# Hybrid.Parallel

## Light-Weight Programming of Hybrid Systems

Frank Feinbube, Jan-Arne Sobania, Peter Tröger, Andreas Polze

*Hasso Plattner Institute, University of Potsdam*
*Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany*

{frank.feinbube, jan-arne.sobania, peter.troeger, andreas.polze}@hpi.uni-potsdam.de

***Abstract-The advent of homogeneous many-core processors has been widely noticed as a major revolution in computer architecture. It has influenced the design of operating systems, programming models and parallelization libraries.***
***Future system designs are expected to combine many-core processors with graphical processing units or other special purpose accelerators to hybrid systems. In contrast to the established programming languages and tools for standard processors, programming for accelerators and hybrid systems is still in its infancy. The programmer explicitly has to address vendor-specific and version-specific characteristics of the particular device.***

***Refactoring existing parallel code to run in such an environment becomes a tedious and error-prone task.***

***We present our approach for automatically transforming high-level parallel code written in .NET to parallel code suitable for hybrid systems.***

***Our approach relies on the transformation of .NET byte code into accelerator-neutral source code, which makes the concept applicable to all .NET-supported programming languages and different accelerator types. We demonstrate the feasibility of our solution with various well-known compute-intense algorithms.***

***Parallel Programming, Application Software, Accelerator architectures***

### I. INTRODUCTION

The ever-increasing transistor count according to Moore's law is currently used for adding more computational cores in each new processor generation. Hardware vendors try to integrate the increasing amount of cores in two different ways. The first is to design uniform general purpose many-core processors. The second is to introduce special purpose co-processors, also called *accelerators* that are capable of executing a restricted class of parallel tasks very fast and energy efficient. Systems built as a mixture of general purpose processors and accelerators are called *hybrid systems*, and represent a more and more common form of system design.

Accelerators used in hybrid system design cannot only be found in supercomputers, but also in business servers, desktop computers and even mobile computers. They are energy efficient and provide great computing powers for their special purpose usage. The accelerator type that has the best market penetration is the *graphics processing unit (GPU)*. Since GPU compute devices are built into many computer systems and have outstanding performance for various problem classes [9][20], they are ideal candidates to be also exploited for other kinds of applications, but rendering.

Parallel programming for many-core processors has lately become a major issue for all developers. Performance penalties for not applying parallelism are so huge on modern hardware architectures that they cannot be ignored any longer.

Dealing with hybrid systems can be considered an additional burden for the developers. Programmers do not only have to think about algorithmic parallelism, but also about the different capabilities of the processors and accelerators in charge [17][25]. Several challenges make GPU programming a painful task. It is much harder to get acceptable performance for applications running on GPU compute devices than to port them there in the first place [22]. GPU programming is also not supported for many widespread programming languages. Cutting-edge APIs like CUDA [8] and OpenCL [24] are only supported for C/C++ and Fortran.

From a researcher's perspective, *general-purpose computing on graphics processing units (GPU Computing)* still represents a promising opportunity for hybrid computing. The combination of fundamentally different parallel hardware in one system gives the chance to optimize the execution of unbalanced and diverse parallel load. For this reason, and with the GPU as the current default accelerator, we will investigate on the following pages how high-level developers can be enabled to use heterogeneous execution environments in a seamless and transparent fashion.

### II. MOTIVATION

A common problem with new programming paradigms is the need to adapt existing code to new language features or libraries. It is not feasible to rewrite all existing programs from scratch, therefore refactoring is used to transform existing code into a representation that is aware of the new paradigm.

Due to the rise of multi-core computing, old and new research results in the area of parallel programming paradigms gained new attention. Libraries, language extensions and even new languages are developed, all aiming to provide good performance, scalable resource usage and good programmability.

In contrast to the widespread consideration of many-core development, support for developers targeting hybrid systems is not mature yet. Since most accelerators are parallel processors, this domain shares the same problems as the parallel programming domain. But in addition, different execution properties and hardware capabilities also need to be considered – not only for accelerators of different kinds or built by different vendors, but even for different generations of the same product line [17][25].

We aim to improve this situation by allowing developers to write code in their preferred language and enabling execution of this code on hybrid systems in an automated fashion. Our objectives are as follows:

1. Support for widely used programming languages.
2. A light-weight programming paradigm to lower the entry barrier for developers.

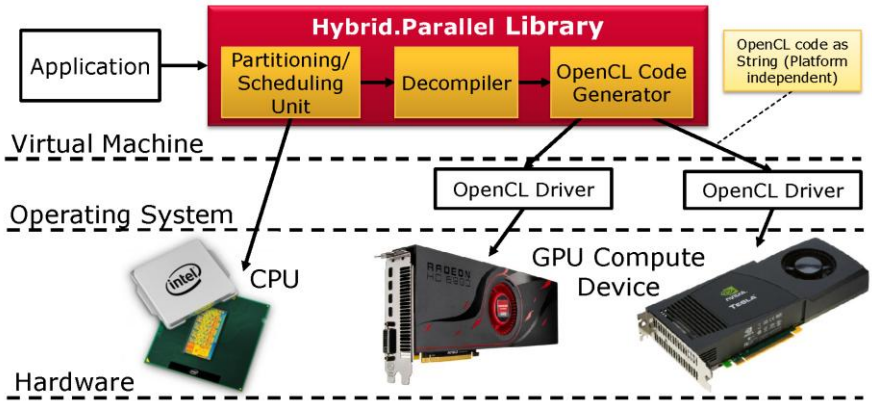


Fig. 1: Architecture Overview

3. Support refactoring, as well as, writing new code from scratch.
4. Support a wide range of accelerator devices – not bound to a particular device vendor.
5. Provide a framework that allows automatic and semi-automatic optimizations.

## III. Approach

We decided to realize our approach for the .NET Framework to be able to support C#, Visual Basic, and a number of other programming languages. Instead of introducing language or compiler extensions, we chose to provide a library, which allows developers to keep the tool chain of the development process unaltered. This enables a seamless integration into existing .NET applications. We target accelerator hardware using the broadly accepted OpenCL API standard.

In order to address the second of our objectives, we decided to utilize a parallel loop construct. This is not only a light-weight and widespread construct for parallel computing [6][21], but also familiar to .NET developers as part of the *Task Parallel Library (TPL)*. The following code example shows how our library can be accessed by an implementation:

Code Example I
Original code vs. Hybrid.Parallel

```
int[] data = ...  // Get vector

// Actions
Action<int> action = delegate(int i)
{
    data[i] = data[i] + 1;
};
Action<int> action2D = delegate(int x, int y)
{ /* ... */ };

// Microsoft's solution
// based on the Task Parallel Library
Threading.Tasks.Parallel.For(0, dataSize, action);

// Our Library
Hybrid.Parallel.For(0, dataSize, action);
Hybrid.Parallel.For2D(0, dataSizeX, 0, dataSizeY,
    action2D);
```

Our implementation of *Parallel.For* accepts the same parameters as the original, so that refactoring only requires the exchange of one namespace declaration. In addition, we define a two-dimensional version of the interface that can substitute two *for* loops and allows better mapping onto GPU hardware which is generally better suited for more compact code fragments.

Since our approach looks equal to the existing solution for multi-processor-based parallelism, refactoring can be applied without additional effort. Besides the implicit automatic execution mode, we also allow experimenting with running parts of the code on CPUs and comparing the execution characteristics against running them on GPUs explicitly. The following code shows the explicit functions that can be used for this:

Code Example II
Explicit execution on GPU compute devices and CPUs

```
// use GPU / CPUs only - explicit version
Hybrid.Parallel.ForGpu(0,size,action);
Hybrid.Parallel.ForCpus(0,size,action);
```

Our fourth objective, accelerator independence, can be achieved by using a corresponding vendor- and device-independent programming interface. The fact that OpenCL is designed to address various hardware architectures, for example GPU compute devices, Cell processors, and general purpose processors, makes it the natural choice for our implementation.

Currently we only focus on CPU and GPU computing, but since our implementation is based on OpenCL, we already support other OpenCL-enabled accelerators. Also, our code generation system can be adapted for other accelerator platforms in the future.

The code generator enables the seamless integration of automatic and semi-automatic optimizations. Existing projects have demonstrated that selected GPU optimization mechanisms can be applied automatically [2]. Since we are working with byte code targeting a virtual machine execution environment, we can optimize based on results from code reflection and meta data checking. In addition, we can inspect runtime characteristics dynamically to adjust scheduling decisions.

The following section describes our software infrastructure supporting this approach.

## IV. Implementation

The architecture of our hybrid computing support library is shown in Fig. 1. The developer implements a parallelized .NET application, based on the support functionalities in the .NET *Parallel* namespace. For refactoring (or development from

scratch), the implementation is linked with our implementation of the *Parallel* namespace for hybrid environments. When the application code calls our library at runtime, first the partitioning and scheduling unit gets active. It implements the decision for an optimal runtime environment with respect to the given parallelized code block.

### A. Partitioning and Scheduling Unit

At startup, the library investigates the system configuration and collects information about all available computing devices. After the call of *Hybrid.Parallel.For*, we analyze the characteristics of the algorithm that is provided. Two major properties must be considered: capability of execution and anticipated performance.

Based on the instruction mix we assess which devices are capable of the execution. Recursion is not yet supported by GPU Compute Devices; atomic functions and double precision floating point operations are not supported in the early generations of GPU Compute Devices.

Table I shows the characteristics of algorithms and devices that we use to predict the execution performance. We schedule the execution based on the prediction on capable devices. The scheduler also measures the time that the actual execution needed and uses this information to improve the prediction for the next run.

TABLE I
PERFORMANCE CRITICAL CHARACTERISTICS

| Good performance on GPU | Good performance on CPU |
|---|---|
| **Algorithms characteristics** | |
| Lots of arithmetic operations | Lots of control flow statements |
| Small amount of registers/memory, many calculations | Huge amount of memory, small amount of calculations |
| Many single precision floating point operations | |
| **Device characteristics** | |
| Large number of multiprocessors | High number of cores |
| High frequency of multiprocessors, memory, … | High frequency of cores |
| Support for caching | Big caches |
| Big amount of registers / concurrent threads | Support for instruction level parallelism |

Developers can check if their code will run on a specific device type at runtime and get an exception if they force the execution on a device type that is not capable of running the provided code.

If the code is to be executed on CPUs, we forward it to Microsoft's *Parallel.For* implementation. Otherwise we transform it into an OpenCL-conformant representation and execute it on suitable accelerators as described in the following section.

### B. Decompiler

Compilers for the .NET platform produce binaries in a language called *Microsoft Intermediate Language (MSIL)*, which has been standardized under the name *Common Intermediate Language (CIL)* as ECMA-335 [14]. It is comparable to Java byte code [12] both in its expressiveness and level of abstraction.

OpenCL uses a subset of the C programming language with some extensions to describe tasks to be executed on an accelerator. To execute CIL code on such a device, we need to translate CIL into equivalent OpenCL C code in a process commonly known as decompilation [19].

Once a program invokes *Hybrid.Parallel.For* and the Partitioning and Scheduling Unit has decided to run a code fragment on an accelerator, the following activities take place at runtime:

1) The fragment to be executed is decompiled into a High-Level Intermediate Representation (HLIR), an abstract representation which is similar to the one used by a compiler.
2) This representation is then transformed to the C dialect of OpenCL. The transformation also includes mapping of .NET/CIL-specific features. (Section C)
3) Once C code is available, we use OpenCL.NET [3] – a lightweight OpenCL wrapper for .NET – to instruct the OpenCL library and device drivers to compile and execute the code on a compute device. (Section D)

Our implementation uses a cache for generated OpenCL programs. In cases when the same fragment is invoked multiple times, the first two steps can be skipped and the precompiled program can be executed directly.

#### 1) Representation of fragments in .NET

To illustrate the steps of the decompiler, we use the program fragment from code example I as an example, which simply increments each element of a one-dimensional array. This example uses an *anonymous delegate*: the code block incrementing the array elements is not a regular (named) method of a class, but specified via the *delegate* keyword. This feature is implemented in the C# compiler, but not the runtime: CIL itself does not support anonymous delegates [14], so internally, the compiler creates a method containing only the code of the delegate block, then passes a reference to this method to the implementation.

To get access to the code block, the implementation uses the passed object of the `Action<int>` delegate class. This object represents a function pointer in .NET, so it could be invoked directly if the code is to run on the CPU. However, the object also provides all necessary information for decompilation.

As shown in code example I, the delegate accesses two variables: the *data* array and the parameter *i* holding the array index. As `Action<int>` delegates only support one parameter of type *int*, the array has to be passed via another mechanism; the same holds for more complex examples that access variables declared in the method that contains the *ForGpu* call.

For all these cases, the C# compiler dynamically creates classes that are containers for needed variables; i.e., closures. Each local variable or parameter of the method contained in the *Hybrid.Parallel.For* call is mapped to a field in such a class; in case the delegate block also accesses fields via its parent's *this* pointer, the generated class will get a reference to that object as well. However, we have also observed a specific compiler optimization: if no local variable or parameter is accessed, creation of the additional class is skipped, and the delegate block is simply added as another method to the parent method's class itself.

Since our example needs access to a local variable (the *data* array), the compiler will create a helper class, then add the delegate block as a method. The CIL code of this method,

which our implementation needs to convert to OpenCL C, is shown in the following code example:

CODE EXAMPLE III
DECOMPILATION EXAMPLE: DELEGATE BLOCK (CIL) AND HLIR MAPPING

```
  .method public hidebysig instance void
'<algorithm>b__0'(int32 i) cil managed
  {
    // Code size       20 (0x14)
    .maxstack  8
    IL_0000: nop
       // do nothing
    IL_0001: ldarg.0
       // cil_stack_0 = this
    IL_0002: ldfld int32[] ...::data
       // cil_stack_0 = cil_stack_0.data
    IL_0007: ldarg.1
       // cil_stack_1 = i
    IL_0008: ldarg.0
       // cil_stack_2 = this
    IL_0009: ldfld int32[] ...::data
       // cil_stack_2 = cil_stack_2.data
    IL_000e: ldarg.1
       // cil_stack_3 = i
    IL_000f: ldelem.i4
       // cil_stack_2 = cil_stack_2[cil_stack_3]
    IL_0010: ldc.i4.1
       // cil_stack_3 = 1
    IL_0011: add
       // cil_stack_2 = cil_stack_2 + cil_stack_3
    IL_0012: stelem.i4
       // cil_stack_0[cil_stack_1] = cil_stack_2
    IL_0013: ret
       // return
  }// end of '<>c__DisplayClass4'::'<algorithm>b__0'
```

*2) Rewriting CIL into high-level expressions*

The CIL execution model resembles that of a stack machine [14]. All arguments and results of operations are stored on the stack and there are explicit instructions for loading and storing data in memory.

Each value on the stack has a specific type, which can be either an integral (numeric, pointer, object reference) or complex type (instance of a value type like a C# *struct*). In this model, a single ``value" at the abstraction level of the programming language is always contained in a single stack location (i.e., a stack location is never “too small” to hold a value).

Most instructions consume and produce a fixed number of values; the only exception is instructions like *ret* or *call*, whose number of arguments or results depends on the respective target method's signature. A *call* consumes as many values as the target has arguments (including the *this* reference for instance methods), and produces zero or one values depending on whether its return type is *void*. Likewise, *ret* consumes zero or one values, depending on the return type of the method it appears in. Regardless of the instruction, the number of values it consumes and produces is statically known and does not change while the program is running.

When CIL code is loaded by the runtime, certain checks are performed to make sure the program's code is syntactically correct and usage of methods (like the type of arguments passed in a call) is consistent to their declaration [14]. Especially, the number of values on the execution stack as well as their respective types can be determined statically for each instruction. This also holds for control flow merge points (e.g., an instruction that can be reached from its preceding one as well as via a branch), at which point the final stack state is calculated by *merging* all incoming stack states: the number of values needs to be identical, and the result type of each stack location is the “closest common supertype” [14] of the types from all incoming edges.

Due to this representation, re-writing CIL instructions into expressions resembling those of a high-level language is quite trivial. For each value that gets pushed onto the evaluation stack, we define a local variable and insert a reference to that variable when it is needed later. Variables are named relating to their position in the stack (e.g., `cil_stack_`*id* for the variable at stack location *id*); a definition of a variable overrides all previous definitions of the same name.

The comments of code example III show the resulting representation of each instruction after this transformation has been applied to the example delegate block. As the data type of each value is statically known, high-level operations (like accessing the *data* field) are well-defined.

CIL code for more complex methods may also contain branch instructions, so a simple list of high-level instructions is not sufficient to represent all input programs. Instead, our CIL decompiler uses *control flow graphs* [1] to break the instruction stream into sequential basic blocks. Control dependencies between these blocks are modeled as graph edges. The example shows one of these basic blocks.

*3) Optimizing generated expressions*

For actual code generation from this representation, the newly-generated local variables would need to be given unique names within the routine, while paying attention to assigning the correct follow-up variables on control-flow merge points.

The main disadvantage of this approach would be an explosion in the number of variables. To reduce this number, we perform expression propagation and dead assignment elimination [1]. Afterwards, the instruction sequence in our example collapses into a single assignment:

```
this.data[i] = this.data[i] + 1
```

The statement resembles the original source code from figure code example I quite closely; the only difference is that, as mentioned above, the access to a local variable has been replaced by an access to a member of a generated “variable container” class.

*C. OpenCL Code Generator*

The high-level representation generated by our decompiler closely resembles the abstract syntax of the high-level .NET language code. For this reason, some of the features do not have a direct correspondence in OpenCL C and must be reformulated accordingly.

*1) The Iteration Number*

Parallelizable code fragments have a parameter that specifies which iteration of the fragment is currently running. This approach can be directly mapped to OpenCL, as it provides the concept of *work groups*: functions running on the compute device can query their relative position within the entire compute task via the built-in function `get_global_id`.

Therefore, the first step in translating our input program for this environment is to remove this parameter of the delegate and use the value returned by `get_global_id` instead. To do so, we create a new local variable, and then replace each occurrence of the now-obsolete parameter by it.

*2) Basic Instructions*

Arithmetic instructions in CIL bear the same semantics as the corresponding operators in OpenCL C for integer, floats and pointer types [14][24], so the code generator can use the generated expressions directly. Two sorts of instructions need special attention, though: double-precision floating point arithmetic and control flow instructions.

According to the OpenCL specification, support for the `double` datatype is optional and, if present, needs to be enabled using a special `#pragma` command [24]. There are multiple variants of these extensions, depending on which features can be enabled for the `double` type. The `cl_khr_fp64` extension, which is supported by NVidia, allows using all features mentioned in the specification. On the other hand, the extension `cl_amd_fp64` that AMD uses instead, only allows a limited subset that depends on the version of the ATI Stream SDK in use [15].

To support both cases, our implementation checks which extension the target device supports and creates pragmas accordingly. If the input code fragment requires a feature that is not provided by any of the available extensions for the device, code generation fails.

*Control flow instructions* present a greater challenge, because the OpenCL specification requires devices to support *reducible* control flows only (a graph is reducible if and only if the graph with backward edges removed is acyclic and all nodes can be reached from the start node [27]). However, it also seems to be implementation-defined what OpenCL compilers consider "reducible", so code generation may need to apply additional formatting rules to the input code. For example, when we targeted AMD graphic cards with the AMD OpenCL compiler, it rejected code containing `goto` statements, even though the resulting flow graph was reducible.

Our implementation does not currently check for reducible flow graphs, nor does it attempt to convert input programs into reducible form. For this reason, our generated code may currently be rejected by OpenCL compilers enforcing this restriction. To support execution on arbitrary devices, we would therefore need to convert the input control flow graph into reducible form and output high-level control flow statements (like `if` and `while`) instead of the currently used `goto`.

*3) Object Support*

CIL code is object-oriented and inherently supports creating and manipulating objects via special opcodes. However, we believe only very few features need to be supported when targeting accelerators. By not mapping all possible operations to OpenCL, we can avoid the overhead of converting complete object hierarchies, as well as issues that would arise if code was distributed over multiple devices. Therefore, we chose to support only a subset of object accesses: generated code can read static fields (of arbitrary classes) as well as instance fields reachable via the `this` reference. Similarly, we support invocation of methods only if they are either static (from an arbitrary class) or instance methods of the object referred to by the `this` parameter.

To implement this, the high-level representation output by the initial decompiler pass is searched for accesses to object fields. If a field can be read as per the rules mentioned above, the field access is removed from the intermediate representation and a corresponding parameter is inserted instead; all reads of the field are then changed to access that parameter. The mapping between fields and parameters is attached to the generated code, so this information is available at invocation time.

Exception handling in CIL is also based on objects, but as OpenCL does not have a comparable mechanism, we chose to not support it either. Input programs containing explicit exception-handling code are rejected and not compiled to our intermediate representation. In addition, we do not support exceptions that may be thrown as a side effect of CIL instruction; such instructions (like memory accesses via a pointer) are translated as-is, and may behave erroneously if the parameters are invalid. If a condition occurs that would result in the corresponding CIL instruction to throw an exception, it is silently ignored.

*4) Access to arrays*

Both CIL and OpenCL support single- and multi-dimensional arrays, but there are two remarkable differences:

- During execution of CIL, the runtime performs an implicit bounds check on each access. In case the specified array element does not exist, this results in an *IndexOutOfRangeException* being thrown. On the other hand, C does not perform any implicit checks, so the program behavior is undefined in this case.
- For CIL, the size of each array dimension is a property of the array *object*, so code is independent of the effective size of the data. On the other hand, in the C programming language, array dimension sizes are part of the type specification (e.g., in the function's argument list).

As we do not map exceptions to OpenCL, the first issue is silently ignored. If generated code uses an invalid array index, it accesses an invalid memory location and further operation of the program is undefined; typically, this would result in the computation continuing with erroneous values.

The second issue is solved by converting all arrays into single-dimensional ones for generated code. If a multi-dimensional array is encountered in CIL, our generated C code performs the necessary index arithmetic explicitly. Therefore, we do not need to generate different C code for invocations of the same fragment with different dimension sizes.

Array mapping is performed as follows: suppose $T[l_{n-1}\ldots u_{n-1}, \ldots, l_0\ldots u_0]$ is a CIL array with element type $T$ and $n$ dimensions with lower bounds $l_i$ and upper bounds $u_i$, respectively. To map it to a single-dimensional array, we lay out elements sequentially, starting from the right-most dimension. $T[l_{n-1}, \ldots, l_0]$ is the first element of the mapped array, $T[l_{n-1}, \ldots, l_{0+1}]$ is the second one and so on. More formally, the single-dimensional array $M[0\ldots\Pi_I\ (u_i - l_i + 1)]$ can be mapped as follows:

$$T(m_{n-1}, \ldots, m_0) = M\left(\sum_{0 \le i \le n} (m_i - l_i) \prod_{1 \le k \le i} s_k\right)$$

$s_k$ are scaling factors for the respective array indexes. To prevent overlap, they must satisfy $s_k \ge (u_{k-1} - l_{k-1} + 1)$. If equality holds, elements are laid out without gaps in memory. However, it may be beneficial for performance to use greater values, for example to align elements to cache lines.

Our implementation currently supports arrays with arbitrary numbers of dimensions, although all lower bounds are limited

to zero. However, we believe this is no severe limitation for practical use, as these are the only types of arrays that can be created using built-in C# language statements [26]. To create arrays of other shapes, appropriate .NET framework methods need to be called explicitly.

As we are using explicit index arithmetic, we also need to pass the scaling indexes $s_k$ to the generated code. Similar to passing object fields, this is performed via additional parameters.

*5) Function calls*

Currently we provide only limited support for function calls from generated code. If any of the call instructions for dynamic targets (`callvirt` for virtual methods, `calli` for pointers to functions) is encountered in the input method, the decompiler rejects the CIL code. Calls to statically-known targets (i.e., those using the `call` instruction) are supported, but only for a specific set of functions:

- Static functions
- Instance functions of the object referenced by the `this` parameter
- Array accesses
- Basic mathematical functions
- OpenCL support functions

Invocation of other CIL functions is supported only if those other functions are either non-virtual and defined on the object referenced by the implicit `this` parameter or if they have been declared `static` (without restriction on the object type defining the function). If a call satisfying these requirements is encountered, the OpenCL code generator recursively invokes the decompiler for the target function. If the function itself satisfies the requirements for mapping to OpenCL, C code will be generated for it and included as a subroutine when outputting code for its caller.

In CIL, only accesses to one-dimensional arrays with a lower bound of zero are supported by the language itself; i.e., via special opcodes. All other arrays are represented as objects of special runtime-provided classes, and all accesses (like getting or setting an element, or querying the size of an array dimension) are performed by calls to methods of these special objects. Our implementation recognizes these calls and represents the operation as ordinary array accesses in the high-level intermediate representation. During code generation, they are transformed into accesses to single-dimensional arrays as discussed above.

If input code uses functions from the runtime-provided class `System.Math`, the call is mapped to appropriate OpenCL built-in functions if possible. For example, `System.Math.Max` is translated to the `max` function. If a function is encountered for which no mapping exists, code generation fails.

The last special case of supported functions concerns those for direct OpenCL support like the built-in function `get_global_id`. To allow calling these functions explicitly, the regular .NET compiler needs a function in a normal class it can bind the call to. We therefore added special static ``dummy'' methods to our support code. These methods do not have a valid implementation on the CPU (they just throw an exception if they are invoked). However, our decompiler recognizes corresponding call instructions and redirects them to their OpenCL counterparts instead.

*D. Invocation*

After all transformations of the input program have been performed, it is ready to be executed on an OpenCL compute device. Actual decompilation and code generation only needs to be performed once; the code is cached and can be reused as needed.

Because different devices may support different features, the code generated for the same CIL method also depends on the target device. Therefore, we cache compiled code using both the device and method objects as cache key.

To execute a code fragment with OpenCL, we follow the steps described in the specification: we identify the available compute devices first, and then create necessary control objects for them. Generated code is compiled and stored by means of a *Program* object. A *Kernel* object is created to hold the parameters that will be passed at invocation time. Commands to the target device (e.g., data transfers and execution) are managed by means of a *Queue*.

All object fields accessed in generated code are read via .NET reflection. Scalar values can be stored directly in the *Kernel* object; if an array is encountered instead, a memory buffer is allocated on the device and corresponding data transfers are inserted into the queue. For multi-dimensional arrays, the index scaling factors are calculated and applied to the appropriate parameters.

## V. Evaluation

We tested our implementation on a computer system with an Intel 2600K Quad-Core-Processor and an Nvidia Geforce GTX 460 GPU accelerator.

The CPU was used for serial execution that served as a baseline for the experimental evaluation. For the parallel code blocks, we used the *Parallel.For* implementation of the .NET Framework 4.0. The GPU was used as the target compute device to test and compare the performance characteristics of our implementation against the baseline and the parallel version.

Since there is no widely accepted benchmark for parallel programs based on the .NET Framework, we selected a number of well-known compute intense algorithms from the parallel computing domain. Besides simple functions such as dot product and vector average, we also covered more complex algorithms such as heat transfer simulation and a Sudoku Validator. Moreover, some of the algorithms, e.g. matrix multiplication and histogram computation, are fundamental for many algorithms in the scientific and financial computing domain.

The examples were programmed using the *Parallel.For* implementation and were not optimized for execution on GPUs. In order to run them on the GPU, we simply exchanged the namespaces in the original code without further adjustments.

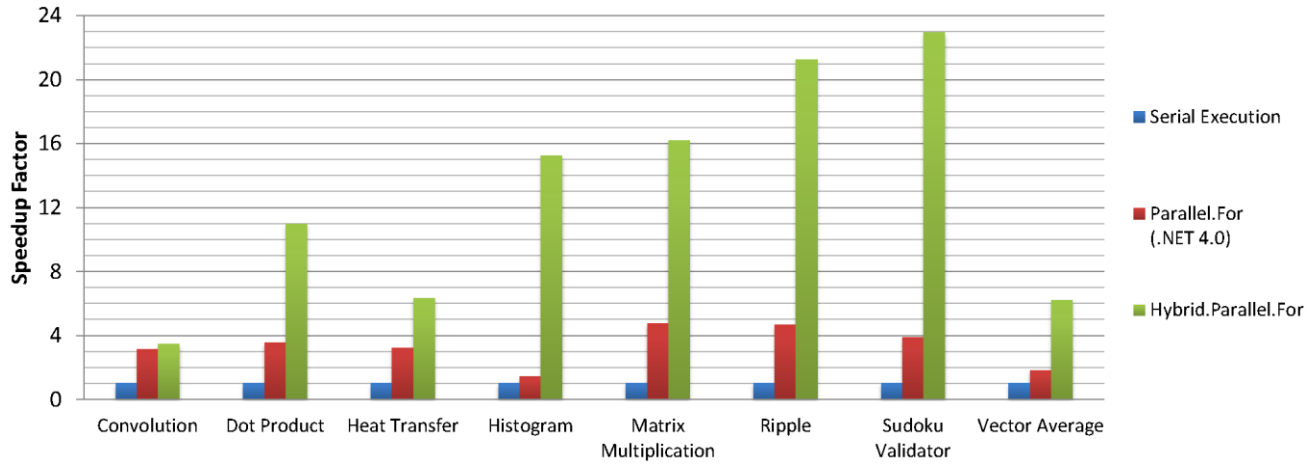


Fig. 2: Evaluation of our approach with various well-known compute intense algorithms.
(CPU: Intel 2600K, GPU: Nvidia GTX460)

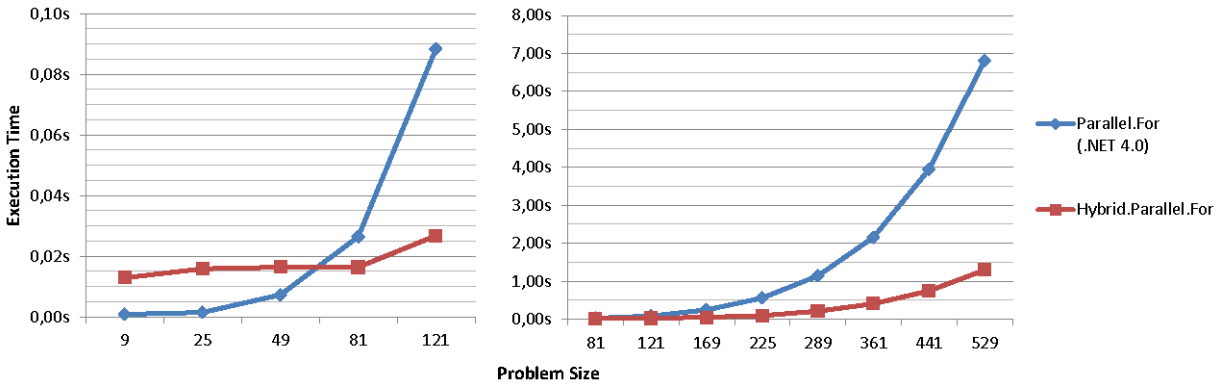


Fig. 3: Execution time to validate Sudoku for various problem sizes.

Our test procedure consisted of three rounds for warm up and 10 rounds for measurements. We chose problem sizes that led to an execution time between two and ten seconds for 10 runs in serial execution. The same problem sizes were tested with Microsoft's *Parallel.For* on CPU and with our *Hybrid.Parallel.For* on GPU. Afterwards the correctness of the results was checked against a sequential implementation.

Memory layout, access patterns and several other execution characteristics have a crucial impact on the performance of a program running on a GPU compute device [7]. Despite the fact that we did not implement any optimizations regarding these characteristics in our library yet, it performed very well; so well in fact, that it can be used to complement Microsoft's *Parallel.For* implementation in its current state. It is expected to perform even better when future enhancements - especially related to memory layout and access patterns - are incorporated in our library.

Fig. 2 shows the results of our initial evaluation. Serial execution time is considered the baseline and therefore normalized to 1.0. The diagram shows that there is a fair scale-up for the CPU runs with most algorithms. *Hybrid.Parallel.For* performed very well compared to Microsoft's *Parallel.For* implementation.

Some algorithms do not scale well on the CPU, but those that do, also achieve excellent performance on the GPU. Exceptions are: *Convolution*, which scales pretty good at the CPU but not on the GPU; and *Histogram*, which does not scale on the CPU but runs very well on the GPU. The data access pattern of *Convolution* is similar to the access pattern of *Heat Transfer* but does not fit as well to the memory fetching mechanisms of GPUs in two out of the three accesses. To improve the performance, the data could be expressed as OpenCL images or stored in the local memory in the GPU memory hierarchy. *Histogram* heavily relies on atomic add operations which seem to be the bottleneck for CPU execution. The GPU implementation in combination with the GPU execution model seems to be better suited for such a workload.

Fig. 3 demonstrates a common case with GPU computing. Due to the initial overhead of setting up the GPU and moving data onto it, GPU computing is generally better suited for larger problem sizes. We expect that by our improvements the performance can also be enhanced so that even smaller examples will benefit from GPUs. In the future versions of our library the break-even will be dynamically supervised.

Furthermore GPUs are best suited for data parallel problems - which many of the examples are. Parallel sorting using a task spawning *Quicksort* would not be appropriate for GPUs. On the other hand there are very fast implementations of *Radixsort* for GPUs [4]. Such basic functionality, as sorting, prefix scans, … can be integrated in and shipped with our library in the future.

Fig. 4 shows the overhead of a Sudoku implementation based on our *Hybrid.Parallel.For* compared to a native version based on OpenCL and C++. Our Sudoku implementation consists of several diverse parallel loops. Hybrid.Parallel has a lower performance as compared to the native implementation.

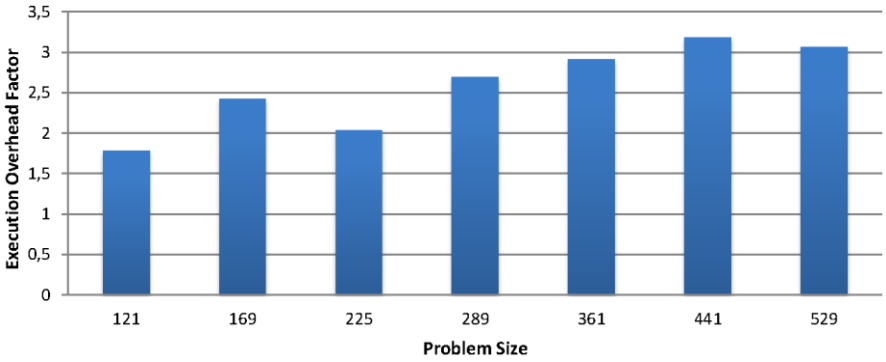


Fig. 4: Overhead of our solution compared to native execution.

On the other hand, a native implementation requires a lot more effort on the part of the programmer, as well as a deep understanding of the OpenCL programming model and characteristics of GPU Compute Devices. Furthermore, optimizations on the native version will only benefit that specific implementation, while optimizing our library will enhance all applications relying on it.

While our approach has the qualitative potential to help developers exploit the performance of widespread accelerators, we expect that extending and enhancing our implementation to support automatic and semi-automatic optimizations will bring further speedup advantage.

## VI. Related Work

**Decompilers and Binary Translators** have been researched for decades. According to [19], the first decompilers date back to the 1960s and were used to help in porting existing binary code to new processor architectures. Decompilers strive not only for *correct*, but also for *human-readable* source code output. Therefore, they do not only focus on individual instructions, but also on optimizations, high-level control flow statements and data structures. One example is the commercial *HexRays Decompiler*, a plug-in for the *Interactive Disassembler IDA* [5] that specifically aims to help in understanding binary code for which no source code is available.

Binary translators share concepts with decompilers, but for a different purpose: translation of binary code for one architecture into semantically-equivalent binary code for another. An example is the *UQBT* framework [18] which accepts x86, SPARC, PA-RISC and 68000 binaries as input and supports C and Java as output.

**Aparapi** [23] is a library that allows to write Java code for AMD GPUs. Application code needs to inherit from the provided *Kernel* class and overwrite its *Run* method. If the kernel is invoked, it will run on the GPU if both a supported graphics card is installed and all features used by the program can be mapped to the GPU; otherwise, it executes on the CPU instead.

Aparapi supports only primitive Java types (with the exception of `double` and `char`). Arrays of primitive types are supported as long as they have only one dimension. They also cannot be used as arguments to calls or aliased within the same method. Static fields of objects cannot be used unless their value is known at compile time; instance fields can be read, not written. Creating new objects (including arrays) is not supported, neither are exception handling nor control flow statements like `break`, `continue` and `switch`.

**GPU.NET** [11] consists of a runtime library and post-processor that works on .NET assemblies. To execute code on a GPU, it needs to be within a static method and annotated with the *Kernel* attribute; such methods are extracted from compiled .NET binaries and processed during an additional compilation step. For buffers like arrays, special annotation can be used to specify the memory space they should be placed in by the runtime.

GPU.NET generates neither CUDA nor OpenCL code, but compiles directly to device-specific binary code. If no supported graphics card is installed, the code is executed on the CPU instead.

Both libraries follow a fundamentally different approach: they do not create OpenCL code, but GPU vendor specific binaries. Because they are tailored for specific vendor hardware, they may exhibit better performance on a subset of accelerators, but are also restricted to specific devices. They also do not hide the OpenCL API, so programmers still need to understand the notion of a kernel, to adjust grid sizes, and calculate thread and block indexes. As described, our solution does this automatically based on best practices, assessment, and runtime evaluation. This leads to code bloat and makes them less suitable for refactoring.

## VII. Future Work – The Vision

In order to get even better performance, we want to implement automatic and semi-automatic optimizations in our library. A full overview on possible optimizations is given in [17].

Fig.  depicts one idea for semi-automatic optimization. Using annotations (called "Attributes" in .NET), a programmer describes the data access patterns of her algorithm. Based on this information, the library can map the data onto the most suitable memory in the hybrid memory architecture. This approach is similar to the description of memory distribution and processor usage in *High Performance Fortran* [13]. Details about the access patterns and mappings we plan to consider are discussed in [16]. Since memory layout and access patterns are the most crucial performance indicators on GPUs, we anticipate that this will enable developers to achieve significant performance improvements while still only focusing on their algorithm instead of thinking about the underlying hardware. Some data access information can be derived automatically without hints from the developer.

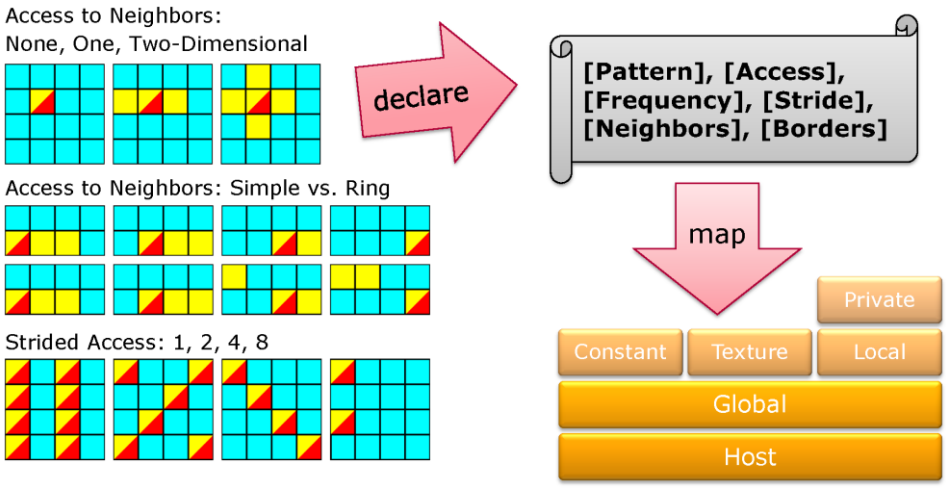


Fig. 5: From Data Access Patterns to Memory Mappings

In addition to semi-automated optimization, we want to support additional parallel programing models such as farmer/worker and domain specific extensions for selected scientific domains. Another possible result from the automated code translation could be the transparent distribution of computational tasks to different machines with a common distributed shared memory [10].

## VIII. CONCLUSION

Hybrid systems are widespread, and meanwhile represent the state-of-the-art in hardware architectures with products such as AMD Fusion or Intel Sandy Bridge. These systems consist of general purpose processors accompanied by accelerators for specific computational tasks on one die.

In this paper, we showed a way to enable developers using high-level languages to seamlessly exploit the power of these systems. From the perspective of a developer, the programming paradigm remains the same, which makes our solution suitable for both code refactoring and create new accelerator-aware applications. We gave a detailed description of the software architecture for our concept, which is based on decompilation and accelerator-neutral code generation. The evaluation with a representative set of parallel algorithms showed acceptable performance. The presented architecture is not bound to a specific device type or vendor, and hides the underlying hardware model and characteristics completely from the developer, which enables light-weight hybrid programming for the masses.

## REFERENCES


[1] A. V. Aho, R. Sethi and J. D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986.

[2] S.-Z. Ueng, M. Lathara, S. S. Baghsorkhi and W.-M. W. Hwu, "CUDA-Lite: Reducing GPU Programming Complexity," in *Proceedings of 21th International Workshop on Languages and Compilers for Parallel Computing, Edmonton, Canada (LCPC), 2008*, Berlin, Heidelberg, 2008.

[3] SourceForge, *OpenCL .Net.*

[4] N. Satish, M. Harris and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," in *2009 IEEE International Symposium on ParallelProcessing*, Washington, DC, USA, 2009.

[5] Hexrays SA, *HexRays Decompiler.*

[6] J. Reinders, Intel threading building blocks, Sebastopol, CA, USA: O'Reilly Associates, Inc., 2007.

[7] Nvidia, "NVIDIA OpenCL Best Practices Guide - Version 2.3," 2009.

[8] Nvidia, "NVIDIA CUDA C Programming Guide 3.2," 2010.

[9] Nvidia, *CUDA Show Case.*

[10] B. Nitzberg and V. Lo, "Distributed shared memory: a survey of issues and algorithms," *Computer*, vol. 24, p. 60, Aug 1991.

[11] TidePowerd Ltd., *GPU.NET.*

[12] T. Lindholm and F. Yellin, *Java Virtual Machine Specification*, Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[13] C. Koelbel, Ed., *High Performance Fortran Specification Version 2.0*, 1997.

[14] ECMA International, *Standard ECMA-335, Common Language Infrastructure (CLI)*, 2002.

[15] B. Gaster, M. Houston, B. Sumner, M. Villmow and B. Zheng, *OpenCL Extension #8: cl_amd_fp64.*

[16] F. Feinbube, *Task and Data Distribution in Hybrid Parallel Systems*, Universitätsverlag Potsdam, 2011.

[17] F. Feinbube, P. Tröger and A. Polze, "Joint Forces: From Multithreaded Programming to GPU Computing," *IEEE Software*, vol. 28, pp. 51-57, Oct 2010.

[18] C. Cifuentes, M. Van Emmerik and N. Ramsey, *UQBT - A Resourceable and Retargetable Binary Translator.*

[19] C. Cifuentes, "Reverse Compilation Techniques," 1994.

[20] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer and K. Skadron, "A performance study of general-purpose applications on graphics processors using CUDA," *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, 2008.

[21] O. A. R. Board, "OpenMP Application Program Interface 3.0," 2008.

[22] D. B. Kirk and W.-m. W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, 1 ed., Morgan Kaufmann, 2010.

[23] Advanced Micro Devices Inc., *Aparapi.*

[24] "The OpenCL Specification - Version 1.1," 2010.

[25] F. Feinbube, B. Rabe, M. von Löwis and A. Polze, "NQueens on CUDA: Optimization Issues," in *Proceedings of 2010 Ninth International Symposium on Parallel and Distributed Computing*, Washington, DC, USA, 2010.

[26] A. Hejlsberg, S. Wiltamuth and P. Golde, *C# Language Specification*, Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[27] J. Janssen and H. Corporaal, "Making Graphs Reducible with Controlled Node Splitting", *ACM Trans. Programming Languages and Systems*, vol. 19, 1997