement a cross-cutting concern such as logging? We could refactor all of the code by creating a logging class and performing the appropriate insertions. However, in a large system, this would be a time-consuming, error-prone job.

AOP is designed to handle cross-cutting concerns by providing a mechanism, the aspect, for expressing these concerns and automatically incorporating them into a system. AOP does not replace existing programming paradigms and languages; instead, it works with them to improve their expressiveness and utility. It enhances our ability to express the separation of concerns necessary for a well-designed, maintainable software system. Some concerns are appropriately expressed as encapsulated objects, or components. Others are best expressed as cross-cutting concerns.

1.3 Cross cut concern Design

To manage the complexity of the software systems, separation of concerns is used as a conceptual tool. A number of issues both fundamental (what is an aspect, what is a concern, which concerns are separable, which aspects are composable) as well as technical (how to use a particular technique to solve a particular problem) have been identified when using Separation of concerns techniques. "Separation of concerns" is a general problem-solving idiom that enables us to break the complexity of a problem into loosely-coupled, easier to solve, sub-problems. It consists of solving a problem by addressing its constraints, first separately, and then combining the partial solutions with the expectation that, they be composable, and the resulting solution is nearly optimal. Concern is any matter of interest in a software system. The concerns that are treated separately should be independent enough (i.e., they don't interfere with each other) to yield satisfactory results. Moreover solutions of problem solving activity should be composable.

Given two requirements, under what conditions can they be "developed" separately, and can their realizations (aspects) be composed at will. The answer to this question will help determine the domain or operating range of the development homomorphism We refer to this problem as the composability of requirement realizations.

Given a realization that addresses several concerns, under what conditions can that realization be untangled into separable aspects, each of which addressing a subset of concerns. The answer to this question may help us assess which systems may be reengineered in such a way that different concerns are addressed in separate—and readily reusable—aspects. We refer to this problem as the separability of requirement realizations.

There are two kinds of requirements in designing a problem, empirical requirements and conceptual requirements. Former specify externally observable or empirically determinable qualities that are desired of the artifacts, and later specify adherence to a particular style. For the software externally observable qualities desired in artifact is based on functionality and run time behavior. These requirements specify an input/output relationship and performance, distribution on the underlying machine conceptual requirements dealing with things such as modularity, reusability, choice of programming language, adherence to specific programming style, etc. This can be achieved using AOSD.

Aspect oriented software development (AOSD) provides the mechanism for encapsulating cross cutting concerns using aspects.

- Abstract model of aspect oriented systems by factoring out three core components:

- Join Point Model: The join point model defines the join points available for adaption in a specific system.
- Pointcut Language: The pointcut language is the query language to select a subset of the join points defined by the join point model.
- Adaption Mechanism: The adaption mechanism provides means to add or modify functionality at selected join points.

1.4 Join point

A point during the execution of a program, such as the execution of a method or the handling of an exception. Aspect: A modularization of a concern that cuts across multiple objects. Transaction management is a good example of a crosscutting concern in J2EE applications. In Spring AOP, aspects are implemented using regular classes (the schema-based approach)
The term join point is defined in as a "principled point in the execution of a program". A static join point can be characterized as a location in the program's source code. join points described as systematic loc of a program or elements of a program. The characteristic of static join points is that the selection and adaption of a certain element only depends on selection criteria referring to the application's static structure. every time the corresponding source code elements is reached/executed at runtime the same adaptation is performed. Dynamic join points do not correspond directly to elements in the application source, but may have an associated source code element.

While static join points address the locations in source code available for an aspect, each static join point can be reached a multitude of times during program execution. A dynamic join point defined as a single hit of a static join point during program execution. The characteristic feature for dynamic join points is that conditions that need to be evaluated at runtime and that check whether or not the join point should be adapted are implicitly expressed in the point cut language the difference between static and dynamic join points is that the decision whether or not a join point should be adapted depends on runtime information. Consequently, a system that provides dynamic join

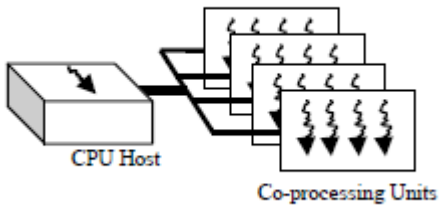
points needs to provide a different kind of pointcut language, as this language needs constructs to refer to runtime values. In contrast to that, a system that provides static join points needs to provide means to reason on an abstraction of the source code.

A pointcut is a set of join points. Whenever the program execution reaches one of the join points described in the pointcut, a piece of code associated with the pointcut (called advice) is executed. This allows a programmer to describe where and when additional code should be executed in addition to an already defined behavior. This permits the addition of aspects to existing software, or the design of software with a clear separation of concerns, wherein the programmer weaves (merges) different aspects into a complete application.

1.5 Multi core processing units-GPU

Mainstream desktop and notebook computers in 2006 started with dual core Processors and in 2008 higher-end desktops and workstations begun to have quad-core processors. In the area of specialized computation, for instance graphics, Graphic Processing Units (GPUs) already contain tens to hundreds of processing cores. CPUs are also evolving with many core direction. For better power efficiency and performance, even processors in ultra-portable devices, such as mobile phones, are heading towards multi core processors. In addition to increase in core count, another trend is the collocation of heterogeneous cores in a single system.

A heterogeneous processing system on a single node can be schematically illustrated in Figure 1



Due to this move there is a huge impact on software programmers because existing sequential programs will not be able to perform better on parallel hardware unless they are properly parallelized. In addition programmers need to take care of the characteristics of the heterogeneity in these processors so that mapping algorithms to these hardware resources are efficient.

1.6 Architecture and Model for Parallel Computation

Parallel processing models can be categorized into those that exploit task parallelism and those that exploit data parallelism.

Another type of processor design, as exemplified by the Graphic Processing Unit (GPU), includes support for coordinating threads of the Single Program Multiple Data (SPMD) model.

2.GPGPU Hardware

Major graphics card manufacturers are now producing equipment that systems designers can use to take advantage of the low cost and high performance of graphics processors. A multicore GPU package specifically designed for general purpose use, in either a 1U rack-mount unit or as a SoM PCI-e card. Each is based on NVIDIA's "CUDA-core", which packs dozens of individual cores together. This density, coupled with a fast interconnect between cores, provides intense performance at a small fraction of the power consumption and cost of multi-core CPU systems with similar performance.

2.1 TESLA GPU 1U RACK-MOUNT SYSTEM

Based on the new NVIDIA CUDA GPU architecture codenamed "Fermi", NVIDIA's Tesla S2050/S2070 computing systems are 1U rack-mountable units that provide four Tesla cores, each with 240 cores, for a total of 960 GPGPU cores. The S2050 provides 12GB of GDDR5 memory, while the S2070 provides 24GB of GDDR5. Designed for datacenters, each S2050/S2070 can provide up to 2.5 TFlops of double-precision performance while consuming 900 watts.

2.2 TESLA GPU SOM

NVIDIA's Tesla C2050/C2070 are card-mounted Systems-on-Module (SOMs) that provide 448 individual GPGPU cores and up to 6GB of GDDR5 memory, all on a PCIe module. Systems integrators can mount as many of these cards into a workstation as there are PCIe slots available. Each C2050/C2070 provides 515 GFlops (double-precision) or over 1TFlop (single-precision) while consuming a maximum of less than 250 watts. One challenge with high-performance computing is finding an operating system that can best take

advantage of heterogeneous multicore environments, particularly if some of those cores are GPUs rather than CPUs. Microsoft has met this challenge by building flexible support for multicore into the Microsoft Windows operating system in both 32- and 64-bit, and including support for the general-purpose GPUs currently available.

Microsoft Windows 7, the latest version of the venerable Windows operating system, supports all currently available GPGPUs. Windows 7 is most appropriate for the Workstation Model, in which GPGPUs are installed as SoMs in one or more PCI Express slots inside a single workstation. This provides a highly economical "supercomputer on a desk" that can have on the order of 1,000 cores (that's four NVIDIA Tesla C1060s, NVIDIA released its Compute Unified Device Architecture (CUDA) in 2007, as an architecture specifically targeting GPUs for both graphics and general-purpose computations.

CUDA supports two programming models: the "language integration" programming model supported by the CUDA C Runtime (CUDART.dll), and the "device API" programming model supported by the CUDA Driver API. Starting with the CUDA Toolkit 3.0, NVIDIA supports mixed use of these two programming models in addition to buffer interoperability (i.e. sharing) between compute and graphics contexts. This enables applications to avoid unnecessary copies between CPU "host" memory and GPU "device" memory.

3. Implementation

3.1 JBoss AOP Language

Joinpoint Model and Pointcut Language

- execution(method or constructor)
- get (field expression)
- set(field expression)
- field(field expression) (read or write or a field)
- all(type expression) (any constructor, method or field of a particular class)
- call(method or constructor)
- within(type expression)
- withincode(method or constructor)
- has(method or constructor)
- hasfield(field expression)

Pointcuts can be named in XML or annotation bindings (see later sections). They can be referenced directly within a pointcut expression.

some.named.pointcut

OR call(void Foo->someMethod())

Dynamic CFlows allow you to define code that will be executed that must be resolved true to trigger positive on a cflow test on an advice binding. The test happens dynamically at runtime and when combined with a pointcut expression allows you to do runtime checks on whether a advice binding should run or not.

3.2 Advice Model and Language

For basic interception, any method that follows the following form can be an advice:

Object methodName(Invocation object) throws Throwable

Invocation objects are the runtime encapsulation of their joinpoint.

Method names can be overloaded for different invocation types, e.g.

Object methodName(MethodInvocation method) throws Throwable

{ .. }

Object methodName(ConstructorInvocation object) throws Throwable

{ .. }

Other types of advice:

- Interface introduction
- Mixin introduction
- Annotation introduction

Also, untyped metadata can be defined within XML files and bound to org.jboss.aop.metadata. Simple Meta Data structures. This XML data can be attached per method, field, class, and constructor. To resolve this type of metadata, the Invocation object provides a method to abstract out where the metadata comes from.

Object get Metadata(Object group, Object attr)

3.3 Aspect Module Model

JBoss AOP is a 100% pure Java framework. All your AOP constructs are defined as pure Java

classes and bound to your application code via XML or by annotations.

The Aspect Class is a plain Java class that can define zero or more advices, pointcuts, and/or mixins.

3.4 Aspect Instantiation Model

The default Scope of an aspect is PER_VM. Another important note is that aspect instances are created on demand and NOT at deployment time. The scope attribute defines when an instance of the aspect should be created. An aspect can be created per vm, per class, per instance, or per join point. JBoss AOP allows for weaving aspects into applications, or preparing specified join points in application code for later decoration with aspect functionality, at load-time. This is achieved using a Java 5 agent that instruments classes (if needed) prior to the class loader inserting them into the running VM. No application class is ever redefined at a later point in time. The basic technique used by JBoss AOP is to replace original join point shadows with invocations to wrappers that are themselves responsible for invoking advice functionality. If multiple aspects apply at a given join point shadow, multiple wrappers wrapping each other are applied, forming a wrapper chain. Access to the AOP infrastructure is provided by an API that is accessible to the programmer and that allows for free assembly, deployment and undeployment of aspects at run-time. Aspects assembled at run-time can however only be attached to join point shadows that have been prepared at load-time.

3.5 Aspect Model

There are two kinds of aspect that can be defined in JBoss AOP, interceptors and aspects. Interceptors are simple aspects that define only one advice method. They have to implement the Interceptor interface. Aspects are far more powerful: they can define an arbitrary number of advice methods, and they do not have to inherit from any specific class or implement any interface. Aspects are first-class entities that can be assembled, passed around, and purposefully deployed and undeployed at run-time. An aspect does not have to pre-exist at load-time in order to be used in a running application. Rather, the JBoss AOP API can be used to set up an aspect specifically whenever it is needed. Complete aspects are represented as instances of the Aspect Definition class, interceptors are self-representing.

3.6 Advice Model

The only kind of advice supported in JBoss AOP are around advice, as JBoss AOP follows a wrapper approach. All advice methods must adhere to a protocol: they must return an Object, and they must accept one parameter of the type Invocation which represents the join point shadow at which the advice applies. Furthermore, they must be declared to throw a Throwable. The Invocation serves as a closure representing the join point. It has a method invokeNext() that, when called, invokes the next wrapper in the wrapper chain applied at a given join point shadow, or the original join point shadow itself. It also exhibits join point context information to advice. Depending on the articular kind of join point shadow it represents, an instance of a respective subclass of Invocation (e.g., for method invocations or field accesses). In the course of assembling aspects at run-time, advice are available as first-class entities. Instances of advice Binding map a pointcut expression to a collection of interceptors or Advisors. An Advisor is a collected representation of all advice applying to a single class.

At the level of execution, all advice are represented as single interceptors that are stored in array mapped to specific join point shadows (see below).

3.7 Pointcut Model

At the user side, point cuts are represented as strings in (mostly) AspectJ syntax. Internally, point cuts are represented as instances of the PointcutExpression class implementing the Pointcut interface. This interface provides various methods to check whether a point cut matches at a given join point. Matching is determined using a visitor hierarchy that applies the point cut to AST nodes of the application. Point cuts are available as Point cut instances at run-time, but they are normally not used as such.

Instead, they are passed to the appropriate API methods as strings and converted to internal representation.

3.8 Functionality

3.8.1 Join Point Shadow Retrieval

The representation of the application the weaver uses to find join point shadows is the application byte code. The byte code is not directly worked on; rather, JBoss AOP uses the Javassist library to instrument classes as they are loaded. During class loading, every class is checked for matching pointcuts. If matches are found, the respective join point shadows are immediately instrumented. Matching is done using the Pointcut instances and their associated visitors described above. The weaving functionality is described in more detail below.

3.8.2 Weaving Approach

JBoss AOP transforms classes using Javassist at load-time, before they are actually passed to theclass loader. No class is ever redefined once it has been loaded; all subsequent weaving steps, e.g.,due to dynamic (un)deployment of aspects, take place through the modification of data structures at the meta-level. For methods, woven code is not inserted via direct bytecode instrumentation, but by passing the according source code strings to the Javassist source code fragment compiler that generates bytecodes from them. Any class subject to decoration with advice is modified to implement the Advised interface. This interface defines a method returning the Advisor responsible for this particular class. The appropriate implementation of this method is also added during class transformation. The Advisor itself is added as a new static member and initialized in a static initialiser block. For each join point shadow, a static member is added to the class that contains an array referencing all of the interceptors applying at the shadow. This array may be null if no interceptors apply. Join point shadows in methods are replaced with code blocks that check whether advice apply at this particular shadow (in fact, it is simply checked whether the interceptors array is null) and that simply execute the original shadow in place if that is not the case. The code that is executed if advice functionality has to be invoked is described below. Method execution shadows lead to the entire method being renamed, and to a new version of the method (under its original name) being introduced that contains the aforementioned infrastructural code.

3.9 Advice invocations

For each join point shadow, a subclass of Invocation is dynamically created during weaving. These classes contain, among others, implementations of invokeNext() specifically suited to the join point shadow they represent. This means that they contain a direct invocation of the original shadow that is performed when the end of the interceptor chain is reached. Advice invocations lead to the creation of an instance of the above Invocation subclass. This instance is populated with the necessary context information (e.g., the target object for a method call) before the invokeNext() method is called on it. This method first checks whether the end of the interceptor chain has been reached and, if so, invokes the original shadow. If there are still interceptors to be executed, the invoke() method is called on the next interceptor in line. These interceptors are dynamically generated subclasses of Interceptor (more specifically, of AbstractAdvice which implements Interceptor) and are specialized to invoke the corresponding advice methods directly. From within the advice, eventually invokeNext() is invoked on the passed Invocation which again leads to the execution of the above functionality. So, if multiple advice apply at a give join point shadow, execution alternates between Invocation.invokeNext(), Interceptor.invoke() and actual advice methods.

4. Conclusion

We are undergoing one of the most important transitions in computing: single-thread performance is flatting out and raw machine performance improvements must be largely derived from thread-level parallelism; parallel programming becomes unavoidable. Due to the efficiency in throughput-oriented hardware such as GPU, parallel computing systems consisting of heterogeneous processing cores become very appealing in power-sensitive environments such as mobile devices and large-scale data centers. These factors introduce additional complexities to the software programming activities. In this paper, we proposed a system, Aspect-Oriented Stream Programming (AOSP), to support programming these heterogeneous CPU + GPU co-processing systems. Using the principles of Aspect-Oriented Programming recognize the aspects such as

memory allocation/de-allocation, data transfers between host device memories concerns. This method reduces clutters in the source code and hence helps to achieve better modularization. To help with programming the very specific aspects arising from heterogeneous co-processing systems, we described a dedicated aspect language for efficiency.

10. References

[1] Eric Bodden and Klaus Havelund. (2010): Aspect-Oriented Race Detection in Java, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 36, NO. 4, pp. 509–527.

[2] MINGLIANG WANG: OBJECT-ORIENTED STREAM PROGRAMMING USING ASPECTS, A dissertation submitted to the Graduate School – New Brunswick Rutgers, The State University of New Jersey.

[3] João L. Sobral, Carlos A. Cunha, Miguel P. Monteiro, Aspect Oriented Pluggable Support for Parallel Computing

[4] S. V. HALSE1 and S. D. PATIL, Aspect-Oriented Programming with AspectJ Programming Approach, Journal of Computer and Mathematical Sciences Vol. 2, Issue 4, 31 August, 2011 Pages (581-692)