Semantic Profiling for Heterogeneous Parallel Languages

TIMOTHY A. K. ZAKIAN, University of Oxford

In today's world parallelism is ubiquitous, and different types of parallel models and processors are becoming increasingly common. Moreover, while our programs become increasingly heterogeneous in the parallelism they employ, user-visible parallelism in our languages become increasingly homogeneous, with the parallelization onus put more-and-more on the compiler or runtime. This increasing divide between what is and what happens in our languages makes being able to determine the costs of running the same program on different types of hardware increasingly important. However, while there are formal means for profiling parallel programs written for the CPU, there are still no means to formally profile and reason about parallel programs that use multiple types of parallel processor. In this paper, we present a theory that allows profiling for both time and space in heterogeneous parallel programs, while also allowing higher-order constructs on the GPU. We go on to show how this work could be useful in the field of compiler optimizations for parallel languages that support CPU and GPU parallelism, and how it could be used to guide optimization and rewrite searches in compilers.

CCS Concepts: •**Theory of computation** → **Operational semantics;** *Program analysis;* •**Computing methodologies** → Parallel programming languages;

Additional Key Words and Phrases: Cost Semantics, Region Based Memory Management, Parallelism, CPU, GPU, Operational Semantics

ACM Reference format:
Timothy A. K. Zakian. 2017. Semantic Profiling for Heterogeneous Parallel Languages. *PACM Progr. Lang.* 1, 1, Article 1 (January 2017), 30 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Determining the space and time usage of the programs that they write is a critical everyday task that programmers perform. The trade-offs between space and time that the programmer considers, and how this informs their programs can have a dramatic impact on performance, and in turn, a seemingly small misjudgment of the resource and time usage of programs can have disastrous consequences. While various theories have been developed to automatically analyze parallel programs for space and time usage, these do not take into account the many different means of computation and parallelism that the modern programmer now has at their disposal.

To see an example of the problems faced by the modern programmer (and compiler writer), consider the following program

```
lambda arrs : (Vec (int, int). parmap (lambda p:(int, int). (fst p * snd p)) arrs
```

where the **parmap** array operation can run in parallel on either the CPU or the GPU. How is the programmer — or even the compiler — supposed to determine where to run this code? Currently, the only ways of determining this are based either on empirically determining what should be done after multiple runs of the program and then using heuristics to determine how the computation should be distributed (Ragan-Kelley et al. 2013), or based purely on programmer know-how, and instinct. But the problems don't simply end here. Indeed, let's imagine the previous program after undergoing a part of the compilation process where we have decided that it should run on the GPU. One of the most common compiler optimization techniques in this case for the GPU are *rewrite rules* (McDonell et al. 2013; Steuwer et al. 2015). But once again we have a problem: while some rewrites can speed up the computation,

2017. 2475-1421/2017/1-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

other ones can have no effect, or even slow the computation down. So how do we determine which rewrite rules are useful? What's more, how do we determine which rewrite rules *don't slow the computation down*? Once again, we must fall back to heuristic methods and instinct (Chakravarty et al. 2011). *Surely there must be a better way.*

Indeed, there is a better way — for homogeneous computations. In the homogeneous case we are almost spoiled for choice, and the theories have been quite well developed with a number of different ways of determining the costs of programs such as amortized resource analysis (Hoffmann et al. 2011), sized types (Vasconcelos 2008), and cost semantics (Blelloch and Harper 2013; Ley-Wild et al. 2009; Sansom and Peyton Jones 1995; Spoonhower et al. 2010). However, as we use more modern domains of computation we quickly run out of options: if we wish to analyze a parallel homogeneous program, cost semantics (Spoonhower et al. 2010), and more empirical methods are the only tools still useful to us; and if we try to add any type of heterogeneity we find that there is no way to successfully profile our program other than through empirical and "best guess" methods. Moreover, as the user-visible line between heterogeneous parts of the computation become increasingly blurred – with some languages even completely eliding the distinction between CPU and GPU computations (Holk et al. 2014) — our lack of ability to formally profile heterogeneous parallel computations is a problem that will only get worse.

Inspired by the work on *cost semantics* (Blelloch and Harper 2013; Spoonhower et al. 2010) this paper presents a theory that will allow programmers to formally profile high-level heterogeneous parallel programs and allow compiler writers to formally reason about, and profile the optimizations that they perform during the compilation process. While developing such a theory offers a number of advantages and opens up many new opportunities for programmers and compiler writers alike, such a theory also presents a number of challenges, and care must be taken to account for a number of different computational patterns that can have significant impacts on the computational complexity of our programs. In particular, a sufficient profiling theory must be able to account for the cost of memory transfers from the CPU to the GPU; for the different costs of parallelism on the CPU and GPU; for the different costs associated with the same computational structures on the CPU versus the GPU; and finally, for how different types of data might have different cost implications on the CPU and the GPU. Moreover, since we are interested in profiling higher-order (functional) heterogeneous parallel languages, we need to be able encode higher-order constructs in the GPU-runnable segments of our programs, and be able to take these into account in our cost system.

In order to allow higher-order constructs on the GPU, we build on the work done in Harlan (Holk 2016; Holk et al. 2014) where a region based memory model (Tofte and Talpin 1997) is used to allow high-level language features to run both on the CPU and the GPU. As we will see in later sections, a region based memory model not only allows us to represent higher-order constructs on the GPU, but also simplifies the process of reasoning about space usage and memory transfer costs in our cost framework. Furthermore, the use of a region based memory management makes this work readily applicable to recent developments in the programming languages community both for low-level languages such as Rust (Mozilla Research 2010), and for distributed computing (Lu et al. 2016; Yang et al. 2015) where grouping data with similar lifetimes together can provide significant benefits in terms of memory transfer, reclamation, and safety guarantees.

In this paper we present the theory and details of a formal profiling framework for a higher-order heterogeneous parallel language based on the operational semantics for the language, and couple this with a region based memory and effect calculus. We make the following contributions.

- We introduce the first framework to formally profile heterogeneous parallel programs;
- We give a region based memory model for heterogeneous parallel languages, with bounded effects and regions;
- We present a cost semantics for both *time* and *space* that takes into account memory transfer times and merges, is parameterized by different scheduling policies, and uses a region based memory management system;
- We show how our cost semantic framework could be used to profile and guide rewrite optimizations for both CPU and GPU.

2 BACKGROUND AND MOTIVATION

2.1 Why Formal Profiling Methods?

There are a variety of different ways of determining the space or time usage of programs. The easiest, and most straightforward of these, is to simply run the program, and from this seek to determine on average how long it takes to run, or how much memory it uses. In this paper, we will classify these types of approaches (broadly) as being 'informal profiling methods', and classify those methods that seek to determine the costs of programs in other ways than running the program (*e.g.*, static or semantic means) as 'formal profiling methods.' In the common case where the programmer has written their program, and simply wants to determine where they should focus their optimization attention, informal methods will in general not only be good enough, but also be the easiest to use. However, as the abstraction level of languages becomes higher and higher, the compiler needs to be able to determine where it should focus its optimization efforts, as well as whether the optimizations that it performs are beneficial to program performance. Using these more informal means to determine the optimizations to perform have demonstrated dramatic improvements in speed and efficiency (Ragan-Kelley et al. 2013), however these more informal methods often take a large amount of time in order to be effective — relying on being able to compile and run the program hundreds, if not thousands of times. This makes the bar for entry to optimize these programs incredibly high from both a time, and resource usage standpoint.

It is here that informal means of profiling fall short; if we are unwilling or unable to devote the amount of time and resources that they require in order to accurately profile our programs, we must currently fall back to heuristic methods in order to guide the optimizations that our compilers perform. These intuition-based methods are exactly where we can, and should make formal profiling methods shine; if we can profile our programs based on their semantic or static descriptions, we can compute *at compile time* whether or not certain transformations performed by the compiler have improved the run time or space efficiency of the program. We can in turn use this extra compile-time information that we glean to iteratively guide search strategies for the compiler, to determine whether or not certain optimizations should be performed (or rolled back), where optimization efforts should be focused, or where to place various computations in the heterogeneous setting. In this way, formal profiling methods open up a number of ways for compiler writers to ask – and answer – runtime-type questions at compile time, and inform their compilation strategies based on these answers.

2.2 Cost Semantics

Cost semantics as we know them today were originally described by Sansom in his dissertation (Sansom and Sansom 1994) and in a paper with Peyton Jones (Sansom and Peyton Jones 1995) as a way to profile higher-order lazy functional languages. The cost-semantic-system was designed to have an abstract — and hence extensible — notion of cost, be syntax directed, and based on the operational semantics of the language. Thus the usual presentation of a cost semantics is through an annotated big-step semantics in which a cost (or costs) are associated with each step. For instance, a run-of-the-mill cost semantics for the untyped λ-calculus where $E; e \Downarrow v; c$ means that the expression e evaluates to the value v with cost c in environment E might look like:

$$\frac{E(x) = v}{E; x \Downarrow v; 1}\ \text{Var} \qquad \frac{}{E; \lambda x.e \Downarrow \lambda x.e; 1}\ \text{Lam} \qquad \frac{E; e_1 \Downarrow \lambda x.e; c_1 \quad E; e_2 \Downarrow v'; c_2 \quad E[x \mapsto v']; e \Downarrow v; c_3}{E; e_1\ e_2 \Downarrow v; c_1 + c_2 + c_3}\ \text{App}$$

The *Var* and *Lam* rules say that looking up a variable in the environment, and reducing a lambda expression have unit cost. The *App* rule says that the cost of applying e_1 to e_2 is the sum of the costs of evaluating e_1 and e_2 to values, plus the cost of evaluating the body of the function in the environment that maps x to the reduced value of e_2. While these rules are remarkably simple for now, as we will see later these semantics will become quite complex.

The syntax-directed nature of the cost semantics is one of the crucial properties that compiler writers can exploit to their benefit, since this allows them to easily reason — formally — about how the optimizations they write affect the cost of the underlying program based upon the syntactic transformations that are performed on the program. However, reasoning about these abstract costs are not very useful unless we can somehow relate

them to more explicit, real-world costs. In particular, a cost semantics needs to be mapped down to an abstract (or physical) machine model. Moreover, the choice of the underlying abstract machine is important, since it often affects the structure of the rules for the cost semantics as well as the structure of the costs themselves.

2.3 Other Cost Theories

Beyond cost semantics based on the operational semantics for the language, there have been a number of other theories developed in order to describe the resource usage and complexity of programs. In particular, there have been a number of non operational-semantics-based description of cost, for example Van Stone (2003) develops a denotational cost semantics by interpreting the programs in certain enriched categories, and through this is able to express costs for higher-order types, and for call-by-value and call-by-name evaluation strategies, in Danner et al. (2015) the denotational approach to measuring complexity in languages is extended to also take into account inductively defined types. Some of the other means that have been used in order to describe the complexity of programs have been using program transformations to convert a source program into equations that compute the original program complexity (Avanzini et al. 2015; Sands 1990). The use of resource-annotated types (Benzinger 2004; Hoffmann et al. 2011) and sized-types (Chin and Khoo 2001; Hughes et al. 1996; Vasconcelos 2008) have also been a popular and useful means for automatically describing and studying the complexity of programs; however, they take a more static approach to the description of costs. Loidl and Hammond (1996) introduce a sized type theory for an eager functional parallel language, specifically looking at being able to infer these sized types, and then using them to determine the cost of evaluating expressions and proving time and space bounds and to use this information to inform the scheduling policies that are used. One other interesting example of parallelism in this area is Gimenez and Moser (2016), who use the inherent parallelism found in interaction nets (Lafont 1990) coupled with sized types to prove complexity results in an extended λ-calculus.

Another means of determining the resource use of languages has been through the use of code instrumentation. In particular the CerCo project worked on creating a way to determine the cost of functions (and of whole programs) via an instrumentation process (see *e.g.*, Amadio et al. (2010) and Amadio and Régis-Gianas (2011)). In particular, their method involves adding labels to the source code such that all loops and function calls (along with a few other things) must go through a label, and verifying the translation process from the labeled source-code to assembly. They then analyze the number of (assembly) instructions executed by each labeled section of code, and then use this to annotate the source code with cost annotations.

2.4 Challenges and Limitations

While we might get the impression from the description of cost semantics above that it is perfect for our purposes, it would be disingenuous to say that things simply fall into place. Indeed, any cost semantics must be accompanied by an abstract machine model and a way of mapping the operational (cost) semantics down to this model. This in particular presents substantial challenges to deal with in the combined CPU/GPU case, since the abstract machine models for this are scarce, and quite difficult to deal with (Fortune and Wyllie 1978; Nakano 2012) , especially in the parallel case. Moreover, due to the heterogeneity of the computation, our machine models must also be heterogeneous – amounting to two separate abstract machine models that are connected via a model of memory transfers. As we will see later on, it is possible to create such a heterogeneous machine model that takes into account the critical and most costly of the aspects of the computation, however, we must at the same time pick our battles and elide some aspects of the underlying machine model in order to present a machine abstraction that is usable. We discuss these trade-offs in more depth and introduce the machine models that we use in Section 5.

While the cost semantic approach may be difficult, we feel that it not only offers the most elegant and easiest way to profile heterogeneous parallel programs, but also opens up the largest number of opportunities to extend this work into more settings. In particular, the fact that various profiling questions can be phrased in terms of questions about the cost graph that is built using the cost semantics not only greatly increases the power of

the theory and tools at our disposal, but also makes the theory more readily extensible to other areas and more amenable to implementation.

3 LANGUAGE DEFINITION AND TYPE SYSTEM

In this section we present the definition of the language that we will be using throughout the rest of this paper and present a region-typing system that statically ensures safety and computational properties about the language.

3.1 Syntax

The core definition of our language which we'll call $\lambda_{\mathcal{G}}$ is given in Figure 1, and is an extended form of the language presented by Fluet and Morrisett (2006) with region abstraction and instantiation forms omitted for brevity. While the language is particularly sparse for a region-based parallel language since we wish it to be a usable intermediate language for a compiler, we have also designed it to be suitably expressive so as to allow programmers to easily and directly express computations within it.

While the majority of the additions to the language are straightforward, the two most subtle – and important forms – in the language that not only give us the ability to run heterogeneous computations, but also express parallelism on them are the **on** and **parmap** forms. In particular, **on** e **at** ρ_ω specifies that the computation e should be run on the GPU and that the result of this computation should be stored in the region ρ that is located on the computational world ω. The other core form in the language that allows us to perform parallel operations is **parmap** $f\ [e_1, \ldots, e_n]$ **at** ρ_ω which specifies that the function f should be applied in parallel to each of the e_i and that the result of this map should be stored in the region ρ on world ω. On the CPU, this form amounts to a normal parallel map call, but on the GPU these coupled with an **on** correspond to GPU kernels, with each $f\ e_i$ (approximately) corresponding to a work item. We will go into more detail on what exactly this entails and how it works in Section 6.

We define a number of metafunctions over the syntax; we define $frv(e)$ and $fv(e)$ to be the set of all free region variables and variables occuring in e respectively. We likewise have a metafunction $|v|$ that calculates the size of heap value v. The definitions of these are standard, with the **letregion** form serving as the binder for region variables.

3.2 Type System

The model of computation that we wish to express, how our computations are distributed, and the hierarchy between computations and memory residing on different devices is a critical aspect of the design of $\lambda_{\mathcal{G}}$. In particular, care must be taken to limit certain operations on the GPU since there are computational restrictions – such as not sparking off sub-kernels – on these devices. With this in mind, we represent the host and device hierarchy in a similar way to Harlan (Holk et al. 2014); we only permit computations and memory to be 'pushed' to the GPU from the CPU, and memory can only be 'pulled' back to the CPU, moreover we limit the creation of new regions of memory to the CPU. In this sense the CPU strictly dominates the GPU in terms of the operations that are permitted to be performed on it.

We encode these device hierarchies and restrictions statically in the language through both the type system and the region system. In particular, we statically disallow bad region accesses (*i.e.*, cross-device region accesses and out-of-scope accesses), invalid computations, and invalid control structures on device worlds by using both a standard region-typing system enriched with "region kinds" that demarcate on which world the given region resides as well as annotate the typing judgement with the current world that the computation is being typechecked on. Through this use of world-annotated typing judgements and world-annotated regions, we can disallow both off-current-world region accesses and invalid device-specific computational structures.

The definitions for our types, contexts, effects, regions stacks, and worlds are given in Figure 1. Our computation worlds are one of either $\mathbb{C}$ (for the CPU) or $\mathbb{G}$ (for the GPU), and our region variables are annotated with the

$$i \in \mathbb{Z} \mid r \in RNames \mid f, x \in Vars \mid \rho_\omega \in RVars \mid o \in \mathbb{N}_{bounded}$$

ℓ	::=	(r, o)	Region Locations
R	::=	$(\mathsf{ap}, s, \mathtt{refcnt}, \{o_1 \mapsto v_1, \ldots, o_n \mapsto v_n\})$	Regions
$\mathcal{S}$	::=	$\cdot \mid \mathcal{S}, r \mapsto R$	Ordered Domain/Region Stack
φ	::=	$\{\rho_1, \ldots, \rho_n\}$	Effect Sets
ω	::=	$\mathbb{C} \mid \mathbb{G}$	Computation Worlds
$\mathcal{T}$	::=	$\{\mathbb{C} \mapsto \mathcal{S}_1, \mathbb{G} \mapsto \mathcal{S}_2\}$	Configuration Tables
τ	::=	$\mathbf{bool} \mid (\mu, \rho)$	Types
μ	::=	$\mathbf{int} \mid \tau_1 \xrightarrow{\varphi} \tau_2 \mid \tau_1 \times \tau_2 \mid \mathbf{Vec}\ \tau$	Boxed Types
Ω	::=	$\{\overline{\rho}\} \mid \rho_\omega, \Omega$	World Contexts
Δ	::=	$\cdot \mid \Delta, \rho_\omega \succeq \varphi$	Region Constraints
Γ	::=	$\cdot \mid \Gamma, x : \tau$	Typing Contexts
e	::=	$i\ \mathbf{at}\ \rho \mid \mathbf{True} \mid \mathbf{False}$	Terms
		$\mid \mathbf{if}\ e\ \mathbf{then}\ e_t\ \mathbf{else}\ e_f \mid x$	
		$\mid \lambda x : \tau.^{\varphi} e\ \mathbf{at}\ \rho \mid e_1\ e_2$	
		$\mid (e_1, e_2)\ \mathbf{at}\ \rho \mid \mathbf{fst}\ e \mid \mathbf{snd}\ e$	
		$\mid \mathbf{letregion}\ \rho\ \mathbf{in}\ e \mid \mathbf{fix}\ f : \tau.u \mid e_1 \| e_2\ \mathbf{at}\ \rho$	
		$\mid [e_1, \ldots, e_n]\ \mathbf{at}\ \rho \mid \mathbf{parmap}\ f\ [e_1, \ldots, e_n]\ \mathbf{at}\ \rho$	
		$\mid \mathbf{on}\ e\ \mathbf{at}\ \rho$	
v	::=	$i \mid b \mid (v_1, v_2) \mid [v_1, \ldots, v_n]$	Heap Values
		$\mid \mathbf{clos}\ (x, e, E) \mid \ell$	

Fig. 1. Syntax of $\lambda_{\mathcal{G}}$

computation world that they reside on. Each region variable in the program is assumed to be unique up-to the world annotation *i.e.*, all region variables on $\mathbb{C}$ are unique. Each region resides in a region stack $\mathcal{S}$, and we keep two region stacks for our program in a configuration table $\mathcal{T}$ – one region stack for $\mathbb{C}$ and another for $\mathbb{G}$. Moreover, we use the naming convention that all regions copied from one world to another retain their names and since we only 'push' regions from the CPU to the GPU, every region variable on the GPU is unique (on the GPU region) and has the same region name as its parent region on the CPU. As we will see later on, this purposeful name shadowing between CPU and GPU region names in the region stacks makes the process of transferring computations from one world to another much simpler. Our types are standard for region-based typing systems: booleans are not heap allocated and hence are not annotated with the region where they reside but all other types are annotated with the region where they reside, and function types are annotated with their set of (latent) effects.

Each latent effect is a set of all current regions that are accessed by the body of the function – both for reading and writing. Typing closures in this way statically ensures that the regions of memory that we access within a given function's body are valid wherever the function may be applied. In many region typing systems, function types can be polymorphic over the set of their latent effects. We do not have effect polymorphism in our language; since we are using a version of the bounded region calculus from Fluet and Morrisett (2006) we can express an equivalent amount of polymorphism at the effect level as we can with effect polymorphism through region abstraction and instantiation with bounding effects.

The typing judgements that we use take the standard form for a region-typing system with the exception that each typing judgment is annotated with the current world of execution of the expression that is being inferred:

$$\Omega; \Delta; \Gamma \vdash^{\omega}_{\text{exp}} e : \tau, \varphi$$

$$\frac{\vdash_{\text{ctxt}} \Omega;\Delta;\Gamma;\varphi \quad \Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} e_1 : \tau_1, \varphi_1 \quad \Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} e_2 : \tau_2, \varphi_2 \quad \Omega \vdash_{\text{place}} \rho_\omega \quad \Omega;\Delta \vdash_{\text{ee}} \varphi \supseteq \varphi_1 \quad \Omega;\Delta \vdash_{\text{er}} \varphi \ni \rho_\omega \quad \Omega;\Delta \vdash_{\text{ee}} \varphi \supseteq \varphi_2}{\Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} e_1 \| e_2 \ \texttt{at}\ \rho_\omega : (\tau_1 \times \tau_2, \rho_\omega), \varphi}$$

$$\frac{\Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} e : (\tau_1 \xrightarrow{\varphi'} \tau_2, \rho'_\omega), \varphi \quad \Omega \vdash_{\text{place}} \rho'_\omega \quad \Omega;\Delta \vdash_{\text{ee}} \varphi \supseteq \varphi' \quad \Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} e_i : \tau_1, \varphi \quad \Omega;\Delta \vdash_{\text{er}} \varphi \ni \rho'_\omega \quad \Omega;\Delta \vdash_{\text{er}} \varphi \ni \rho_\omega}{\Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} \ \texttt{parmap}\ e\ [e_1, \ldots, e_n]\ \texttt{at}\ \rho_\omega : (\ \texttt{Vec}\ \ \tau_2, \rho_\omega), \varphi}$$

$$\frac{\Omega;\Delta \vdash_{\text{type}} \tau \quad \vdash_{\text{ctxt}} \Omega;\Delta;\Gamma;\{\rho^1_{\mathbb{C}}, \ldots, \rho^n_{\mathbb{C}}\} \quad \rho_{\mathbb{C}}, \Omega;\Delta, \rho_{\mathbb{C}} \geq \{\rho^1_{\mathbb{C}}, \ldots, \rho^n_{\mathbb{C}}\};\Gamma \vdash^{\mathbb{C}}_{\text{exp}} e : \tau, \{\rho^1_{\mathbb{C}}, \ldots, \rho^n_{\mathbb{C}}, \rho_{\mathbb{C}}\}}{\Omega;\Delta;\Gamma \vdash^{\mathbb{C}}_{\text{exp}} \ \texttt{letregion}\ \rho_{\mathbb{C}}\ \texttt{in}\ e : \tau, \{\rho^1_{\mathbb{C}}, \ldots, \rho^n_{\mathbb{C}}\}}$$

$$\frac{\vdash_{\text{ctxt}} \Omega;\Delta;\Gamma;\{\rho^1_{\mathbb{C}}, \ldots, \rho^n_{\mathbb{C}}\} = \varphi' \quad \Omega \vdash_{\text{place}} \rho_{\mathbb{C}} \quad \rho^{\mathbb{C}}_{\mathbb{G}}, \rho^{\mathbb{C}_1}_{\mathbb{G}}, \ldots, \rho^{\mathbb{C}_n}_{\mathbb{G}}, \Omega;\Delta, \rho^{\mathbb{C}_i}_{\mathbb{G}} \geq \{\rho^{\mathbb{C}}_{\mathbb{G}}\}, \rho^{\mathbb{C}}_{\mathbb{G}} \geq \varphi';\Gamma \vdash^{\mathbb{G}}_{\text{exp}} e : \{\rho^{\mathbb{C}_i}_{\mathbb{G}}, \rho^{\mathbb{C}}_{\mathbb{G}}\} \quad \Omega;\Delta \vdash_{\text{er}} \varphi \ni \rho_{\mathbb{C}} \quad \Omega;\Delta \vdash_{\text{ee}} \varphi \supseteq \varphi'}{\Delta;\Gamma \vdash^{\mathbb{C}}_{\text{exp}} \ \texttt{on}\ e\ \texttt{at}\ \rho_{\mathbb{C}} : (t, \rho_{\mathbb{C}}), \varphi}$$

Fig. 2. Typing rules for the parallel constructs of $\lambda_{\mathcal{G}}$

where Ω is the current set of valid computational regions, Δ is the current set of region constraints[1], Γ is a mapping from variables to types, and φ is a set of bounding effects for the expression e, and is read "expression e running on computational world ω has type τ and bounding effects φ." The most interesting typing judgements for the language are presented in Figure 2. The typing rules for the the rest of the language are fairly standard, and can be found along with context and type well-formedness rules in Appendix A.

It is in the typing rules in Figure 2 that we enforce the majority of the constraints on our language and which operations are permitted on a given computational world. In particular, in the **letregion** rule, we enforce that the current computational world must be the CPU when we encounter such a form. Since the typing rule for the **on** form causes the subexpression to be typed on the GPU computational world, and since there is no way of relocating computation to the CPU from the GPU, this then statically disallows having **letregion** expressions within **on** forms. Likewise, the fact that the subexpression in the **on** form is typed in $\mathbb{G}$ further statically disallows sparking off sub-kernels on the GPU.

4 MEMORY MODELS

In this section we describe the memory model and layout for $\lambda_{\mathcal{G}}$ and detail the creation, layout, access, and transfer of regions in the underlying memory model for this language. In particular we describe a region based memory model with immutable regions and reference counting to deal with task parallelism. While region-based memory models have been well studied in the general setting (Tofte and Talpin 1997), region-based memory models for heterogeneous languages are a relatively new avenue of research having only been introduced in the past three years (Holk 2016; Holk et al. 2014). And while we re-use a large amount of this previous work on regions, and particularly the work on regions for heterogeneous languages, we depart from previous work in some critical areas – especially on the GPU.

4.1 Region Based Memory Management

Representing our heap as a series of "heaplets" on a stack—the *region stack*—that grows and shrinks as we progress through the expression and encounter **letregion** forms greatly simplifies the space and transfer cost analysis that we will later conduct. However, introducing a RBMM system especially in a hetrogeneous parallel setting introduces its own challenges that need to be overcome. In particular, we need to be careful not only about disallowing same-world race conditions for allocation within regions, but also disallowing cross-world race conditions while merging parent and child regions. However, before we can conquer these challenges we must first define the lay of the land. We thus start out defining what exactly a region 'is':

Definition 4.1 (Region). We define a *region* R to be a tuple $(\texttt{ap}, s, \texttt{refcnt}, \nu)$ where:

[1]Where $\rho_\omega \geq \rho'_\omega$ means that ρ'_ω outlives ρ_ω, or in other words, that ρ_ω occurs to the left of ρ'_ω in the region stack.

$$\frac{|v| + \mathsf{ap} > |R| \quad o = \mathsf{ap} \quad n = max(BSIZE, |v| + \mathsf{ap} - |R|)}{R; v \downarrow^{\delta} o@(\mathsf{ap} + |v| + 1, s + n, \mathtt{refcnt}, \nu[o \mapsto v])}$$

$$\frac{|v| + \mathsf{ap} \leq |R| \quad o = \mathsf{ap}}{R; v \downarrow^{\delta} o@(\mathsf{ap} + |v| + 1, s, \mathtt{refcnt}, \nu[o \mapsto v])} \qquad \frac{\ell = (r, o) \quad \mathcal{S}(r) = (\mathsf{ap}, \_, \_, \_, \nu) \quad o < \mathsf{ap} \quad o \in \mathrm{dom}(\nu) \quad v = \nu(o)}{\mathcal{S}; \ell \uparrow^{\delta} v}$$

Fig. 3. Allocation and Reading

- ap is the allocation pointer for the region. Access is atomic;
- s is the size (in bytes) of the region. Access is atomic;
- `refcnt` is the reference count for the region. Access is atomic;
- $\nu : \mathbb{N}_{bounded} \rightarrow v$ is a mapping from offsets o in the underlying memory for the region to heap values v. The offsets are bounded by the size s of the region.

Since our regions are simply a pair of values to manage the book-keeping for the underlying memory of the region, creating them is a fairly straightforward process with the exception of initializing the allocation pointer; since we must record the base of the region (*i.e.*, where the region begins in global memory) this involves in the simplest case, global synchronous access to a global offset. Therefore in order to simplify things, in this paper we assume that creation of regions involves a singular atomic fetch-and-add on the global allocation pointer (where we add a predetermined base size $BSIZE$ to the global allocation pointer). Bumping a global allocation pointer to determine allocation areas for new regions is a non-optimal algorithm since we will quickly run out of space in which to create regions, with a better solution being to use a global free-list of start locations for regions (and push and pop to these). We do not do this however since it is not critical for the theory, and merely complicates things. We thus make the following simplifying assumptions about allocation (and de-allocation) of regions:

i). All creation of regions are race-free (either through a critical section, or since they can be created atomically);
ii). Once a region is popped off from the region stack, the portion of memory used by that region may now be used either by surrounding regions, and/or may be used to create other regions;
iii). The size of global memory is large enough that all allocations of new regions will succeed.

Also, in order to lessen the notational burden, we shall often conflate the region $R = (\mathsf{ap}, s, \mathtt{refcnt}, \nu)$ with its underlying memory ν where there is no ambiguity. *e.g.*, $|R|$ is the size of $\nu, \mathrm{dom}(R) = \mathrm{dom}(\nu)$ etc.

Once a region has been allocated, allocations within that region simply involve an atomic bumping of the region allocation pointer, with the only difficulty arising when we run out of space in the underlying memory. In the latter case we extend the existing region's underlying memory with enough space and then retry the allocation. [2] This allocation into regions is formalized by the judgement $R; v \downarrow^{\delta} o@R'$ which says that allocation of the heap value v has cost δ and is allocated at offset o in the modified region R'. We similarly have a rule $\mathcal{S}; \ell \uparrow^{\delta} v$ for reading locations in memory which says that reading location ℓ results in value v in store $\mathcal{S}$ with cost δ. The judgements for these two forms can be found in Figure 3. It may be useful to the reader (indeed we encourage the reader) to ignore the costs for now, since these will be explained and only become useful in Section 6.

A critical aspect of our regions is that the memory in them is immutable. In particular, not subject to race conditions. Forcing our regions to be immutable gives us the internal structure to the region that allows us to easily merge regions that have possibly evolved in parallel on both host and device; since during the merge process, we know that those sections of the region that are shared will be the same (*i.e.*, we don't have to worry

[2] As a simplifying assumption, we will assume for now that this simply involves increasing the size of the underlying memory, and that we do not have to reallocate the whole underlying memory buffer in some other area global memory.

about merge conflicts in our regions). Another imporant aspect that will be useful is the external structure between regions given by the region stack; regions in the regions stack form a bounded join semilattice with respect to the relation :> and a given region stack $\mathcal{S}$, where we say that $R_1 :> R_2$ if there exist region stacks $\mathcal{S}_1, \mathcal{S}_2$, and $\mathcal{S}_3$ (all possibly empty) such that for our current store $\mathcal{S}$, $\mathcal{S} = \mathcal{S}_1, R_1, \mathcal{S}_2, R_2, \mathcal{S}_3$. In this case we say that R_1 is an *ancestor* of R_2. We also define any region to be an ancestor of itself (*i.e.*, $R :> R$ for any region R).

4.2 Properties of GPU Memory

While CPU and GPU memory are similar in many ways, there are some crucial differences both in how memory accesses take place, and how where in memory the data resides can affect the cost of reading that location. The memory hierarchy of the GPU, while similar to that of the CPU, differs in some critical ways that affect both who can access the memory as well as the cost of the accesses[3]. In particular, in a similar vein to CPU memory, we have a hierarchy of levels of memory – private work item memory, local work group memory, and global or constant memory. However, access to global memory is unsynchronizable, but we do have access to atomics. With these restrictions in mind, this then means that on the GPU, we cannot resize the region – since this would require synchronization operations (since we would need to perform multiple operations atomically in order to avoid race conditions). This means that we are only able to allocate from the space already available within the region when it is copied over to the GPU. This is also one of the main reasons for restricting region creation to be on the CPU only.

On the GPU, the regions are represented using standard memory buffers (and hence reside in global memory). While this leaves open the possibility of optimizing region placement in non-global memory in order to speed-up access from within a kernel based on the work on logical regions by Bauer et al. (2012) we do not address this here and leave it to future work – however the theory we present does take into account local (work group) memory characteristics. While it may be more optimal to instead lazily transfer regions across as they are needed, this complicates the cost reasoning both for space usage and memory transfer costs in our semantics. We therefore take the view that data is copied and allocated immediately for each region that is needed by the expression that we are about to run on the device when we perform an **on** call, and that these regions are immediately deallocated upon completion of the **on** form and after the requisite copies from device to host memory are performed. Since allocating buffers on the GPU is relatively inexpensive as compared to the cost of transferring region contents from host to device memory, we are not overly concerned with coalescing memory transfers and trying to not deallocate data that may just be copied back later (although this would be possible, it would complicate the theory considerably).

4.3 Transferring Memory

Copying regions from the host to the device and vice-versa is done in a piecewise fashion for a given expression; if e is an expression that uses regions R_1, R_2 and R_3, then we will generate a separate copy call[4] for each separate region at the time that the **on** form is encountered. Since this copy is done 'pointwise' on each region in our memory, we not only have that this will preserve the same lattice structure on the GPU as we had on the CPU between the copied regions, but also that the composition of copies will preserve the region structure that we have from :> *i.e.*, :> is a bounded join semilattice on regions, and our copy functions create a lattice homomorphism. This preservation of lattice structure throughout the memory transfer process is critical to the safety of our language, since if we lose this lattice structure, our type system may no longer reflect the liveness properties that are used at the type level of our program – in other words, it would possibly allow well-typed programs to go wrong. The fact that preservation of the term-level ancestor relationship (:>) preserves the type-level outlives ($\geq$) relationship that we require and vice-versa is formalized in the following theorem:

Theorem 4.2. *Let ρ_ω and ρ'_ω be two region types, and let e be an expression whose type has effects that encompass*

[3]See Section 6 for how these memory differences affect the read and write judgements in Figure 3.
[4]Such as `clEnqueueReadBuffer` and `clEnqueueWriteBuffer`

ρ_ω and ρ'_ω, and such that there exists a subexpression e' of e such that $\rho_\omega, \rho'_\omega \in frv(e')$. Then if, during an evaluation step of e, $\mathcal{T}(\omega)(E(\rho_\omega)) = R_1$ and $\mathcal{T}(\omega)(E(\rho'_\omega)) = R_2$ then the following statements hold:

1). R_1 and R_2 are both regions on ω;

2). Let $e = C[e']$ for some valid context C. Then $R_1 :> R_2$ on the subexpression e' of e iff $\Omega; \Delta \vdash_{rr} \rho_\omega \geq \rho'_\omega$. Where Ω and Δ are the region context, and region constraints respectively that are built from typechecking e until the end of the e'.

Proof. By induction on the structure of the typing rules and operational semantics. □

Now that we are transferring regions of memory to- and from device memory, as hinted at earlier, one of the critical questions to ask is whether regions that have possibly evolved in parallel on both CPU and GPU can have 'merge conflicts' when we go to transfer a given region on the GPU back to its parent region on the CPU. In the end, due to immutability of the data in our regions, we can prove that conflicts do not arise in device-to-host transfers. In order to formalize this however, we first need to define the device-host region merging process:

Definition 4.3 (Region Merging). Let R be a region on $\mathbb{C}$ and R' be a copy of R on $\mathbb{G}$. Take $R_1 = (\mathsf{ap}_1, s_1, \mathtt{refcnt}_1, \nu_1)$ and $R'_1 = (\mathsf{ap}'_1, s'_1, \mathtt{refcnt}'_1, \nu'_1)$ to be (possibly empty) evolutions of R and R' respectively. Then we define the *merge* R_2 of R_1 and R'_1 with cost δ, written $R_1; R'_1 \rhd R_2; \delta$ as:

$$\frac{\delta = \omega^\dagger \beta(R_1 \setminus R'_1) \qquad R_2 = memcpy(R_1 \setminus R'_1, \mathsf{ap}_1)}{R_1; R'_1 \rhd R_2; \delta}$$

Where *memcpy* is a native way of transferring memory from device to host[5] and each of the underlying functions that we use to transfer memory from one computational world to another are associated with primitive cost functions $\omega^\star_\beta : \nu \to \mathbb{N}_{bounded}$ and $\omega^\dagger_\beta : \nu \to \mathbb{N}_{bounded}$ that given a segment of memory on one computational world determine the cost of transferring that memory over the bus. We will define what exactly these bus-transfer cost functions are, and how they can be derived in Section 6.

Using this definition of region-merging, we can now formalize merge-conflict freedom in the device-host transfer process:

Theorem 4.4 (Merge Conflict Freedom). *Let $R_1; R'_1 \rhd R; \delta$. Then $\forall v \in range(R_1).v \in range(R) \wedge \forall v \in range(R'_1).v \in range(R)$ and $\forall o \in dom(R_1) \cap dom(R'_1).\nu_1(o) \equiv \nu_2(o)$.*

Proof. By definition of $R_1; R'_1 \rhd R; \delta$ and by immutability of region memory. □

Now that we've defined how to reason about transferring memory between our different computational worlds, we move on to defining the machine models for those worlds and how they interact with one another.

5 ABSTRACT MACHINE MODELS

Since the cost semantics that we will be using are derived from a low-level operational semantics in which we read and write to locations in memory, phrasing the costs in terms of the amount of time it takes to access those locations in memory is the most useful and straightforward way of obtaining accurate costs using our semantic model. The de facto standard for detailing the costs of reading and writing to memory is through an *abstract machine model*, wherein each read or write access to memory is associated with a certain cost or latency. Abstract machine models are well studied, designed to take into account the critical aspects of memory accesses without concerning themselves with the actual workings of the machine or by describing the memory accesses so accurately that the theory becomes unusable. In this way an abstract machine model is perfect for our uses; since we will be defining the cost

[5]Such as `clEnqueueReadBuffer`

of the underlying reads and writes to memory in terms of the underlying abstract machine model for the current computation, we want these to be as accurate as possible while still being usable. However, before we get too far ahead of ourselves defining these costs though, let's first define the machine models we'll be using to derive them.

5.1 CPU Machine Model

There are a number of different well-known abstract machine models that exist for the CPU, and in this paper we will be using the Parallel Random Access Machine (PRAM) model for the CPU (Fortune and Wyllie 1978) with Concurrent Read and Concurrent Write (CRCW). While other models such as LogP (Culler et al. 1993) and BSP (Valiant 1990) can be used, we have chosen the PRAM model due to its simplicity while still offering relatively accurate descriptions of asymptotic complexity

In the PRAM model, a set of P processors share a single memory system. In a single unit of time, each processor can perform an arithmetic, logical, or memory access operation. Moreover each execution step performed by a given processor is synchronized with all other currently running processors. The most important facts to us about the PRAM model are that each operation is unit cost, and that operations are synchronized between processors.

5.2 GPU Machine Model

For the GPU, we have less choice for the machine models that we can use. Particularly, we use the Unified Memory Machine (UMM) model presented by Nakano (2012). Since the UMM model for the GPU is based off of the Discrete Memory Machine (DMM) also presented by Nakano (2012), it's best to start with the DMM and then derive the UMM model from this.

The DMM model for the GPU is parametrized by the number of threads p, the width w, and the latency l, which corresponds to memory banks of size w with (global) read and write latency l. In particular the memory is separated into *banks* B of memory, where $B[j] = \{m[j], m[j+w], m[j+2w], \ldots\}$ denotes the j'th bank of memory (where each $m[j]$ is a memory cell). Memory cells that reside in different banks of memory can be accessed in unit time, and l time units are required in order to complete an access request to global memory, and continuous requests are processed in a pipeline fashion. Thus, it would take $k + l - 1$ time units to perform k continuous access requests to global memory.

The machine is assumed to have a number of threads $T(0), \ldots, T(p-1)$. These p threads are then partitioned into p/w work groups (or warps) $W(0), \ldots, W(p/w - 1)$ such that the i'th work group $W(i) = \{T(i+w), T(i*w+1), \ldots, T((i+1)*w-1)\}$. Each work group $W(i)$ is activated for (possible) memory access in a round-robin fashion if any of the work groups requests memory access, and when a given work group is activated, each of its w threads send memory access requests (one request per thread) to the memory bank. Moreover once a given thread has sent a memory request, it cannot send another memory request until the previous request has been completed *i.e.*, it must wait l time units between memory requests.

The UMM model is then defined in terms of the DMM model by adding *address groups* A where $A[j] = \{m[j*w], m[m*w+1], \ldots, m[(j+1)*w-1]\}$ denotes the j'th address group. Thus memory cells in the same address group do not reside in the same memory bank, and can therefore be accessed simultaneously (and thus a warp access to a single address group takes only unit time regardless of the number of memory cells in the address group that are accessed). Moreover, memory accesses in different address groups cost one time unit each since we will then be accessing multiple memory locations in the same memory bank possibly and since we take into account pipelining of instructions. We also have the same work group, threading, and work group access pattern as in the DMM model.

This is all that we need to know about the UMM model in order to develop a reasonable cost model for the GPU. The interested reader is directed to Nakano (2012) for examples of the UMM and DMM models, and a comparison between the two.

5.3 Connecting Machines

Connecting the machines is, in fact, quite straightforward – or at least moreso than one would imagine. In particular, since we assume implementation-specific memory transfer functions (*i.e.*, memory transfer functions from either openCL (Stone et al. 2010), or CUDA (Nickolls et al. 2008)), the actual connections between the two machine models are straightforward. In a sense the underlying memory transfer functions are extrinsic forces acting on our memory and machines, where the only things we can do are to observe how much they cost via the primitive cost providers (Section 6.6) for those transfer mechanisms. Thus we need not worry about how the two machine models interact with one another beyond the fact that when we transfer memory from $\mathbb{C}$ to $\mathbb{G}$ (or $\mathbb{G}$ to $\mathbb{C}$), that the data will reside in global memory, some of it will be placed on local memory on the GPU, and we have a way to measure how much this process costs.

6 COST SEMANTICS

Now that we have defined the machine models for our language, we can develop the cost theory for $\lambda_{\mathcal{G}}$. Since we are dealing with a parallel language, the scheduling of the threads that we have on the CPU can directly affect the space and time usage of our program. In order to account for these different schedules, we use a similar technique to that introduced by Spoonhower et al. (2010) whereby our cost semantics build a series-parallel DAG of operations for the portions of our programs that reside on the CPU, on which we then impose different traversal strategies. Moreover, in order to account for the high degree of parallelism for GPU programs (*i.e.* kernels), we introduce the notion of a *device box* that takes into account work item cost and is given certain information about the work group size and memory state of the GPU in order to calculate the cost of the GPU expression (more on this in Section 6.4). We then use the machine models that we have just defined to develop costs for the nodes that represent the underlying memory accesses of the the computation in the cost graph that we build.

6.1 Cost Graphs

Earlier, we left some detail out when we said that we build up a DAG with our cost semantics; in reality we wind up building a weighted DAG for our cost graph coupled with special 'blackboxes' – or device boxes – that represent GPU kernels in the CPU cost graph. With this graph structure in mind, we define the the grammar for our graphs along with our core graph combinators in Figure 4. The notation, terminology, and structuring of our graphs does not differ that much from standard graph terminology with the exception of the *dvb* edges which correspond to device boxes. We'll hear more about these special edges shortly in Section 6.4, and while we prepare ourselves for this, let's in the meantime discuss the basic operations that we have on our cost graphs.

Since we aren't concerned with expressing data parallelism on the CPU – and hence only need to express task parallelism in our CPU cost graph – we simply need to be able to express two computations either running in parallel or in serial. We thus only need two core combinators for our graphs: $\oplus$ for serial composition of graphs (run one computation after the other); and $\otimes$ for parallel composition, in which we run the two computations in parallel, and where the δs represent the edge weights for the new edges that we will be adding to the graph via these combinators. In the case that all of the δs are 1, we simply write $\oplus$ and $\otimes$. Moreover, we abuse notation so that when we write $\otimes_i c_i$ and the current computational world is $\mathbb{G}$ this gives us an n-ary version of parallel composition that *does not* form a series-parallel graph for our GPU cost graphs.

Notation & Cost Paths. Since our cost graphs are associated with expressions it will prove useful to be able reason about the unweighted cost graph a given expression generates, we thus define $\mathcal{E}_{\downarrow}(e)$ to be the unweighted cost graph generated by e. Another notion that will prove useful to us later on is that of a *cost path*, which is a cost graph made entirely out of sequential compositions (*i.e.*, it is a purely sequential cost graph); we'll sometimes call these threads.

$$v \in VVars \mid \delta \in \mathbb{N}_{bounded} \mid \delta^{\star}, \delta^{\dagger} : R \times R \times \ldots \times R \to \mathbb{N}_{bounded}$$

dvb	::=	$\left(v_1, v_2, \mathcal{D}, \{R_1, \ldots, R_n\}, \delta^{\star}, \delta^{\dagger}\right)$	Device Boxes
e	::=	$(v_1, v_2, \delta) \mid dvb$	Directed Weighted Edges
V	::=	$\{v_1, \ldots, v_n\}$	Vertex Set
E	::=	$\{e_1, \ldots, e_n\}$	Edge Set
G	::=	(v_1, v_2, V, E)	Graphs

$$
\begin{array}{ll}
[v] & = (v, v, \{v\}, \emptyset) \\
[(v_1, v_2, \delta)] & = (v_1, v_2, \{v_1, v_2\}, \{(v_1, v_2, \delta)\}) \\
(v_1', v_1, V, E) \oplus_{\delta} (v_2', v_2, V', E') & = (v_1', v_2, V \cup V', E \cup E' \cup \{(v_1, v_2', \delta)\}) \\
(v_1', v_1, V, E) \overset{\delta_1'}{\underset{\delta_1}{\otimes}} \overset{\delta_2'}{\underset{\delta_2}{}} (v_2', v_2, V', E') & = \\
 & (v', v, V \cup V' \cup \{v', v\}, E \cup E' \cup \{(v', v_1', \delta_1'), (v', v_2', \delta_2'), (v_1, v, \delta_1), (v_2, v, \delta_2)\}) \quad (v, v' \text{ fresh})
\end{array}
$$

Fig. 4. Cost Graph Operations

6.2 Semantic Rules

The low-level (heap-location-based) operational semantics for the language are also responsible for building the cost graphs that we analyze to determine the cost of our programs. The rules for the operational semantics and how they build up the cost graph are presented in Figures 5 and 6 where the judgment

$$E; \sigma; \mathcal{T}; \omega; e \Downarrow \ell; \sigma'; \mathcal{T}'; c$$

is read "in environment E, local store σ, and configuration table $\mathcal{T}$, e evaluates to a value stored at location ℓ with updated local store σ and configuration table $\mathcal{T}'$ and has cost graph c on world ω." The fact that these are low-level operational semantics is important for the correctness of our theory, since the underlying costs are calculated based on the cost of reads and writes to memory. The majority of the rules for the language are straightforward except for the parallel, region creation, and computational placement rules, so we'll focus on explaining those in this section and leave the rest of the reasoning about the simpler rules to the reader. Since the GPU is a distinctly second-class citizen in terms of its power to control computational placement in our framework, the operational semantics, and the cost graph that they build all take place and reside on $\mathbb{C}$, and GPU computations are represented as device box edges in the (CPU) cost graph which are then analyzed separately.[6]

One of the critical aspects about the cost graphs that we build is that every region operation needs to be explicitly represented in the cost graph, and that any pushes and pops to the region stack are marked by nodes in the cost graph as well. This explicit representation of creation and death of regions along with memory operations in our cost graph allows us to profile for space much more easily than we would otherwise be able to, as well as take into account the cost of allocating new regions as well as any (possible) deallocation costs for our regions. Moreover, making our memory operations explicit in the cost graph is critical to proper cost inference on the graph that is built up and to ensure that we meet the requirements of the abstract machine models that we are using. We also introduce some notation that we haven't seen before so we don't get bogged down with notation in the rules: we write $\mathcal{T}[R'/R]$ for updating (in place) the region R in $\mathcal{T}$ with the updated region R'; we write $\mathcal{T}(\omega)$ to mean that we are accessing the region stack for computational world ω; and finally we write $\mathcal{T}_1; \ldots; \mathcal{T}_n \blacktriangleright \mathcal{T}; \delta$ to mean that $\mathcal{T}$ is the merge of all of the shared regions in the $\mathcal{T}_i$ and a copy of all the non-shared regions, and that this has computational cost δ. It is important to note that we have this explicit merging and separation of configuration tables in parallel rules in order to ensure that on a per-thread basis that

[6] We can get away with this since the CPU/host is the one responsible for computational placement. Thus we don't have to worry about GPU code ever calling into CPU code.

$$\frac{x \in \mathrm{dom}(E)}{E;\sigma;\mathcal{T};\omega;x \Downarrow E(x);\sigma;\mathcal{T};\emptyset} \qquad \frac{\begin{array}{c} r = E(\rho_\omega) \quad R = \mathcal{T}(\omega)(r) \quad R;\sigma;i \downarrow^\delta o@R';\sigma' \\ \mathcal{T}' = \mathcal{T}(w)[R'/R] \quad \ell = (r,o) \end{array}}{E;\sigma;\mathcal{T};\omega;i \ \texttt{at}\ \rho_\omega \Downarrow \ell;\sigma';\mathcal{T}';[u_1] \oplus_\delta [u_2]} \ (u_1, u_2 \text{ fresh})$$

$$\frac{}{E;\sigma;\mathcal{T};\ \texttt{True} \ \Downarrow\ \texttt{True}\ ;\sigma;\mathcal{T};[u]}\ (u \text{ fresh}) \qquad \frac{}{E;\sigma;\mathcal{T};\omega;\ \texttt{False} \ \Downarrow\ \texttt{False}\ ;\sigma;\mathcal{T};[u]}\ (u \text{ fresh})$$

$$\frac{\begin{array}{c} E;\sigma;\mathcal{T};\omega;e_1 \Downarrow \ell;\sigma';\mathcal{T}';c_1 \quad \mathcal{T}'(\omega);\sigma';\ell \uparrow^\delta \ \texttt{True}\ ;\sigma'' \\ E;\sigma'';\mathcal{T}';e_2 \Downarrow \ell';\mathcal{T}'';c_2 \end{array}}{E;\sigma;\mathcal{T};\omega;\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow \ell';\mathcal{T}'';c_1 \oplus_\delta c_2} \qquad \frac{\begin{array}{c} E;\sigma;\mathcal{T};\omega;e_1 \Downarrow \ell;\sigma';\mathcal{T}';c_1 \quad \mathcal{T}'(\omega);\sigma';\ell \uparrow^\delta \ \texttt{False}\ ;\sigma'' \\ E;\sigma'';\mathcal{T}';e_3 \Downarrow \ell';\mathcal{T}'';c_2 \end{array}}{E;\sigma;\mathcal{T};\omega;\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow \ell';\mathcal{T}'';c_1 \oplus_\delta c_2}$$

$$\frac{\begin{array}{c} r = E(\rho_\omega) \quad R = \mathcal{T}(\omega)(r) \quad R;\sigma;\ \texttt{clos}\ (x,e,E) \downarrow^\delta o@R';\sigma' \\ \mathcal{T}' = \mathcal{T}[R'/R] \quad \ell = (r,o) \end{array}}{E;\sigma;\mathcal{T};\omega;\lambda x:\tau.^\varphi e\ \ \texttt{at}\ \rho_\omega \Downarrow \ell;\sigma';\mathcal{T}';[u_1] \oplus_\delta [u_2]}\ (u_1, u_2 \text{ fresh})$$

$$\frac{\begin{array}{c} E;\sigma;\mathcal{T};\omega;e_1 \Downarrow \ell_1;\sigma_1;\mathcal{T}_1;c_1 \quad \mathcal{T}_1(\omega);\sigma_1;\ell_1 \uparrow^\delta \ \texttt{clos}\ (x,e,E);\sigma_2 \\ E;\sigma_2;\mathcal{T}_1;\omega;e_2 \Downarrow \ell_2;\sigma_3;\mathcal{T}_2;c_2 \quad E[x \mapsto \ell_2];\sigma_3;\mathcal{T}_2;e \Downarrow \ell_3;\sigma_4;\mathcal{T}_3;c_3 \end{array}}{E;\sigma;\mathcal{T};\omega;e_1\ e_2 \Downarrow \ell_3;\sigma_4;\mathcal{T}_3;c_1 \oplus_\delta c_2 \oplus c_3}$$

$$\frac{\begin{array}{c} E;\sigma;\mathcal{T};\omega;e_1 \Downarrow \ell_1;\sigma_1;\mathcal{T}_1;c_1 \qquad E;\sigma_1;\mathcal{T}_1;\omega;e_2 \Downarrow \ell_2;\sigma_2;\mathcal{T}_2;c_2 \\ r = E(\rho_\omega) \quad R = \mathcal{T}_2(r) \quad R;\sigma_2;(\ell_1,\ell_2) \downarrow^\delta o@R';\sigma_3 \quad \mathcal{T}_3 = \mathcal{T}_2[R'/R] \quad \ell = (r,o) \end{array}}{E;\sigma;\mathcal{T};\omega;(e_1,e_2)\ \texttt{at}\ \rho_\omega \Downarrow \ell;\sigma_3;\mathcal{T}_3;c_1 \oplus c_2 \oplus_\delta [u]}\ (u \text{ fresh})$$

$$\frac{E;\sigma;\mathcal{T};\omega;e \Downarrow \ell;\sigma';\mathcal{T}';c_1 \quad \mathcal{T}'(\omega);\sigma';\ell \uparrow^\delta (\ell_1,\ell_2);\sigma''}{E;\sigma;\mathcal{T};\omega;\ \texttt{fst}\ e \Downarrow \ell_1;\sigma'';\mathcal{T}';c_1 \oplus_\delta [u]}\ (u \text{ fresh})$$

$$\frac{E;\sigma;\mathcal{T};\omega;e \Downarrow \ell;\sigma';\mathcal{T}';c_1 \quad \mathcal{T}'(\omega);\sigma';\ell \uparrow^\delta (\ell_1,\ell_2);\sigma''}{E;\sigma;\mathcal{T};\omega;\ \texttt{snd}\ e \Downarrow \ell_2;\sigma'';\mathcal{T}';c_1 \oplus_\delta [u]}\ (u \text{ fresh})$$

$$\frac{\begin{array}{c} r = E(\rho_\omega) \quad R = \mathcal{T}(\omega)(r) \quad R;\sigma;\ \texttt{clos}\ (f,u,E[f \mapsto (r,o)]) \downarrow^\delta o@R';\sigma' \\ \mathcal{T}' = \mathcal{T}[R'/R] \quad E[f \mapsto (r,o)];\sigma';\mathcal{T}';\omega;u \Downarrow \ell';\sigma'';\mathcal{T}'';c \end{array}}{E;\sigma;\mathcal{T};\omega;\ \texttt{fix}\ f : (\mu,\rho_\omega).u \Downarrow \ell';\sigma'';\mathcal{T}'';[u_1] \oplus_\delta c}\ (u \text{ fresh})$$

Fig. 5. Cost semantics for the serial portion of $\lambda_{\mathcal{G}}$

the ordering constraints on our region stacks are not violated.[7]

The parameterization of the cost semantics by the world of execution is a critical aspect of its definition. Since as we will see in Section 6.4 the GPU cost graphs overlay these cost semantics with scheduling policies and extra state that we pass in through the local store σ. This then gets reflected in the read and write costs to memory. Thus, these operational semantics are used not only for building up the CPU cost graph, but also the cost path for each individual GPU thread executing within a warp on the device.

Since the local store is only used for GPU computations currently[8] we don't have to worry about merging, or how we thread the local state through these rules on the CPU. However, we do need to worry about how the local state is dealt with in the parallel rules for the GPU. In this case, we use the underlying machine serialization to memory banks and read/write consistency of global GPU memory to our benefit, and since local request state Σ_i^j in σ is only there for us to track our memory costs (which take into account these GPU memory artifacts) we can get around having to deal with parallel merges of the local store[9]. In particular, we use these underlying machine and semantic artifacts to say that all access to the local store is seen globally within that warp's work

[7] Although the underlying memory for the regions that the region names within the region stack point to are shared between threads.
[8] Although, the theory could be easily extended to also encompass CPU caches.
[9] Although we could do this via merge functions, but this would then lead to a number of uni-task functions in order to calculate bank-conflict costs etc.

$$\frac{\begin{array}{c} E;\sigma;\mathcal{T};\omega;e_1 \Downarrow \ell_1;\sigma';\mathcal{T}_1;c_1 \quad E;\sigma;\mathcal{T};\omega;e_2 \Downarrow \ell_2;\sigma';\mathcal{T}_2;c_2 \\ \mathcal{T}_1;\mathcal{T}_2 \blacktriangleright \mathcal{T}_3;\delta \quad r = E(\rho_\omega) \quad R = \mathcal{T}_3(\omega)(r) \\ R;\sigma';(\ell_1,\ell_2) \downarrow^{\delta_a} o@R';\sigma_1 \quad \mathcal{T}_4 = \mathcal{T}_3[R'/R] \quad \ell = (r,o) \end{array}}{E;\sigma;\mathcal{T};\omega;e_1 \| e_2 \ \texttt{at}\ \rho_\omega \Downarrow \ell;\sigma_1;\mathcal{T}_4;(c_1 \otimes c_2) \oplus_\delta [u_1] \oplus_{\delta_a} [u_2]} \ (u_1, u_2 \text{ fresh})$$

$$\frac{\begin{array}{c} E;\sigma_{i-1};\mathcal{T}_{i-1};\omega;e_i \Downarrow \ell_i;\sigma_i;\mathcal{T}_i;c_i \quad r = E(\rho_\omega) \quad R = \mathcal{T}_n(\omega)(r) \\ R;\sigma_n;[\ell_1,\ldots,\ell_n] \downarrow^{\delta} o@R';\sigma_n' \quad \ell = (r,o) \quad \mathcal{T}_n' = \mathcal{T}_n[R'/R] \end{array}}{E;\sigma_0;\mathcal{T}_0;\omega;[e_1,\ldots,e_n] \ \texttt{at}\ \rho_\omega \Downarrow \ell;\sigma_n';\mathcal{T}_n';\left(\bigoplus_i c_i\right) \oplus_\delta [u]} \ (u \text{ fresh})$$

$$\frac{\begin{array}{c} E;\sigma;\mathcal{T};f\ e_i \Downarrow \ell_i;\sigma';\mathcal{T}_i;c_i \quad \mathcal{T}_1;\ldots;\mathcal{T}_n \blacktriangleright \mathcal{T}';\delta_m \\ r = E(\rho_\omega) \quad R = \mathcal{T}'(\omega)(r) \\ R;\sigma';[\ell_1,\ldots,\ell_n] \downarrow^{\delta_a} o@R';\sigma'' \quad \mathcal{T}'' = \mathcal{T}'[R'/R] \end{array}}{E;\sigma;\mathcal{T};\omega;\ \texttt{parmap}\ f\ [e_1,\ldots,e_n] \ \texttt{at}\ \rho_\omega \Downarrow \ell;\sigma'';\mathcal{T}'';\left(\bigotimes_i c_i\right) \oplus_{\delta_m} [u_1] \oplus_{\delta_a} [u_2]} \ (u_1, u_2 \text{ fresh})$$

$$\frac{\begin{array}{c} R = newRegion(\mathcal{T}) \quad r_{\mathbb{C}} \text{ fresh} \quad r_{\mathbb{C}} \notin \text{dom}(\mathcal{T}[\mathbb{C}]) \quad \delta_R = \Delta^a(|R|) \\ E[\rho_{\mathbb{C}} \mapsto r_{\mathbb{C}}];\sigma;\mathcal{T}(\mathbb{C})[r_{\mathbb{C}} \mapsto R];\mathbb{C};e \Downarrow \ell;\sigma;\mathcal{T}';c \\ \mathcal{T}'' = \mathcal{T}' \setminus r_{\mathbb{C}} \end{array}}{E;\sigma;\mathcal{T};\mathbb{C};\ \texttt{letregion}\ \rho_{\mathbb{C}}\ \ \texttt{in}\ e \Downarrow \ell;\sigma;\mathcal{T}'';[u_1] \oplus_{\delta_R} [u_2] \oplus c \oplus [u_3]} \ (u_1, u_2, u_3 \text{ fresh})$$

$$\frac{\begin{array}{c} \{\rho_{\mathbb{C}}^1,\ldots,\rho_{\mathbb{C}}^n\} = frv(e) \quad r_i = E(\rho_{\mathbb{C}}^i) \quad R_i = \mathcal{T}(\mathbb{C})(r_i) \\ push(R_i,R_i') \quad \mathcal{T}' = \mathcal{T}(\mathbb{G})[r_i \mapsto R_i'] \\ \delta^\star = \lambda\{R_1,\ldots,R_n\} \to \textstyle\sum_{R_i} \omega_\beta^\star(R_i) \\ \mathcal{D} = [\![e,\mathcal{T}',\ell]\!]_{\mathcal{D}} \quad R_i;R_i' \rhd \widetilde{R}_i;\delta_i \\ \mathcal{T}'' = \mathcal{T}[\widetilde{R}_1/R_1,\ldots,\widetilde{R}_n/R_n] \quad \delta^\dagger = \textstyle\sum_i \delta_i \end{array}}{E;\sigma;\mathcal{T};\mathbb{C};\ \texttt{on}\ e\ \ \texttt{at}\ \rho_{\mathbb{C}} \Downarrow \ell';\mathcal{T}'';(u_1,u_2,\mathcal{D},\{R_1,\ldots,R_n\},\delta^\star,\delta^\dagger)} \ (u_1, u_2 \text{ fresh})$$

Where ℓ' is the corresponding location on $\mathbb{C}$ to the location ℓ on $\mathbb{G}$ after region merging.

Fig. 6. Cost semantics for the parallel and region creation portion of $\lambda_{\mathcal{G}}$

group, and is atomic. Thus in the cost graph computation nodes can be represented as running in parallel, but memory accesses in their underlying read/write judgments assume sequentiality and the accesses to the local store are serialized (/atomic); we simply 'forget' about these serialization costs in the graph we build up in order to represent that reads and writes will (or can) happen in parallel in reality.

Now that we've discussed the overall structuring of our rules, it's time to discuss the intricacies of some of the more interesting rules that are presented in Figure 6: the parallel composition rule $e_1 \| e_2$ first builds the cost graphs c_1 and c_2 for the the expressions e_1 and e_2 respectively, our configuration tables have "split" at this point, so we must merge them back together with cost δ in order to continue on; we then write this pair to memory with cost δ_a. We thus create a cost graph structure in which we compose c_1 and c_2 in parallel, and then charge a cost of δ, and then a cost of δ_a. We explicitly *don't* combine the edge weights of δ and δ_a in order to get rid of an extra node, since we require that each memory operation be made explicit in our cost graph.

For the **letregion** rule, we create a new backing region R in memory, as well as an underlying name $r_{\mathbb{C}}$ for the region, and then add this mapping into the correct region stack in our configuration table and update the environment so that the environment points to the correct underlying name in our region stack. We then charge

δ_R cost of creating this backing region via our Δ^a primitive cost provider, and then build the cost graph c for the body of the **letregion** . We then create an edge with weight δ_R to account for the region creation cost, and add a node (u_2) to demarcate the pushing of the new region onto the region stack. We then serially compose this with the cost graph for the body, and then add an edge to another node (u_3) to demarcate popping off R from the region stack at the end of **letregion** s scope.

The expression e within an **on** expression corresponds to a kernel on the GPU. Thus we first transfer regions that e uses to the GPU using *push* which corresponds to a native copy function from the CPU to the GPU (such as `clEnqueueWriteBuffer`). We then update the configuration table with these new regions, shadowing the underlying region names from the CPU region stack in the GPU region stack. This purposeful name shadowing ensures that any uses of a given region variable within e will result in the correct region on $\mathbb{G}$ being used. We then form a kernel graph (or *device box*) for e by first creating an unweighted cost graph $\mathcal{E}_{\downarrow}(e)$ from the body of the **on** form, but without firing the allocation rules in the semantics. We then eagerly group paths through G into w threads of warps, and then group each of these warps into groups of size p which we call work groups. The precise way we form these graphs from the normal computation graph is described in Section 6.4. Through this process, we generate a device box $\mathcal{D}$ for the expression e. We then serially compose vertices u_1 and u_2 onto this device box that represent the memory transfers to- and from the GPU, as well as have the appropriate edge weight given by $\delta^\star$ that takes into account the regions that need to be transferred to the GPU, and $\delta^\dagger$ which accounts for both the cost of transferring memory back to the CPU, but also the process of merging regions.

6.3 Schedules

While we do not need to worry about different schedules on the GPU since our underlying GPU machine model requires a given schedule (see Nakano (2012) for more), we still need to factor in how different scheduling policies for CPU computations can directly effect which, and when nodes in the cost graph become active. This, in turn, can have a large impact on the space and time usage of the program. In order to parameterize our CPU cost graphs by the scheduling policy that is chosen, we use a similar technique to that of Spoonhower et al. (2010) where a *schedule order* $\trianglelefteq$ for the cost graph g is defined on g's nodes such that $\forall v \in g \;.\; v \trianglelefteq v$, and $\forall v_1, v_2 \in g \;.\; v_1 \prec_g v_2 \Rightarrow v_1 \triangleleft v_2$, where $\prec_g$ means that v_1 is a parent of v_2[10] and $\triangleleft$ is the corresponding strict partial order given by the reflexive reduction of $\trianglelefteq$. This notion of a schedule between nodes can then be easily extended to schedule orders for sets of nodes:

$$V_1 \trianglelefteq V_2 \iff \forall v_1 \in V_1, v_2 \in V_2 \;.\; v_1 \triangleleft v_2 \tag{1}$$

This then provides a way to talk — globally — about all of the events that occur during the evaluation of a program. However as we will see in a moment, when we are discussing space usage it will be useful to be able to restrict our view at times to only those nodes that are scheduled *simultaneously*:

$$v_1 \bowtie v_2 \iff v_1 \trianglelefteq v_2 \wedge v_2 \trianglelefteq v_1 \tag{2}$$

We then use this to partition the graph into sets of simultaneously scheduled nodes called *steps*:

$$\text{steps}(\trianglelefteq) = V_1, \ldots, V_k \;\text{ such that }\; \begin{cases} \exists i \;.\; \forall v \in g \;.\; \exists V_i . v \in V_i, \\ V_1 \triangleleft \ldots \triangleleft V_k, \\ \forall i \;.\; \forall V_i \;.\; \forall v_1, v_2 \in V_i \;.\; v_1 \bowtie v_2 \end{cases} \tag{3}$$

From this, we can then define the *closure* $\widehat{V_i}$ of the step V_i as

$$\widehat{V_i} = \bigcup_{k=1}^{i} V_k \tag{4}$$

[10] the ordering $v_1 \trianglelefteq v_2$ can be read as "v_1 is scheduled no later than v_2"

$$\left\{\begin{array}{lcl} a & \in & ValidAddresses \subset \mathbb{N}_{bounded} \\ \Sigma_i & ::= & \{a_1,\ldots,a_n\} \\ L_i & ::= & \{B[0],\ldots,B[n]\} \\ \mathcal{W}_i^j & ::= & \left\langle W^{k_0}(0),\ldots,W^{k_n}(n), L_i^j \right\rangle \\ \widetilde{v} & ::= & \mathcal{W}_i^j \mid \texttt{finish} \mid \texttt{start} \end{array}\right. \quad \left\{\begin{array}{lcl} W^j(i) & ::= & \left\langle \left\{G^{j_{i+w}}(i+w),\ldots,G_l^j((i+1)*w-1)\right\}, \Sigma_i^j \right\rangle \\ \widetilde{E} & ::= & \left(\mathcal{W}_i^j, \mathcal{W}_i^{j+1}, \delta\right) \mid \left(\mathcal{W}_i^j, \texttt{finish}, \delta\right) \\ & \mid & \left(\texttt{start}, \mathcal{W}_i^j, \delta\right) \\ \widetilde{G} & ::= & \left(\texttt{start}, \texttt{finish}, \{\widetilde{v}\}, \widetilde{E}\right) \\ \mathcal{D} & ::= & \texttt{start} \oplus \bigotimes_i \left(\bigoplus_j \mathcal{W}_i^j \right) \oplus \texttt{finish} \end{array}\right.$$

Fig. 7. Warp- and device box definitions.

Thus while all the nodes in $\widehat{V_i}$ are not necessarily scheduled simultaneously, we have that $\widehat{V_i} \triangleleft V_{i+1}$. With this framework, we can then think of the schedule as defining a wavefront that progresses across the graph (where each "wave" is the next step V_i).

Since our DAG is weighted, it will prove useful to be able to talk about the maximum weight edge that we encounter during a given step since this will determine the amount of time that a given step in the graph $G = (v_1, v_2, V, E)$ takes:

$$\text{step}_w(V_i) = \begin{cases} 0 & i = 1 \\ \max\{\delta \mid v_i \in V_i,\ v_{i-1} \in V_{i-1}, (v_{i-1}, v_i, \delta) \in E\} & \text{otherwise} \end{cases} \tag{5}$$

6.4 Device Boxes

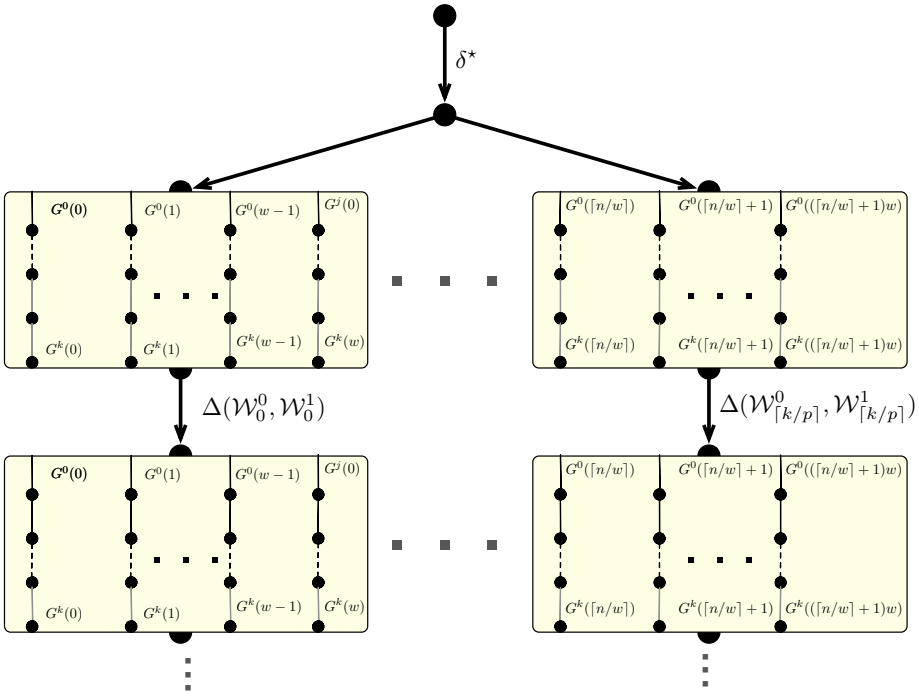
Due to the high amount of parallelism and special memory characteristics of the GPU, the standard approach of analyzing costs via a normal computation graph structure annotated with costs is infeasible. In order to get around these parallelism and memory problems we group GPU computations in our cost graph based on the underlying computational structure of the GPU; work items (threads) represent a single piece of work, which are then grouped into warps, and then into work groups (a *warp box*) which then constitute vertices in the GPU cost graph (a *device box*). Each of these work group nodes has some local state – or local memory – that it keeps track of in addition to a set of warps that make up that particular work group. Since the GPU can have many of these work groups running simultaneously for a given kernel, the GPU cost graph will be a (not series-parallel) weighted DAG, where each node in the graph is a single step of one of these work groups. Thus the definition of these cost graphs for the GPU are – while similar from the macro perspective – quite different in important ways from the cost graphs we have seen before, and the definition of them can be found in Figure 7.

A *device box* $\mathcal{D}$ is a weighted DAG, in which each node in the graph is a warp box at a given step j which contains the local (shared) memory for the warps that are in the work group that it represents. Since every warp is run in parallel within each work group we dictate that at each step of the warp box $\mathcal{W}_i^j \multimap \mathcal{W}_i^{j+1}$ every (runnable) warp $W^k(i)$ is advanced one (and only one) step: $\forall W^k(h) \in \mathcal{W}_i^j \,.\, W^k(h) \twoheadrightarrow W^{k+1}(h)$[11]. Thus if we have n warps in a warp box $\mathcal{W}_i$ with each warp taking k steps to complete, then it would take at least k steps for $\mathcal{W}_i$ to complete. A brief definition of the two stepping relations between our warp boxes and warps along with an example of what the interaction between the two might look like in the device box graph is given in Figure 8.

In order to determine bank conflicts, within each step of a warp we define meta-functions $[-]_B$ and $\mathcal{A}(-)$[12] that given an address returns the bank equivalence class for that address, and the global address for a given (region) location. We can then easily define what it means to have a bank conflict, multicast, and broadcast using these two meta-functions: we say that a warp at step j has a *bank conflict* if there exists a cost path $G^j(k)$ in

[11] When a given warp has no more work left, we mark it as dead.

[12] We shall abuse notation somewhat and also use $\mathcal{A}(R, o)$ to mean the global address for offset o in region R.



$$\begin{array}{lll} \mathcal{W}_i^j \multimap \mathcal{W}_i^{j+1} & ::= & \forall W_k^j \in \mathcal{W}_i^j \;.\; W^j(k) \twoheadrightarrow W^{j+1}(k) \\ W^j(i) \twoheadrightarrow W^{j+1}(i) & ::= & \forall G^{j_l}(k) \in W^j(i) \;.\; G^{j_l}(k) \rightharpoonup G^{j_l+1}(k) \end{array}$$

Fig. 8. Stepping relations between warp boxes and warps. Dashed lines are currently active edges, grey ones are to-be-active, and black ones have been 'run'.

$W^j(i)$ that accesses or write to a location ℓ such that

$$\exists a \in \Sigma_i^j \;.\; [a]_B \equiv [\mathcal{A}(\ell)]_B \wedge a \not\equiv \mathcal{A}(\ell)$$

and we also say in this case that address ℓ creates a bank conflict with local request-state Σ_i^j and we write this as $[\ell]_B \between \Sigma_i^j$. We can likewise say that a warp box at step j has a *multicast* if there exists an $a \in L_i^j$ such that

$$\left|\{G^j(k) \mid G^j(k) \in \mathcal{W}_i^j \wedge \mathcal{A}(\ell_k) \equiv a\}\right| > 1 \tag{6}$$

where ℓ_k is the location being accessed by cost path $G^j(k)$. We say that this is a *broadcast* if Equation (6) is equal to the number of active threads (*i.e.*, cost paths) in $W^j(i)$.

Building device box graphs. Now that we've formalized bank conflicts and broadcasts, we can formalize the construction of both the device box graph, the warp boxes, and the warps: let $G = \mathcal{E}_\downarrow(e)$ and let $\{G(0), \ldots, G(n)\}$ be the set of *distinct* paths through this graph. We define the step $\rightharpoonup$ of a cost path to be the traversal of one (and only one) edge in the graph and we denote the path $G(i)$ at step j by $G^j(i)$, and if there are no more vertices

to traverse, $\rightharpoonup$ is a no-op [13]. We then create $\lceil n/w \rceil$ warps, where w is our warp-width, as follows:

$$W^0(0) = \left\langle \left\{ G^0(1), \ldots, G^0(w) \right\}, \{\} \right\rangle, \ldots, W^0(\lceil n/w \rceil) = \left\langle \left\{ G^0(\lceil n/w \rceil w + 1), \ldots, G^0((\lceil n/w \rceil + 1)w) \right\}, \{\} \right\rangle$$

Where the {} are the empty local-memory request states for each warp. We then group these warps into $\lceil k/p \rceil$ warp boxes (where p is our work group size and $k = \lceil n/w \rceil$) in the same manner as we built up our warps:

$$\mathcal{W}_0^0 = \left\langle \left\{ W^0(0), \ldots, W^0(w) \right\}, L_i^0 \right\rangle, \ldots, \mathcal{W}_{\lceil k/p \rceil}^0 = \left\langle \left\{ W^0(\lceil k/p \rceil p), \ldots, W^0(\lceil k/p \rceil + 1)p \right\}, L_{\lceil k/p \rceil}^0 \right\rangle$$

We now define $\text{time}_\delta(G^j(i), G^{j+1}(i))$ to be the weight of the edge between the j'th and j'th plus 1 vertices in the cost path $G(i)$[14]. Using this, we then define the edge weight $\Delta\left(\mathcal{W}_i^j, \mathcal{W}_i^{j+1}\right)$ to be

$$\Delta\left(\mathcal{W}_i^j, \mathcal{W}_i^{j+1}\right) = \max_{G^j(i) \in \mathcal{W}_i^j} \text{time}_\delta(G^j(i), G^{j+1}(i)) - \mathbb{B}_j^i$$

where $\mathbb{B}_j^i$ corrects for the memory reads and writes that we have counted as bank conflicts which in fact have led to multicasts or broadcasts, and is thus calculated as $\mathbb{B}_j^i = \max\{0, (n-k)2\}$ where k is the size of the set in Equation (6) and n is the total number of (GPU) threads in $\mathcal{W}_i^j$. We then define our device box $\mathcal{D}$ using these warp boxes, and the definition of $\mathcal{D}$ in Figure 7 where we recall that since we are on $\mathbb{G}$ that $\otimes$ forms the n-ary parallel composition of its arguments[15]. We then define the (meta) function $[\![e, \mathcal{T}, \ell]\!]_{\mathcal{D}}$ to be the function that returns to us this device box graph given an expression e that starts out with configuration table $\mathcal{T}$ and returns its result value at location ℓ.

The intuition behind this construction of a warp's cost graph is that it is a normal (CPU) cost graph that has been built up of a number of parallel threads in which each thread is a cost path through a cost graph built using the operational and cost semantics in Figures 5 and 6 using the evaluation world $\mathbb{G}$ passing in the local state (L_i^j, Σ_i^j). We then impose a stepping relation $\twoheadrightarrow$ on the set of cost paths in the warp that ensures that all of the threads in the warp are evaluated in lock-step. The fact that the threads in a warp move in lock-step and that at each step of the warp each cost path in the warp may send one – and only one – memory access request is critical in order to stay true to the UMM abstract machine model. The fact that these cost paths may send only one memory request per step makes our explicit representation of every memory operation as a node in the cost graph crucial to the correctness of the costs, since in this way we can directly link traversal of an edge in a cost path (*i.e.*, $\rightharpoonup$) and a (possible) memory access request.

6.5 Costs of Reading and Writing Memory

Now that we've defined device boxes, we need to update the allocation and read judgements for the GPU, as well as further refine the costs for reads and writes on the CPU. We therefore amend Figure 3 so that the cost determinations in our allocation judgements now differ based upon the computational world that we are on, as well as the underlying memory characteristics of the machine. Since we need to be able to track local memory and local memory requests on the GPU code we also introduce another argument and return value σ in the read and write judgement. We thus obtain the following judgements:

$$R; \sigma; v \downarrow^c o@R'; \sigma'$$
$$\mathcal{S}; \sigma; \ell \uparrow^c v; \sigma'$$

These are read respectively as "allocating heap value v in region R with current local store σ has cost c and results in an updated region R' and updated local store σ'", and "reading location ℓ from region stack $\mathcal{S}$ with

[13] This can be defined in terms of our schedule orders before, but we elide that here for simplicity.

[14] Note that the weight on these paths is computed on the $\mathbb{G}$ computational world and therefore this cost takes into account local memory and bank conflicts.

[15] Hence why the graphs on the GPU are *not* series-parallel.

Memory Costs Metrics for CPU

$$\textsf{ExpandW}\ \frac{n = max(BSIZE, |v| + \mathsf{ap} - |R|) \quad |v| + \mathsf{ap} > |R| \quad o = \mathsf{ap} \quad \delta = \Delta^a(n)}{R; (); v \downarrow^{\delta} o@(\mathsf{ap} + |v| + 1, s + n, \mathtt{refcnt}, v[o \mapsto v]); ()} \qquad \textsf{NoExpandW}\ \frac{|v| + \mathsf{ap} \leq |R| \quad o = \mathsf{ap}}{R; (); v \downarrow^{1} o@(\mathsf{ap} + |v| + 1, s, \mathtt{refcnt}, v[o \mapsto v]); ()}$$

$$\frac{\ell = (r, o) \quad \mathcal{S}(r) = (\mathsf{ap}, \_, \_, \_, v) \quad o < \mathsf{ap} \quad o \in \mathrm{dom}(v) \quad v = v(o)}{\mathcal{S}; (); \ell \uparrow^{1} v; ()}$$

Memory Costs Metrics for GPU

$$\textsf{NoExpandWE}\ \frac{|v| + \mathsf{ap} > |R|}{R; (L_i^j, \Sigma_i^j); v \downarrow^{\delta} \ \mathbf{err}\ ; (L_i^j, \Sigma_i^j)} \qquad \textsf{BankConfW}\ \frac{|v| + \mathsf{ap} \leq |R| \quad o = \mathsf{ap} \quad [\mathcal{A}(R, o)]_B \not\emptyset \Sigma_i^j \quad \Sigma_i^{j+1} = \Sigma_i^j \quad \delta = 2}{R; (L_i^j, \Sigma_i^j); v \downarrow^{\delta} o@(\mathsf{ap} + |v| + 1, s, \mathtt{refcnt}, v[o \mapsto v]); (L_i^j, \Sigma_i^{j+1})}$$

$$\textsf{NBankConfW}\ \frac{|v| + \mathsf{ap} \leq |R| \quad o = \mathsf{ap} \quad [\mathcal{A}(R, o)]_B \not\emptyset \Sigma_i^j \quad \Sigma_i^{j+1} = \mathcal{A}(R, o) :: \Sigma_i^j \quad \delta = \mathcal{A}(R, o) \in L_i^j \ \mathbf{?}\ 1 : l}{R; (L_i^j, \Sigma_i^j); v \downarrow^{\delta} o@(\mathsf{ap} + |v| + 1, s, \mathtt{refcnt}, v[o \mapsto v]); (L_i^j, \Sigma_i^{j+1})}$$

$$\textsf{BankConfR}\ \frac{\ell = (r, o) \quad \mathcal{S}(r) = R\{\mathsf{ap}, \_, \_, \_, v\} \quad [\mathcal{A}(R, o)]_B \not\emptyset \Sigma_i^j \quad o < \mathsf{ap} \quad o \in \mathrm{dom}(v) \quad v = v(o) \quad \Sigma_i^{j+1} = \Sigma_i \quad \delta = 2}{\mathcal{S}; (L_i^j, \Sigma_i^j); \ell \uparrow^{\delta} v; (L_i^j, \Sigma_i^{j+1})}$$

$$\textsf{NBankConfR}\ \frac{\ell = (r, o) \quad \mathcal{S}(r) = R\{\mathsf{ap}, \_, \_, \_, v\} \quad [\mathcal{A}(R, o)]_B \not\emptyset \Sigma_i^j \quad o < \mathsf{ap} \quad o \in \mathrm{dom}(v) \quad v = v(o) \quad \Sigma_i^{j+1} = \mathcal{A}(R, o) :: \Sigma_i \quad \delta = \mathcal{A}(R, o) \in L_i^j \ \mathbf{?}\ 1 : l}{\mathcal{S}; (L_i^j, \Sigma_i^j); \ell \uparrow^{\delta} v; (L_i^j, \Sigma_i^{j+1})}$$

Fig. 9. Definitions of the read and write cost metrics for $\mathbb{C}$ and $\mathbb{G}$.

local store σ has cost c and results in heap value v and updated local store σ'."[16]

The CPU read and write judgements in Figure 9 are largely unchanged from Figure 3 with the exception of the ExpandW rule in which we see a *primitive cost provider* $\Delta^a(n)$ introduced. Primitive cost providers are extrinsic cost functions that are derived either through empirical measurement, or based upon the hardware or software specification (see Section 6.6 for more). In this case, we say that the cost of allocating a value in a region that needs to be expanded in order to fit the new value is the cost of expanding the given region – either by a given base expansion-step size *BSIZE* or by the amount needed in order to make enough room for the new value.

The read and write rules for the GPU on the other hand have been changed drastically. In particular, they have been changed in order to take into account not only local memory but also bank conflicts within local memory based upon the description of the UMM model. Moreover the rules specifically disallow any possible increase in region size on the GPU with any allocation that might lead to such a situation raising an error (NoExpandWE). The BankConfW rule says that if the region that we are allocating into has enough space, and if the (global address) location that has been chosen for its allocation has a bank conflict, that the write operation is charged a cost of 2, and similarly for BankConfR except for the read we don't need to ensure that there is enough room in the region. The NBankConfW rules states that if there is enough room in the region, and if there are no bank conflicts

[16]Note that we will only use this local store for the GPU for now, but this local store and how the judgements (might) interact with them has similarities to the cost judgments found in Blelloch and Harper (2013) for determining the cache efficiency of programs.

for the chosen location, then the write is charged either 1 or l cost based on whether or not it is a write to local or global memory. We reason similarly for reading non bank-conflicting locations in the NBankConfR rule as we did in the NBankConfW rule.

6.6 Primitive Cost Providers

The primitive cost providers (PCPs) for the language are the extrinsic actors in our cost-semantic theory that ground it in reality and are critical to both its expressivness and extensibility. Specifically, PCPs are costs that cannot accurately be determined within the cost framework due to them being implementation and/or machine-type specific. And just as the read and write costs from the underlying abstract machines form the "atomic cost element" in our framework that we use to build our cost graphs, these PCPs form the "atomic cost elements" for memory allocation and transfer costs[17]. These user-provided costs have the benefit of making our cost model more extensible and accurate over a wider range of computational layouts and CPU/GPU configurations since we can have different PCPs that model the different behaviors of allocation and transfer on the different machine-types and layouts that we may encounter. However, these external costs also introduce a point of failure in our cost model; if invalid or inaccurate primitive cost providers are supplied, then the results calculated by the cost model may well be very far off from reality.

The three primitive cost providers that the user needs to supply are the following:

i). A function $\Delta^a : \mathbb{N}_{bounded} \to \mathbb{N}_{bounded}$ that is used to calculate the cost of allocating a chunk of memory of a given size. How this cost is calculated is up to the user, but as was discussed earlier, lack of accuracy here will only be magnified by the cost semantics.

ii). Two functions

$$\begin{aligned} \omega^{\star}_{\beta} &: \nu \to \mathbb{N}_{bounded} \quad \text{Cost of CPU/GPU Transfer of Memory } \nu \\ \omega^{\dagger}_{\beta} &: \nu \to \mathbb{N}_{bounded} \quad \text{Cost of GPU/CPU Transfer of Memory } \nu \end{aligned} \tag{7}$$

Since the cost of transferring memory from the CPU to the GPU and vice-versa can be very costly in comparison to the cost of the actual computation that is done on the GPU the accuracy of these functions are also quite imporant, with lack of accuracy also being magnified in the overall cost that we derive for the program [18].

6.7 Costs

Determining space costs. In order to determine the space usage of our program, we utilize techniques similar to that described by Spoonhower et al. (2010). However instead of keeping a heap graph and having a reachability relation on this graph to track the locations that need to be kept alive, we instead look at the region stacks for both the CPU and the GPU that we maintain and build throughout our graph traversal (on both the CPU cost graph, and within our device boxes). Moreover, since the operations on our configuration table and region stacks are explicit nodes in our cost graphs and device boxes, the configuration table at any given point is completely determined by the closure of our current computation step. We thus define $\mathcal{T}(V_i, \trianglelefteq)$ to be the configuration table at step i under schedule $\trianglelefteq$.

With this definition of a configuration table at step i, we may then define the space usage at a given step in the computation as the total space used by the locations in the configuration table at that step:

$$\text{space}(V, \trianglelefteq) = |\{\ell \in R \mid R \in \mathcal{S}_1 \vee R \in \mathcal{S}_2\}|\,, \quad \mathcal{T}(V_i, \trianglelefteq) = \{\mathcal{S}_1, \mathcal{S}_2\} \tag{8}$$

We can then say that the space required by a schedule $\trianglelefteq$ that computes value v is the maximum space usage

[17] Indeed, if we had an abstract machine that provided both transfer costs and accounted for allocation we could use those cost metrics here.

[18] There has been recent work on calculating accurate costs for these transfers in (Van Werkhoven et al. 2014) but the programmer is free to use other means of calculating the costs of memory transfers.

required by any step in the schedule

$$\text{space}_v(\trianglelefteq) = \max\{\text{space}(V_i, \trianglelefteq) \mid V_i \in \text{steps}(\trianglelefteq)\} \tag{9}$$

Determining time costs for our program is relatively straightforward, with the only intricacies arising from the device boxes in our cost graph since these determine 'higher-order edges.' The weights of these device box edges in the CPU cost graph can be determined as the sum of applying the regions that need to be transferred to the function $\delta^\star$ that accounts for the transfer costs of the regions used by the device box[19], and the time cost for the device box $\mathcal{D}$ is calculated as the longest path from `start` to `finish` node. Thus the total amount of time taken by a program e under schedule $\trianglelefteq$ with cost graph $G = (v_1, v_2, V, E)$ is the sum of the time taken for each step given by $\trianglelefteq$ through G, or in other words

$$\text{time}(e, \trianglelefteq) = \sum_{V_i \in \text{steps}(\trianglelefteq)} \text{step}_w(V_i)$$

where step_w was defined in Equation (5).

7 DISCUSSION

In this section we give a brief example of how the theory infers costs for a program, and use this example to motivate future possible applications and avenues of research building on the theory that we have developed. In particular, we briefly look at how the cost semantic theory that we've presented in this paper creates the opportunity to:

- Determine whether certain expressions within an **on** form should actually be run on the GPU (*i.e.*, should an **on** form be inserted);
- Use the cost semantics as a scoring function to guide rewrite and computational placement strategies and memory placement strategies during compilation.

7.1 A brief example

Take the following program in $\lambda_{\mathcal{G}}$ that given two vectors of integers that have been zipped together returns a vector with the product of the two:

```
letregion r0 in
  lambda arrs : (Vec (int r0, int r0), r0) [r0].
    on   parmap (lambdap:((int r0, int r0), r0) [r0].(fst p * snd p) at r0 arrs at r0
    at r0
```

and where we want to determine whether or not we want to insert the underlined **on** form. Moreover, in case we do decide to run the expression in gray on the GPU, we also want to know how much it would cost if we didn't use any local memory on the GPU, and if we used w work groups of size p for the kernel.

We fix the following for our analysis: we will use $P = 2$ processors on the CPU, and set $\trianglelefteq$ to be a greedy depth-first schedule on the nodes; we shall simplify and assume that all values within the array `arrs` have been evaluated, and are simply pairs of locations (ℓ_1, ℓ_2) that both point to integers i allocated in the `r0` region.

We thus have from the rules in Figure 5 that we will build up the cost graph in Figure 10 for each application $f\ e_i$ where δ_r is the cost of reading a memory location on our current computation world, and δ_w is the cost of writing to a memory location. In order to simplify the cost analysis, we'll also assume that the region doesn't need to be resized during the running of the grayed-out code.

[19] Since regions cannot be expanded on the GPU, we can calculate the merging cost by taking the worst case scenario that all the free space in the regions that are being transferred to the GPU are filled and need to be transferred back to their CPU parent regions, however instead we have it tracked in the region merging definition in Definition 4.3.

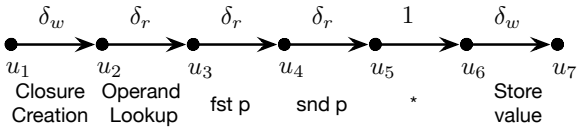


Fig. 10. The (annotated) cost graph generated from the lambda expression inside the grayed-out code in Listing 7.1

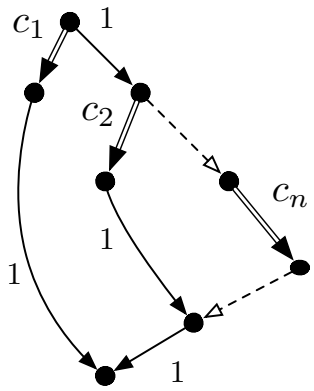


Fig. 11. The DAG generated by the grayed code in Listing 7.1 and where the double-struck edges represent the DAG in Figure 10.

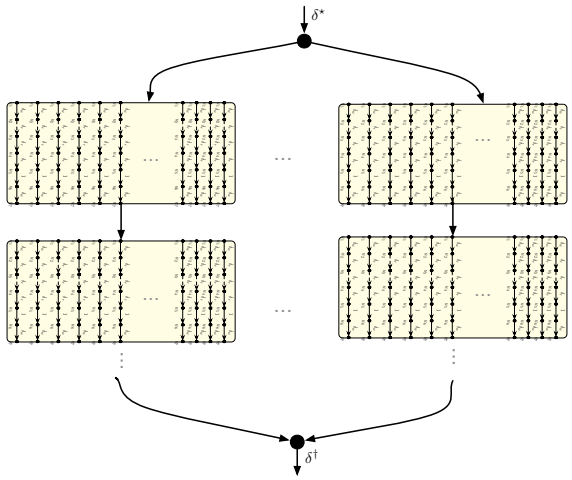


Fig. 12. Device box generated by the on in Listing 7.1. Where each thread inside the boxes is an $f\ e_i$ generated from the parmap.

7.1.1 CPU cost. On the CPU, the PRAM model tells us that the read and write costs to memory are both 1, we therefore have that $\delta_r = \delta_w = 1$, and that $f\ e_i$ has a cost of 6 time units. Based on the rules in Figure 6 we have that the **parmap** will then create the DAG in Figure 11. Analyzing this, we then determine that given an array of size n, the grayed-out code will take $\frac{4(n-1)+6n}{2}$ units of time to compute the result where the $4(n-1)$ term comes from the edges of weight 1 that are added by the $\otimes$ operator. Since we only use one region (`r0`) for the entirety of the computation, and since this is not resized, as well as since other parts of the region are not deallocated, we have that our space usage for the program will be bounded by $|$`r0`$|$ at the last step in the cost graph that we build up. We thus get that the space usage for this program is $3n$ (2 for the closure and 1 for the allocation of the result for each element of the result array).[20]

7.1.2 GPU cost. Since we will be running the code in gray on the GPU, this will generate the device box in Figure 12, which takes into account the cost of transferring memory to the GPU and the cost of merging that memory back to the CPU. We shall assume the size of our warps are w and that we have p of these per work group (warp box). Thus for a kernel on an array of size n we will have $k = n/(w * p)$ work groups. We'll use H to denote the total number of threads available on the GPU.

Global memory only. In the case where we use only global memory on the GPU, we have that each memory read and write will cost l, and therefore that $\delta_a = \delta_w = l$ and hence from Figure 10 that each thread will cost

[20]We could very well change the semantics to only create the closure once but we have decided against that for clarity.

$5 \cdot l + 1$. Since all of these will be run in parallel, we have that running the code on the GPU will have total time cost $\delta^\star(r_0) + \delta^\dagger(r_0) + (5 \cdot l + 1) \cdot (\lceil (n/H)/k \rceil)$ where the $(\lceil (n/H)/k \rceil)$ term denotes the number of 'rounds' of work groups that need to be run in the case where the size n of our vector is larger than the total number of threads H that we have on the GPU.

Using local memory. If we use local (work group shared) memory (We'll assume no bank conflicts), each read or write will cost 1 and therefore $\delta_r = \delta_w = 1$. We again have by Figure 10 that the cost per thread will then be $5 * 1 + 1$, and we then have that it will have total time cost: $\delta^\star(r_0) + \delta^\dagger(r_0) + (5 \cdot 1 + 1) \cdot (\lceil (n/H)/k \rceil) + (l * k)$. Where the $l * k$ term comes from the fact that one thread in each work group must write the results for its work group to global memory after all the threads in the work group have finished their computations.

The space usage for the GPU cases will be in the same asymptotic class as the CPU, and will take up $6n$ and $9n$ space respectively – since during the transfer process from the GPU to the CPU both regions need to be alive, and since the local version will replicate the local memory state to global memory at the end of the work group computations.

Once the user provides definitions for the primitive cost providers, they can then start calculating the total computational cost for the various options that they have tried, both in terms of computational and memory placement.

7.2 Discussion

While the example above is small and not that complex since we are working through it by hand, we don't see the main application of this theory being user-inspection based analysis of the costs of our programs. Instead, we view this theory as the core of a profiling framework similar to Spoonhower et al. (2010) where computational costs and memory usage can be automatically inferred given a set of primitive cost providers and information on the various underlying abstract machine model costs, and various scheduling policies to consider.

Viewing the theory that we have presented in this light, a profiling framework based on this theory could be used to better determine placement of computation using similar reasoning to that in the example (but automated), or after we have decided that a particular portion of the program should run on the GPU, to determine not only how we should segment the kernel generated by the computation amongst work groups, but also how we should layout work-group-shared and global memory. Moreover, such a profiling framework could be used to create scoring functions for guiding other optimizations throughout the compilation process.

8 CONCLUSIONS AND FUTURE WORK

In this paper we have developed a cost semantic theory for the profiling of heterogeneous parallel programs. In order to do this however, we have had to make a number of simplifying assumptions. In particular, we haven't considered the full power of the GPU and the other forms of memory such as constant and thread local memory that are available on it. We have also not considered the cache characteristics for our programs on the CPU. We feel as though that this would be a fairly straightforward addition and could be seen as an extension of the work by Blelloch and Harper (2013).

While the theory we have presented takes into account work-group-shared memory we do not tackle how exactly to direct the placement of memory into this work-group-shared memory when we place computations on the GPU. An interesting extension to this work would be to look at logical regions (Bauer et al. 2012) and use this work to help guide the placement of memory into local (work group) shared memory.

Another interesting avenue that remains to explore is how this theory could be used to create scoring functions both for machine learning better optimizations for programs and how to better determine placement of computations. We feel as though this work coupled with the ideas one can find in query optimization (Chaudhuri 1998) specifically as it applies to the computation graph that we build up could prove incredibly useful to both determine the efficacy of optimization techniques for heterogeneous parallel languages in general, as well as help guide the placement of computation both in the heterogeneous and distributed parallel settings.

REFERENCES

Roberto M. Amadio, Nicolas Ayache, Yann Régis-Gianas, and Ronan Saillard. 2010. *Certifying cost annotations in compilers.* Technical Report. https://hal.archives-ouvertes.fr/hal-00524715

Roberto M. Amadio and Yann Régis-Gianas. 2011. Certifying and reasoning on cost annotations of functional programs. *CoRR* abs/1110.2350 (2011). http://arxiv.org/abs/1110.2350

Martin Avanzini, Ugo Dal Lago, and Georg Moser. 2015. Analysing the Complexity of Functional Programs: Higher-order Meets First-order. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, New York, NY, USA, 152–164. DOI: http://dx.doi.org/10.1145/2784731.2784753

Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing Locality and Independence with Logical Regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, Article 66, 11 pages. http://dl.acm.org/citation.cfm?id=2388996.2389086

Ralph Benzinger. 2004. Automated Higher-order Complexity Analysis. *Theor. Comput. Sci.* 318, 1-2 (June 2004), 79–103. DOI: http://dx.doi.org/10.1016/j.tcs.2003.10.022

Guy E. Blelloch and Robert Harper. 2013. Cache and I/O Efficent Functional Algorithms. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*. ACM, New York, NY, USA, 39–50. DOI: http://dx.doi.org/10.1145/2429069.2429077

Manuel M T Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. 2011. Accelerating Haskell array codes with multicore GPUs. In *DAMP'11: Declarative Aspects of Multicore Programming*. 3–14.

Surajit Chaudhuri. 1998. An Overview of Query Optimization in Relational Systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '98)*. ACM, New York, NY, USA, 34–43. DOI: http://dx.doi.org/10.1145/275487.275492

Wei-Ngan Chin and Siau-Cheng Khoo. 2001. Calculating Sized Types. *Higher Order Symbol. Comput.* 14, 2-3 (Sept. 2001), 261–300. DOI: http://dx.doi.org/10.1023/A:1012996816178

David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. 1993. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '93)*. ACM, New York, NY, USA, 1–12. DOI: http://dx.doi.org/10.1145/155332.155333

Norman Danner, Daniel R. Licata, and Ramyaa Ramyaa. 2015. Denotational cost semantics for functional languages with inductive types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ACM, 140–151. DOI: http://dx.doi.org/10.1145/2784731.2784749

Matthew Fluet and Greg Morrisett. 2006. Monadic Regions. *J. Funct. Program.* 16, 4-5 (July 2006), 485–545. DOI: http://dx.doi.org/10.1017/S095679680600596X

Steven Fortune and James Wyllie. 1978. Parallelism in Random Access Machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing (STOC '78)*. ACM, New York, NY, USA, 114–118. DOI: http://dx.doi.org/10.1145/800133.804339

Stéphane Gimenez and Georg Moser. 2016. The Complexity of Interaction. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 243–255. DOI: http://dx.doi.org/10.1145/2837614.2837646

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2011. Multivariate Amortized Resource Analysis. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 357–370. DOI: http://dx.doi.org/10.1145/1926385.1926427

Eric Holk. 2016. *Region-based Memory Management for Expressive GPU Programming.* Ph.D. Dissertation. School of Informatics and Computer Science, Indiana University, Indiana University, Bloomington. http://blog.theincredibleholk.org/papers/dissertation.pdf

Eric Holk, Ryan Newton, Jeremy Siek, and Andrew Lumsdaine. 2014. Region-based Memory Management for GPU Programming Languages: Enabling Rich Data Structures on a Spartan Host. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 141–155. DOI: http://dx.doi.org/10.1145/2660193.2660244

John Hughes, Lars Pareto, and Amr Sabry. 1996. Proving the Correctness of Reactive Systems Using Sized Types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*. ACM, New York, NY, USA, 410–423. DOI: http://dx.doi.org/10.1145/237721.240882

Yves Lafont. 1990. Interaction Nets. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '90)*. ACM, New York, NY, USA, 95–108. DOI: http://dx.doi.org/10.1145/96709.96718

Ruy Ley-Wild, Umut A. Acar, and Matthew Fluet. 2009. A Cost Semantics for Self-adjusting Computation. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*. ACM, New York, NY, USA, 186–199. DOI: http://dx.doi.org/10.1145/1480881.1480907

Hans-Wolfgang Loidl and Kevin Hammond. 1996. A Sized Time System for a Parallel Functional Language. In *Proceedings of the Glasgow Workshop on Functional Programming*. Ullapool, Scotland.

Lu Lu, Xuanhua Shi, Yongluan Zhou, Xiong Zhang, Hai Jin, Cheng Pei, Ligang He, and Yuanzhen Geng. 2016. Lifetime-Based Memory

Management for Distributed Data Processing Systems. *CoRR* abs/1602.01959 (2016). http://arxiv.org/abs/1602.01959

Trevor L. McDonell, Manuel M T Chakravarty, Gabriele Keller, and Ben Lippmeier. 2013. Optimising Purely Functional GPU Programs. In *ICFP'13: International Conference on Functional Programming*. 49–60.

Mozilla Research. 2010. The Rust Programming Languge. https://www.rust-lang.org. (2010). Accessed: 2017-01-10.

K. Nakano. 2012. Simple Memory Machine Models for GPUs. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*. 794–803. DOI: http://dx.doi.org/10.1109/IPDPSW.2012.98

John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable Parallel Programming with CUDA. *Queue* 6, 2 (March 2008), 40–53. DOI: http://dx.doi.org/10.1145/1365490.1365500

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 519–530. DOI: http://dx.doi.org/10.1145/2491956.2462176

David Sands. 1990. Complexity Analysis for a Lazy Higher-Order Language. In *In Proceedings of the 3rd European Symposium on Programming*. Springer-Verlag, 361–376.

Patrick M. Sansom and Simon L. Peyton Jones. 1995. Time and Space Profiling for Non-strict, Higher-order Functional Languages. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. ACM, New York, NY, USA, 355–366. DOI: http://dx.doi.org/10.1145/199448.199531

Patrick M. Sansom and Patrick M. Sansom. 1994. *Execution Profiling for Non-strict Functional Languages*. Technical Report. University of Glasgow.

Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. 2010. Space profiling for parallel functional programs. *Journal of Functional Programming* 20 (11 2010), 417–461. Issue Special Issue 5-6. DOI: http://dx.doi.org/10.1017/S0956796810000146

Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating Performance Portable Code Using Rewrite Rules: From High-level Functional Expressions to High-performance OpenCL Code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, New York, NY, USA, 205–217. DOI: http://dx.doi.org/10.1145/2784731.2784754

John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *IEEE Des. Test* 12, 3 (May 2010), 66–73. DOI: http://dx.doi.org/10.1109/MCSE.2010.69

Mads Tofte and Jean-Pierre Talpin. 1997. Region-based memory management. *Information and computation* 132, 2 (1997), 109–176.

Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (Aug. 1990), 103–111. DOI: http://dx.doi.org/10.1145/79173.79181

Kathryn Van Stone. 2003. *A Denotational Approach to Measuring Complexity in Functional Programs*. Ph.D. Dissertation. Carnegie Mellon University, Pittsburgh, PA, USA. Advisor(s) Brookes, Stephen. AAI3262826.

Ben Van Werkhoven, Jason Maassen, Frank J Seinstra, and Henri E Bal. 2014. Performance models for CPU-GPU data transfers. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*. IEEE, 11–20.

Pedro B Vasconcelos. 2008. *Space cost analysis using sized types*. Ph.D. Dissertation. University of St Andrews.

Edward Z Yang, Giovanni Campagna, Ömer S Ağacan, Ahmed El-Hassany, Abhishek Kulkarni, and Ryan R Newton. 2015. Efficient communication and collection with compact normal forms. *ACM SIGPLAN Notices* 50, 9 (2015), 362–374.

A TYPE SYSTEM

In this section we detail the type and region system for $\lambda_{\mathcal{G}}$. This takes the form of a bounded region calculus similar to that found in Fluet and Morrisett (2006), which allows us to elide effect polymorphism in our type system. The type system is also designed to statically enforce the constraints that regions cannot be created on the GPU, and that we cannot create more GPU computations while we are on the GPU; essentially, we enforce that memory and computations can only be "pushed" from the CPU to the GPU, and "pulled" back from the GPU to the CPU.

The various typing judgements for our type and region system can be found in Figure 14, and Figure 2. Region contexts Ω are ordered lists of region variables, and region constraint contexts Δ are ordered lists of region constraints. Value contexts Γ are unordered lists of variables and types. The well-formedness rules for region contexts, typing contexts, and effects are given in Figures 15, 16, and 17. We summarize the various typing and context and effect well-formedness judgements in the following table:

Judgement	Meaning
$\vdash_{\text{reg}} \Omega$	Region context Ω is well-formed
$\Omega \vdash_{\text{rctxt}} \Delta$	Region constraint context Δ is well-formed
$\Omega; \Delta \vdash_{\text{vctxt}} \Gamma$	Typing context Γ is well-formed
$\vdash_{\text{ctxt}} \Omega; \Delta; \Gamma; \varphi$	In valid contexts Ω, Δ, and Γ, the latent effect φ is well-formed.
$\Omega; \Delta \vdash_{\text{btype}} \mu$	The boxed type μ is well-formed
$\Omega \vdash_{\text{place}} \rho_\omega$	The region ρ is valid region variable in Ω
$\Omega; \Delta \vdash_{\text{type}} \tau$	Type τ is well-formed
$\Omega; \Delta \vdash_{\text{eff}} \varphi$	In region context Ω and region constraints Δ, the latent effect φ is well-formed
$\Omega; \Delta \vdash_{\text{rr}} \rho \geq \rho'$	If region ρ is alive, then ρ' is live as well (region ρ' outlive ρ)
$\Omega; \Delta \vdash_{\text{re}} \rho \geq \varphi$	If region ρ is alive, then all regions in φ are alive (all regions in φ outlive ρ).
$\Omega; \Delta \vdash_{\text{er}} \varphi \ni \rho_\omega$	The region ρ_ω is a region in the latent effect φ
$\Omega; \Delta \vdash_{\text{ee}} \varphi \supseteq \varphi'$	The latent effect set φ' is a subset of the latent effect φ
$\Delta; \Gamma \vdash^{\omega}_{\text{exp}} e : \tau, \theta$	Term e running on world ω has type τ and effects bounded by region θ

$$i \in \mathbb{Z} \mid r \in RNames \mid f, x \in Vars \mid \rho_\omega \in RVars \mid o \in \mathbb{N}_{bounded}$$

ℓ	::=	(r, o)	Region Locations
R	::=	$(\mathsf{ap}, s, \mathtt{refcnt}, \{o_1 \mapsto v_1, \ldots, o_n \mapsto v_n\})$	Regions
$\mathcal{S}$	::=	$\cdot \mid \mathcal{S}, r \mapsto R$	Ordered Domain/Region Stack
φ	::=	$\{\rho_1, \ldots, \rho_n\}$	Effect Sets
ω	::=	$\mathbb{C} \mid \mathbb{G}$	Computation Worlds
$\mathcal{T}$	::=	$\{\mathbb{C} \mapsto \mathcal{S}_1, \mathbb{G} \mapsto \mathcal{S}_2\}$	Configuration Tables
τ	::=	$\mathbf{bool} \mid (\mu, \rho)$	Types
μ	::=	$\mathbf{int} \mid \tau_1 \xrightarrow{\varphi} \tau_2 \mid \tau_1 \times \tau_2 \mid \mathbf{Vec}\ \tau$	Boxed Types
Ω	::=	$\{\overline{\rho}\} \mid \rho_\omega, \Omega$	World Contexts
Δ	::=	$\cdot \mid \Delta, \rho_\omega \geq \varphi$	Region Constraints
Γ	::=	$\cdot \mid \Gamma, x : \tau$	Typing Contexts
e	::=	$i\ \mathbf{at}\ \rho \mid \mathbf{True} \mid \mathbf{False}$	Terms
	|	$e_1 * e_2 \mid e_1 < e_2$	
	|	$\mathbf{if}\ e\ \mathbf{then}\ e_t\ \mathbf{else}\ e_f \mid x$	
	|	$\lambda x : \tau.^{\varphi} e\ \mathbf{at}\ \rho \mid e_1\ e_2$	
	|	$(e_1, e_2)\ \mathbf{at}\ \rho \mid \mathbf{fst}\ e \mid \mathbf{snd}\ e$	
	|	$\mathbf{letregion}\ \rho\ \mathbf{in}\ e \mid \mathbf{fix}\ f : \tau.u \mid e_1 \| e_2\ \mathbf{at}\ \rho$	
	|	$[e_1, \ldots, e_n]\ \mathbf{at}\ \rho \mid \mathbf{parmap}\ f\ [e_1, \ldots, e_n]\ \mathbf{at}\ \rho$	
	|	$\mathbf{on}\ e\ \mathbf{at}\ \rho$	
	|	$e[\rho] \mid \lambda \rho \geq \varphi.^{\varphi'} u\ \mathbf{at}\ \rho$	
v	::=	$i \mid b \mid (v_1, v_2) \mid [v_1, \ldots, v_n]$	Heap Values
	|	$\mathbf{clos}\ (x, e, E) \mid \ell$	

Fig. 13. Full Syntax of $\lambda_{\mathcal{G}}$

$$\boxed{\Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} e : \tau, \varphi}$$

$$\frac{\vdash_{\text{ctxt}} \Omega;\Delta;\Gamma;\varphi \quad \Omega \vdash_{\text{place}} \rho_\omega \quad \Omega;\Delta \vdash_{\text{er}} \varphi \ni \rho_\omega}{\Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} i \ \texttt{at}\ \rho_\omega : (\ \texttt{int}\ , \rho_\omega), \varphi}$$

$$\frac{\Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} e_1 : (\ \texttt{int}\ , \rho^1_\omega), \varphi \quad \Omega;\Delta \vdash_{\text{er}} \varphi \ni \rho^1_\omega \quad \Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} e_2 : (\ \texttt{int}\ , \rho^2_\omega), \varphi \quad \Omega;\Delta \vdash_{\text{er}} \varphi \ni \rho^2_\omega \quad \Omega \vdash_{\text{place}} \rho_\omega \quad \Delta \vdash_{\text{er}} \varphi \ni \rho_\omega}{\Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} e_1 * e_2 \ \texttt{at}\ \rho_\omega : (\ \texttt{int}\ , \rho_\omega), \varphi}$$

$$\frac{\Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} e_1 : (\ \texttt{int}\ , \rho^1_\omega), \varphi \quad \Omega;\Delta \vdash_{\text{er}} \varphi \ni \rho^1_\omega \quad \Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} e_2 : (\ \texttt{int}\ , \rho^2_\omega), \varphi \quad \Omega;\Delta \vdash_{\text{er}} \varphi \ni \rho^2_\omega}{\Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} e_1 < e_2 : \ \texttt{bool}\ , \varphi}$$

$$\frac{\vdash_{\text{ctxt}} \Omega;\Delta;\Gamma;\varphi}{\Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} \ \texttt{False}\ : \ \texttt{bool}\ , \varphi} \qquad \frac{\vdash_{\text{ctxt}} \Omega;\Delta;\Gamma;\varphi}{\Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} \ \texttt{True}\ : \ \texttt{bool}\ , \varphi}$$

$$\frac{\Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} e_b : \ \texttt{bool}\ , \varphi \quad \Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} e_t : \tau, \varphi \quad \Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} e_f : \tau, \varphi}{\Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} \ \texttt{if}\ e_b \ \texttt{then}\ e_t \ \texttt{else}\ e_f : \tau, \varphi} \qquad \frac{\vdash_{\text{ctxt}} \Omega;\Delta;\Gamma;\varphi \quad x \in dom(\Gamma) \quad \Gamma(x) = \tau}{\Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} x : \tau, \varphi}$$

$$\frac{\Omega;\Delta;\Gamma, x : \tau_1 \vdash^{\omega}_{\text{exp}} e' : \tau_2, \varphi' \quad \Omega \vdash_{\text{place}} \rho_\omega \quad \Omega;\Delta \vdash_{\text{er}} \varphi \ni \rho_\omega}{\Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} \lambda x : \tau_1.^{\varphi'} e' \ \texttt{at}\ \rho_\omega : (\tau_1 \xrightarrow{\varphi'} \tau_2, \rho_\omega), \varphi} \qquad \frac{\Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} e_1 : (\tau_1 \xrightarrow{\varphi'} \tau_2, \rho_\omega), \varphi \quad \Omega;\Delta \vdash_{\text{er}} \varphi \ni \rho_\omega \quad \Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} e_2 : \tau_1, \varphi \quad \Omega;\Delta \vdash_{\text{ee}} \varphi \supseteq \varphi'}{\Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} e_1\ e_2 : \tau_2, \varphi}$$

$$\frac{\Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} e_1 : \tau_1, \varphi \quad \Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} e_2 : \tau_2, \varphi \quad \Omega \vdash_{\text{place}} \rho_\omega \quad \Omega;\Delta \vdash_{\text{er}} \varphi \ni \rho_\omega}{\Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} (e_1, e_2) \ \texttt{at}\ \rho_\omega : (\tau_1 \times \tau_2, \rho_\omega), \varphi} \qquad \frac{\Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} e : (\tau_1 \times \tau_2, \rho_\omega), \varphi \quad \Omega;\Delta \vdash_{\text{er}} \varphi \ni \rho_\omega}{\Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} \ \texttt{fst}\ e : \tau_1, \varphi}$$

$$\frac{\Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} e : (\tau_1 \times \tau_2, \rho_\omega), \varphi \quad \Omega;\Delta \vdash_{\text{er}} \varphi \ni \rho_\omega}{\Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} \ \texttt{snd}\ e : \tau_2, \varphi} \qquad \frac{\Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} e_i : \tau, \varphi \quad \Omega \vdash_{\text{place}} \rho_\omega \quad \Omega;\Delta \vdash_{\text{er}} \varphi \ni \rho_\omega}{\Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} [e_1, \ldots, e_n] \ \texttt{at}\ \rho_\omega : (\ \texttt{Vec}\ \tau, \rho_\omega), \varphi}$$

$$\frac{\rho_\omega, \Omega;\Delta, \rho_\omega \succeq \varphi'';\Gamma \vdash^{\omega}_{\text{exp}} u' : \tau, \varphi' \quad \Omega \vdash_{\text{place}} \rho_\omega \quad \Omega;\Delta \vdash_{\text{er}} \varphi \ni \rho_\omega}{\Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} \lambda\ \rho_\omega \succeq \varphi''.^{\varphi'} u' \ \texttt{at}\ \rho_\omega : (\Pi \rho_\omega \succeq \varphi''.^{\varphi'} \tau, \rho_\omega), \varphi}$$

$$\frac{\Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} e : (\Pi \rho_\omega \succeq \varphi''.^{\varphi'} \tau, \rho^1_\omega), \varphi \quad \Omega;\Delta \vdash_{\text{er}} \varphi \ni \rho^1_\omega \quad \Omega \vdash_{\text{place}} \rho^2_\omega \quad \Omega;\Delta \vdash_{\text{re}} \rho^2_\omega \succeq \varphi'' \quad \Omega;\Delta \vdash_{\text{ee}} \varphi \supseteq \varphi'[\rho^2_\omega/\rho_\omega]}{\Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} e[\rho^2_\omega] : \tau[\rho^2_\omega/\rho_\omega], \varphi} \qquad \frac{\Omega;\Delta;\Gamma, f : \tau \vdash^{\omega}_{\text{exp}} u : \tau, \varphi}{\Omega;\Delta;\Gamma \vdash^{\omega}_{\text{exp}} \ \texttt{fix}\ f : \tau.u : \tau, \varphi}$$

Fig. 14. The base typing rules for $\lambda_{\mathcal{G}}$based on the Bounded Effect Calculus of Fluet and Morrisett (2006)

$$\boxed{\Omega;\Delta \vdash_{\text{rr}} \rho_\omega \succeq \rho'_\omega}$$

$$\frac{\Omega \vdash_{\text{rctxt}} \Delta \qquad \Omega \vdash_{\text{place}} \rho_\omega \qquad \Delta \vdash_{\text{re}} \rho_\omega \succeq \{\rho^1_\omega,\ldots,\rho^i_\omega,\ldots,\rho^n_\omega\}}{\Omega;\Delta \vdash_{\text{rr}} \rho_\omega \succeq \rho^i_\omega} \qquad \frac{\Omega \vdash_{\text{place}} \rho_\omega}{\Delta \vdash_{\text{rr}} \rho_\omega \succeq \rho_\omega} \qquad \frac{\Omega;\Delta \vdash_{\text{rr}} \rho_\omega \succeq \rho'_\omega \qquad \Omega;\Delta \vdash_{\text{rr}} \rho'_\omega \succeq \rho''_\omega}{\Omega;\Delta \vdash_{\text{rr}} \rho_\omega \succeq \rho''_\omega}$$

$$\boxed{\Omega;\Delta \vdash_{\text{re}} \rho_\omega \succeq \varphi} \qquad \boxed{\Omega;\Delta \vdash_{\text{er}} \varphi \ni \rho_\omega} \qquad \boxed{\Omega;\Delta \vdash_{\text{ee}} \varphi \supseteq \varphi'}$$

$$\frac{\Omega \vdash_{\text{rctxt}} \Delta \qquad \Omega;\Delta \vdash_{\text{rr}} \rho_\omega \succeq \rho^i_\omega \quad i=1,\ldots,n}{\Omega;\Delta \vdash_{\text{re}} \rho_\omega \succeq \{\rho^1_\omega,\ldots,\rho^n_\omega\}} \qquad \frac{\Omega \vdash_{\text{place}} \rho^i_\omega \qquad \Omega;\Delta \vdash_{\text{eff}} \{\rho^1_\omega,\ldots,\rho^n_\omega\}}{\Omega;\Delta \vdash_{\text{er}} \{\rho^1_\omega,\ldots,\rho^n_\omega\} \ni \rho^i_\omega} \qquad \frac{\Omega;\Delta \vdash_{\text{eff}} \varphi \qquad \Omega;\Delta \vdash_{\text{er}} \varphi \ni \rho^i_\omega}{\Omega;\Delta \vdash_{\text{ee}} \varphi \supseteq \{\rho^i_\omega\}^n_{i=1}}$$

Fig. 15. Well formedness rules for regions and effects

$$\boxed{\Omega;\Delta \vdash_{\text{btype}} \mu}$$

$$\frac{\Omega \vdash_{\text{rctxt}} \Delta}{\Omega;\Delta \vdash_{\text{btype}} \textbf{int}} \qquad \frac{\Omega;\Delta \vdash_{\text{type}} \tau_1 \qquad \Omega;\Delta \vdash_{\text{eff}} \varphi \qquad \Omega;\Delta \vdash_{\text{type}} \tau_2}{\Omega;\Delta \vdash_{\text{btype}} \tau_1 \xrightarrow{\varphi} \tau_2} \qquad \frac{\Omega;\Delta \vdash_{\text{type}} \tau_1 \qquad \Omega;\Delta \vdash_{\text{type}} \tau_2}{\Omega;\Delta \vdash_{\text{btype}} \tau_1 \times \tau_2}$$

$$\frac{\Omega;\Delta \vdash_{\text{eff}} \varphi' \qquad \rho_\omega,\Omega;\Delta,\rho_\omega \succeq \varphi' \vdash_{\text{eff}} \varphi \qquad \rho_\omega,\Omega;\Delta,\rho_\omega \succeq \varphi' \vdash_{\text{type}} \tau}{\Omega;\Delta \vdash_{\text{btype}} \Pi\rho_\omega \succeq \varphi'.^{\varphi}\tau}$$

$$\boxed{\Omega \vdash_{\text{place}} \rho_\omega} \qquad \boxed{\Omega;\Delta \vdash_{\text{type}} \tau} \qquad \boxed{\Omega;\Delta \vdash_{\text{eff}} \varphi}$$

$$\frac{\rho_\omega \in \Omega}{\Omega \vdash_{\text{place}} \rho_\omega} \qquad \frac{\Omega \vdash_{\text{rctxt}} \Delta}{\Omega;\Delta \vdash_{\text{type}} \textbf{bool}} \qquad \frac{\Omega;\Delta \vdash_{\text{btype}} \mu \qquad \Omega \vdash_{\text{place}} \rho_\omega}{\Omega;\Delta \vdash_{\text{type}} (\mu,\rho_\omega)} \qquad \frac{\Omega \vdash_{\text{rctxt}} \Delta \qquad \Omega \vdash_{\text{place}} \rho^i_\omega}{\Omega;\Delta \vdash_{\text{eff}} \{\rho^i_\omega\}^n_{i=1}}$$

Fig. 16. Well formedness rules for types

$$\boxed{\Omega \vdash_{\text{rctxt}} \Delta} \qquad \boxed{\Omega;\Delta \vdash_{\text{vctxt}} \Gamma}$$

$$\frac{}{\vdash_{\text{rctxt}} \cdot} \qquad \frac{\Omega \vdash_{\text{rctxt}} \Delta \qquad \rho_\omega \notin dom(\Delta) \qquad \Omega \vdash_{\text{place}} \rho_\omega \qquad \Omega;\Delta \vdash_{\text{eff}} \varphi}{\Omega \vdash_{\text{rctxt}} \Delta,\rho_\omega \succeq \varphi} \qquad \frac{\Omega \vdash_{\text{rctxt}} \Delta}{\Omega;\Delta \vdash_{\text{vctxt}} \cdot} \qquad \frac{\Omega;\Delta \vdash_{\text{vctxt}} \Gamma \qquad \Omega;\Delta \vdash_{\text{type}} \tau \qquad x \notin dom(\Gamma)}{\Omega;\Delta \vdash_{\text{vctxt}} \Gamma,x:\tau}$$

$$\boxed{\vdash_{\text{reg}} \Omega} \qquad \boxed{\vdash_{\text{ctxt}} \Omega;\Delta;\Gamma;\varphi}$$

$$\frac{}{\vdash_{\text{reg}} \{\underline{\rho_\omega}\}} \qquad \frac{\vdash_{\text{reg}} \Omega \qquad \rho_\omega \in RVars}{\vdash_{\text{reg}} \rho_\omega,\Omega} \qquad \frac{\Omega;\Delta \vdash_{\text{vctxt}} \Gamma \qquad \Omega;\Delta \vdash_{\text{eff}} \varphi}{\vdash_{\text{ctxt}} \Omega;\Gamma;\varphi}$$

Fig. 17. Well formedness rules for contexts