Technische Universiteit
Eindhoven
University of Technology

A Modular and Parameterisable Classification of Algorithms

C. Nugteren and H. Corporaal

ES Reports
ISSN 1574-9517

ESR-2011-02
12 December 2011

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems

http://www.es.ele.tue.nl/esreports
esreports@es.ele.tue.nl

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems
PO Box 513
NL-5600 MB Eindhoven
The Netherlands

A Modular and Parameterisable Classification of Algorithms

Cedric Nugteren Henk Corporaal
Eindhoven University of Technology, The Netherlands

http://parse.ele.tue.nl
{c.nugteren, h.corporaal}@tue.nl

Initial version December 2011
Updated January 2012

Abstract

Multi-core and many-core were already major trends for the past six years, and are expected to continue for the next decades. With this trend of parallel computing, it becomes increasingly difficult to decide on which architecture to run a certain application or algorithm. Additionally, it brings forth the problem of parallel programming, leading to the so-called software engineering crisis.

In this work we present a new algorithm classification. This algorithm classification is designed for programmers and tools to capture and reason about parallel algorithms. The classification is initially intended to be used to address two challenges: 1) the challenge of parallel programming, and 2), performance prediction for parallel and heterogeneous systems. In order to address these two challenges, we introduce a new algorithm classification in this work. The classification uses a limited vocabulary and a well-defined grammar, creating a modular classification. Additionally, the classification is parameterisable. Both the modularity and the parameterisability of the algorithm classification make it possible to enable a very fine-grained and widely applicable classification.

We furthermore illustrate the new algorithm classification by classifying a number of example algorithms from the field of image processing. We also show a number of code snippets to give an intuitive feel for the classification.

Contents

1 Introduction

The past five decades have shown an exponential growth of single-processor performance. This exponential growth has enabled technology to become pervasive and ubiquitous in our society. A few years ago, in 2004, this exponential growth has ended. The growth of single-processor performance is limited by two major aspects: 1) it has become unfeasible to increase clock frequencies because of power dissipation problems, and 2), micro-architecture improvements have seen an increasingly lower impact [9]. To re-enable performance growth, parallelism is exploited. Enabled by Moore's law, more processors per chip (i.e. multi-core) was already a major trend for the past six years, and is expected to continue for the next decade [6].

While multi-core is expected to enable 100-core processors by 2020 [6], another trend already enables more than 2000 cores. This trend (many-core) uses much simpler and smaller processing cores, creating high throughput parallel processors. An example of such a many-core processor is the Graphics Processing Unit (GPU). Although many-core processors might be suitable for a certain type of application, other applications might prefer multi-core processors. This creates a heterogeneous environment, with both types of processors in one system or even on a single chip (e.g. NVIDIA's Project Denver, AMD's Fusion).

We identify two major challenges that come with these new trends of parallel and heterogeneous computing. The first challenge is programmability: a recent study [12] lists programmability of heterogeneous and parallel processors as a major research challenge. The second challenge is related to performance prediction: how to select a suitable processor for an application prior to code development?

In this work we introduce a new algorithm classification to be able to reason about parallelism in applications. This classification is intended to be used in separate work to address the challenges of programming and performance prediction in a parallel and heterogeneous computing environment. The new classification is therefore focused on the automated use in compilers and tools in particular. The classification uses modularity and parameters to solve the granularity trade-off: it enables a fine-grained classification of algorithms while using a limited vocabulary. This work is focused on the domains of image processing and computer vision, but we believe that the concepts of the classification can be extended to other domains as well.

The remainder of this document is organized as follows. Firstly, we discuss related work on algorithm classifications and motivate the development of a new classification in section 2. Then, in section 3, we introduce the new classification formally. We show a number of example classes and code snippets in section 4. Following, we evaluate the classification in section 5. We briefly give an overview of two applications of the classification in section 6. Finally, we discuss future work and conclude the work in section 7.

2 Related work and motivation

In this section we present related work on algorithm classifications. Algorithm classifications exist in different forms, with different names and for different purposes. We discuss classifications intended to be used with the *algorithmic skeletons* technique [7] separately from other classifications, as they show many similarities with our algorithm classification. While discussing the existing classifications, we motivate the development of a new algorithm classification.

2.1 Classes, patterns and idioms

A number of algorithm classification are introduced as a scheme to capture solutions for recurring design problems in systematic and general ways. These classes are typically not used in automated tools, but are rather used to discuss and reason about design problems. They are in general very coarse-grained and spawn across multiple application domains. Examples of such algorithm classifications are the *dwarfs* and *computational patterns* developed in the ParLab research group [2]. A hierarchical set of *design patterns* [13] was later introduced to reason about design problems at various granularity levels.

Algorithm classifications have also been developed to be used by tools and compilers. An example is the use of *idioms* to classify algorithms. Idioms have been used to automatically classify C-code [16] and to predict performance of complete applications [5]. The idioms as introduced in [5] are very limited in number and are still not as fine-grained as the classification we introduce in this work.

In other work we see that algorithms are classified for the purpose of algorithmic choice. In such cases, it is important to capture the functionality of the algorithms in the classes rather than memory access behaviour as is done in our work. Examples of classifications introduced to enable algorithmic choice are [1] and [11].

2.2 Classifications used with algorithmic skeletons

Many variations of algorithm classifications have been introduced as part of work on the *algorithmic skeletons* technique [7], which is used for code generation purposes. There is a great body of work in this area, most of which introduces their own algorithm classification as part of research on algorithmic skeletons. In a survey of algorithm classification for algorithmic skeletons [4], a total of 10 different classifications are presented. On average 4 classes are included in these classifications, with *divide and conquer*, *pipeline*, and *farm* being the most common among all 10 classifications. In comparison to these classifications, we provide much finer-grained and detailed classes.

Other work on algorithmic skeletons introduces algorithm classifications highly related to this work. Examples are [3], [8], [10] and [15]. In comparison to these works, we provide more detail in our classes (key-enabler: parameters) and cover a large number of applications while maintaining understandability (key-enabler: modularity).

3 Algorithm classification

To address the challenges of parallel programming and performance prediction, we introduce a new algorithm classification. In this section, we first give a small toy example, after which we introduce the vocabulary and grammar of the classification.

3.1 Code examples

Before introducing the classification formally, we classify a number of example code snippets to give an intuitive feel for the classification. The four examples as shown in listing 1 are classified as follows:

- In lines 1-3 a vector of size K is element-wise multiplied, incremented, and stored as another vector. Since every *element* of the input corresponds to an *element* of the output and the vector size is K, we classify this code snippet as 'K|element → K|element'.
- The for-loop in lines 5-7 performs a similar operation, but now also requires two *neighbours* to compute one output *element*. The classification becomes 'K|neighbourhood(3) → K|element', since the neighbourhood is of size 3 (including the element itself).

```
1 for(i=0; i<K; i=i+1) {                    K|element → K|element
2   B[i] = 2 * A[i] + 5;
3 }
4
5 for(i=0; i<K; i=i+1) {                    K|neighbourhood(3) → K|element
6   B[i] = 0.3*A[i-1] + 0.4*A[i] + 0.3*A[i+1];
7 }
8
9 for(a=0; a<10; a=a+1) {                   10x10|element → 10x10|element
10   for(b=0; b<10; b=b+1) {
11     value = A[a][b];
12     if (value > 255) {
13       value = 255;
14     }
15     B[a][b] = value;
16   }
17 }
18
19 for(i=0; i<K; i=i+1) {                    K|element → 1|shared
20   B = B + A[i];
21 }
```

Listing 1: Four example code snippets of different algorithm classes. The examples are classified using the algorithm classification introduced in this work.

- Similar to the code snippet in lines 1-3, the code in lines 9-17 performs an *element* to *element* computation. However, in this case, we process two dimensional matrices of size 10 by 10. The code is therefore classified as '10x10|element → 10x10|element'.
- The final snippet (lines 19-21) processes the input per *element*, but stores the result in a *shared* output. It is therefore classified as 'K|element → 1|shared', with 1 being the size of the output.

3.2 Vocabulary and grammar

The classification consists of both a grammar and a vocabulary. In this section, we formally define the algorithm classification by first introducing the grammar and the vocabulary. Following, we introduce the meaning of the algorithm classification.

Algorithm classes can be described using a vocabulary and a grammar. The classification's grammar is defined as follows:

$$P\ S|D\ \ [\wedge\ S|D]^*\ \ \rightarrow\ \ S|D\ \ [\wedge\ S|D]^*$$

In this definition, the asterisk symbol (*) implies zero or more occurrences of everything contained by brackets preceding the asterisk. Entries for P, S and D need to be replaced by the classification's vocabulary. Combinations of $S|D$ represent input and output data-structures, in which S represents a dimension and D represents a data structure. Furthermore, P represents a prefix applicable to the whole algorithm class. The prefix P, data access patterns D and dimensions S need to be replaced using the algorithm classification's vocabulary, which is defined as follows:

$$P = \begin{pmatrix}\text{unordered}\\ \text{multiple}\end{pmatrix} \qquad S = \begin{pmatrix}\text{A}\\ \text{AxB}\\ \text{...}\\ \text{Ax...xN}\end{pmatrix} \qquad D = \begin{pmatrix}\text{element}\\ \text{tile}\,(S)\\ \text{neighbourhood}\,(S)\\ \text{shared}\\ \text{full}\end{pmatrix}$$

The algorithm classification distinguishes input from output data structures through the arrow (→). Data structures preceding the arrow are considered input data structures, while data structures listed after the arrow are output data structures. Furthermore, when multiple data structures are present as input or output, the wedge operator (∧) separates these data structures. For example, the class 'S|D ∧ S|D → S|D' contains two input data structures and one output data structure. The dimensions of the data structures are given by S, which defines both the number of dimensions and the size of each dimension. Sizes are separated by an 'x'. For example, '10x25x3' defines a 3-dimensional data structure with sizes of respectively 10, 25 and 3.

The data structure D itself is characterized by its data access patterns. We provide a graphical comparison between the access patterns element, tile and neighbourhood in figure 1. The five different patterns which are part of the classification's vocabulary are defined as follows:

element: The element access pattern represents a single access of a data structure's contents at each coordinate in the structure. Accesses to individual elements of the data

structure are assumed to be independent of each other. The amount of parallelism for this pattern is therefore equal to the size of the data structure.

tile(T): When accessing data in a tile pattern, a structure of dimensions T is accessed simultaneously, but independent of other tiles in the same data structure. There is no data re-use, all contents are accessed once. The parallelism present within this pattern is equal to the size of the data structure divided by the tile size.

neighbourhood(N): The neighbourhood access pattern is similar to the element pattern, but enables overlap. The pattern describes the re-use of a data structure's contents for a neighbourhood of dimensions N centered around each coordinate of the structure. This data access pattern is to be used as input pattern only.

shared: The shared data access pattern is an output only pattern. It is used when multiple accesses occur to contents at a single coordinate in a data structure. The pattern also enables reading from the output at the same coordinate.

full: The full access pattern accesses the complete data structure. This implies that no parallelism is available. This pattern is equal to the use of the tile pattern with a tile size equal to the data structure's size.

For each set of accesses to a data structure's contents according to the patterns, an operation needs to be performed to produce output from the input. This operation is given by the function $f()$, which is introduced as part of the classification. For example, if the algorithm class is equal to 'K|element → K|tile(2)', then $f()$ defines the operation required to produce two output elements from one input element.

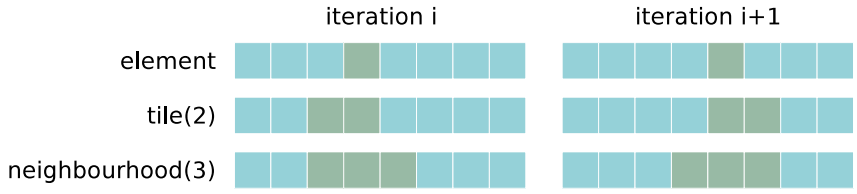

Figure 1: An illustration of the differences of the element, tile and neighbourhood patterns.

In addition, the classification provides the two prefixes P, which can be placed in front of an algorithm class. The two prefixes are defined as follows:

unordered: In general, the relation between the input and output patterns is based on a one to one coordinate mapping. The use of the unordered prefix implies that any coordinate mapping can be used, e.g. a scattered or reverse mapping.

multiple: The shared pattern assumes a single access only to the shared data structure. When more than one output is written by the operator $f()$ for the shared pattern, the multiple prefix can be used.

4 Illustrating the algorithm classification by example

In this section we illustrate the presented algorithm classification in three ways. Firstly, we present a number of example classes and show their formal meaning. Then, we classify a number of example elementary algorithms from the field of image processing. Finally, we classify a number of code snippets according to the presented classification.

4.1 Example classes and their formal notations

In this section we present a number of example classes to illustrate the use of the classification's vocabulary and grammar. We describe the meaning of each example class in natural language, but also give a formal description of the class. To do so, we first define a number of notations.

The following notations are used in this section: 1) X^i, Y^i and Z^i represent i-dimensional data structures, 2) $X^i_{\mathbf{d}}$ denotes the contents of a data structure X^i at coordinate $\mathbf{d}$, and 3), k gives the natural numbers: $k \in \mathbb{N}$. Furthermore, to represent coordinates in example data structures, we define a number of coordinate ranges. The coordinate ranges have varying sizes (e.g. UxV) and represent different parts of a data structure (e.g. a 2D-tile). They are given as follows:

AxB (matrix)	$\mathcal{D} = \{(x, y) \mid 0 \leq x < A \wedge 0 \leq y < B\}$
UxV (2D-tile)	$\mathcal{T} = \{(x, y) \mid 0 \leq x < U \wedge 0 \leq y < V\}$
B (1D-tile)	$\mathcal{T}' = \{(x, y) \mid 0 \leq x < B \wedge y = 0\}$
NxM (2D-neighbourhood)	$\mathcal{N} = \{(x, y) \mid -(\frac{N-1}{2}) \leq x \leq (\frac{N-1}{2}) \wedge -(\frac{M-1}{2}) \leq y \leq (\frac{M-1}{2})\}$
N (1D-neighbourhood)	$\mathcal{N}' = \{(x, y) \mid -(\frac{N-1}{2}) \leq x \leq (\frac{N-1}{2}) \wedge y = 0\}$
C (vector)	$\mathcal{V} = \{(x) \mid 0 \leq x < C\}$

Using the introduced notation, we give a number of example classes in table 1. In this table we give a formal notation for each class. We also give a description of the different classes in natural language:

AxB|element → AxB|element: This example class implies that for each data element in the input data structure of size AxB one data element at a corresponding coordinate in the output data structure is produced by the operator $f()$. These operations and accesses are independent of each other and can thus be done in parallel.

unordered AxB|element → AxB|element: Similar to the previous class, elements are accessed and produced element-wise. However, in this case, the coordinate relation is unknown. Elements might for example be written in inverse or pseudo random order. Still, each input and output element will only be accessed once.

AxB|tile(1xB) → A|element: For this class the input data structure is not accessed element-wise, but tile-wise. The tile size is equal to one column of the input matrix, producing only A output elements. The operator $f()$ accesses B input elements and produces one output element. This results in a reduction of a matrix of size AxB into a vector of length A. There is no data re-use and a maximum parallelism of A.

example class	formal notation
AxB|element → AxB|element	$\forall \mathbf{d} \in \mathcal{D}$: $f\left(X^2_{\mathbf{d}}\right) \rightarrow Y^2_{\mathbf{d}}$
unordered AxB|element → AxB|element	$\forall \mathbf{d} \in \mathcal{D}\ \exists \mathbf{d}' \in \mathcal{D}$: $f\left(X^2_{\mathbf{d}}\right) \rightarrow Y^2_{\mathbf{d}'}$
AxB|tile(1xB) → A|element	$\forall \mathbf{d} \in \mathcal{D} \mid d = k \cdot \mathcal{T}'$: $f\left(\forall t' \in \mathcal{T}'\left(X^2_{\mathbf{d}+t'}\right)\right) \rightarrow Y^2_c \mid c = d_x$
AxB|tile(UxV) → $\frac{A}{U}$x$\frac{B}{V}$|element	$\forall \mathbf{d} \in \mathcal{D} \mid d_x = k \cdot U \wedge d_y = k \cdot V$: $f\left(\forall \mathbf{t} \in \mathcal{T}\left(X^2_{\mathbf{d}+\mathbf{t}}\right)\right) \rightarrow Y^2_{\mathbf{c}} \mid \mathbf{c} = (\frac{d_x}{U}, \frac{d_y}{V})$
AxB|tile(UxV) → AxB|tile(UxV)	$\forall \mathbf{d} \in \mathcal{D} \mid d_x = k \cdot U \wedge d_y = k \cdot V$: $f\left(\forall \mathbf{t} \in \mathcal{T}\left(X^2_{\mathbf{d}+\mathbf{t}}\right)\right) \rightarrow \forall \mathbf{t} \in \mathcal{T}\left(X^2_{\mathbf{d}+\mathbf{t}}\right)$
AxB|element → A·UxB·V|tile(UxV)	$\forall \mathbf{d} \in \mathcal{D}$: $f\left(X^2_{\mathbf{d}}\right) \rightarrow \forall \mathbf{t} \in \mathcal{T}\left(X^2_{\mathbf{c}+\mathbf{t}} \mid \mathbf{c} = (d_x \cdot U, d_y \cdot V)\right)$
AxB|neighbourhood(NxM) → AxB|element	$\forall \mathbf{d} \in \mathcal{D}$: $f\left(\forall \mathbf{n} \in \mathcal{N}\left(X^2_{\mathbf{d}+\mathbf{n}}\right)\right) \rightarrow Y^2_{\mathbf{d}}$
AxB|neighbourhood(N) → AxB|element	$\forall \mathbf{d} \in \mathcal{D}$: $f\left(\forall \mathbf{n}' \in \mathcal{N}'\left(X^2_{\mathbf{d}+\mathbf{n}'}\right)\right) \rightarrow Y^2_{\mathbf{d}}$
AxB|element → 1|shared	$\forall \mathbf{d} \in \mathcal{D}$: $f\left(X^2_{\mathbf{d}}, Y^0\right) \rightarrow Y^0$
AxB|element → C|shared	$\forall \mathbf{d} \in \mathcal{D}\ \exists v \in \mathcal{V}$: $f\left(X^2_{\mathbf{d}}, Y^1_v\right) \rightarrow Y^1_v$
multiple AxB|element → C|shared	$\forall \mathbf{d} \in \mathcal{D}$: $f\left(X^2_{\mathbf{d}}, \forall v \in \mathcal{V}\left(Y^1_v\right)\right) \rightarrow \forall v \in \mathcal{V}\left(Y^1_v\right)$
AxB|element ∧ AxB|element → AxB|element	$\forall \mathbf{d} \in \mathcal{D}$: $f\left(X^2_{\mathbf{d}}, Y^2_{\mathbf{d}}\right) \rightarrow Z^2_{\mathbf{d}}$
AxB|element ∧ AxB|full → AxB|element	$\forall \mathbf{d} \in \mathcal{D}$: $f\left(X^2_{\mathbf{d}}, \forall d' \in \mathcal{D}\left(Y^2_{\mathbf{d}'}\right)\right) \rightarrow Z^2_{\mathbf{d}}$

Table 1: A number of example algorithm classes and their corresponding formal definition.

AxB\|tile(UxV) → $\frac{A}{U}$x$\frac{B}{V}$\|element: Similar to the previously discussed class, the operator $f()$ accesses a tile and produces one element. However, in this case the tile is 2-dimensional and of size UxV. This operation results in a reduction in size of the input matrix by a factor UxV. Individual tiles do not overlap and can be processed independently of each other.

AxB\|tile(UxV) → AxB\|tile(UxV): This class is similar to the previous tile-based classes. However, in this case, the operator $f()$ produces a tile as an output as well. The input and output data structures are of the same size since the tiles of the input and output are of the same size.

AxB\|element → A·UxB·V\|tile(UxV): For this class the operator $f()$ accesses individual elements from the input data structure and produces tiles in the output structure. This results in an increase of the matrix dimensions by a factor UxV.

AxB\|neighbourhood(NxM) → AxB\|element: This neighbourhood-based class is similar to the first discussed class, because the data structures are accessed element-wise.

However, in contrast to the access of a single element, a surrounding neighbourhood NxM of elements is accessed by the operator $f()$. In contrast to the tile-based classes there is overlap and thus data re-use in this class. This also implies that the input and output data structure are of the same size.

AxB|neighbourhood(N) → AxB|element: This class is similar to the previous class, but now accesses a 1-dimensional neighbourhood instead of a 2-dimensional. Since only a single dimension given, the neighbourhood is taken in the first dimension (corresponding to the size A of the input data structure).

AxB|element → 1|shared: For this class the input is accessed element-wise as seen before. However, the output data structure, which consists of a single element in this case, is shared among each operation performed by $f()$. The operator $f()$ might even access the previous value of the output data structure as input, for example to perform accumulation.

AxB|element → C|shared: This class is similar to the previous class. The difference is found in the fact that the shared output is now a vector. The output coordinate accessed by the operator $f()$ is undefined for the class.

multiple AxB|element → C|shared: With the addition of the multiple prefix, this class enables the production of multiple output elements by the operator $f()$. As for the previous class, the output data structure is shared and the coordinates accessed are undefined.

AxB|element ∧ AxB|element → AxB|element: Very similar to the first class, this class performs an element-wise computation. However, now, two elements from two different data structures are accessed as inputs to the operator $f()$.

AxB|element ∧ AxB|full → AxB|element: This class also takes two input data structures and produces one output data structure. However, the second data structure can now be accessed entirely each time the operator $f()$ accesses one element from the first input and produces a single output element. In this case the size of the second input is equal to the sizes of the other data structures.

4.2 Classification of example algorithms

Related to the example classes in table 1, we list a number of elementary algorithms from the domains of image processing and computer vision in table 2. Many of these elementary algorithms are building blocks for larger algorithms or applications. We take four algorithms from table 2 and explain their classification:

- We classify the algorithm CONTRAST ENHANCEMENT as 'AxB|element → AxB|element', in which A and B represent the x and y-dimensions of the input and output image. The algorithm iterates over all the elements of the input image, each time producing one output element. The structure of the algorithm therefore resembles the example given in lines 9-17 of listing 1, but performs a different computation (lines 12-14).

- For the algorithm ROTATE 90 DEGREES, we rotate an image clockwise. We classify this algorithm as 'unordered AxB|element → AxB|element', since the coordinate relation between the input and output is not one-to-one, but involves a rotation. Still, as for the previous example algorithm, an element-wise operation is performed.
- The algorithm PIXELIZATION is classified as 'AxB|tile(UxV) → AxB|tile(UxV)'. First, an operation is performed per *tile* on the input image, resulting in a fewer number of intermediate values. These intermediate values are then written multiple times into a *tile* of the output image, resulting in an output image of the same dimensions as the input.
- A large number of 2D convolution filters can be separated into a sequence of two 1D filters. We classify each part of a SEPARABLE 2D FILTER as a 1-dimensional neighbourhood operation on a 2-dimensional image: 'AxB|neighbourhood(N) → AxB|element'.

class	example algorithms
AxB|element → AxB|element	image copy binarization colour transformation gamma correction contrast enhancement arithmetic & logic (monadic)
unordered AxB|element → AxB|element	rotate 90 degrees xy-mirroring rotate (forward warping, no interpolation) shear-x (forward warping, no interpolation)
AxB|tile(1xB) → A|element	column-projection
AxB|tile(UxV) → $\frac{A}{U}$x$\frac{B}{V}$|element	scale down
AxB|tile(UxV) → AxB|tile(UxV)	2D type II discrete cosine transform pixelization adaptive binarization (alternative 1)
AxB|element → A·UxB·V|tile(UxV)	enlarge
AxB|neighbourhood(NxM) → AxB|element	convolution (static filter) adaptive binarization (alternative 2) erosion/dilation
AxB|neighbourhood(N) → AxB|element	separable 2D filter (in one dimension) 1D convolution
AxB|element → 1|shared	sum min/max
AxB|element → C|shared	histogram
AxB|element ∧ AxB|element → AxB|element	differencing arithmetic & logic (dyadic)

Table 2: The classification of example algorithms from the domain of image processing and computer vision. The classes are taken from the example classes which are formally defined in table 1.

4.3 Classification of code snippets

We furthermore classify a number of code snippets. The classes along with the code snippets are shown in listings 2-5. The examples are grouped as: element-based (listing 2), shared-based (listing 3), neighbourhood-based (listing 4), and tile-based (listing 5).

```
100x16|element → 100x16|element
for(i=0;i<100;i=i+1)
  for(j=0;j<16;j=j+1)
    B[i][j] = 2*A[i][j];

unordered 4x8|element → 4x8|element
for(i=0;i<4;i=i+1)
  for(j=0;j<8;j=j+1)
    B[i][j] = A[i][7-j];

64|element → 64|element
for(i=0;i<64;i=i+1)
  temp = 3;
  for (t=0;t<8;t=t+1)
    temp = temp * A[i];
  B[i] = cos(temp);

8|element + 8|element → 8|element
for(i=0;i<8;i=i+1)
  C[i] = 2*A[i] + B[i];
```

Listing 2: Element-based classes.

```
16x48|element → 1|shared
for(i=0;i<16;i=i+1)
  for (j=0;j<48;j=j+1)
    B = B + 2*A[i][j];

16x48|element → 5x5|shared
for(i=0;i<16;i=i+1)
  for (j=0;j<48;j=j+1)
    in = A[i][j];
    B[i%5][j%5] = B[i%5][j%5] + in;

16x48|element → 12|shared
for(i=0;i<16;i=i+1)
  for (j=0;j<48;j=j+1)
    value = A[i][j];
    B[value] = B[value] + i;

multiple 50|element → 100|shared
for(i=0;i<50;i=i+1)
  value = A[i];
  B[value] = B[value] + i;
  B[value*3] = B[value*3] + 1;
```

Listing 3: Shared-based classes.

```
32x32|neighb.(3x3) → 32x32|element
for(i=0;i<32;i=i+1)
  for(j=0;j<32;j=j+1)
    avg = 0;
    for(a=-1;a<=1;a=a+1)
      for (b=-1;b<=1;b=b+1)
        avg = avg + A[i+a][j+b];
    B[i][j] = avg / 9;

300|neighb.(3) → 300|element
for(i=0;i<300;i=i+1)
  B[i] = 2*A[i-1]+4*A[i]+2*A[i+1];
```

Listing 4: Neighbourhood-based classes.

```
50x10|tile(1x10) → 50|element
for(i=0;i<50;i=i+1)
  result = 0;
  for (j=0;j<10;j=j+1)
    result = result + j * A[i][j];
  B[i] = result;

100x100|tile(2x2) → 50x50|element
for(i=0;i<50;i=i+1)
  for (j=0;j<50;j=j+1)
    out = 0;
    for(a=0;a<2;a=a+1)
      for (b=0;b<2;b=b+1)
        out = out * A[2*i+a][2*j+b];
    B[i][j] = out;

50|tile(5) → 50|tile(5)
for(i=0;i<10;i=i+1)
  result = 0;
  for(a=0;a<5;a=a+1)
    if (A[5*i+a] > 12)
      result = result + 3;
  for(a=0;a<5;a=a+1)
    B[5*i+a] = result;

32|element → 128|tile(4)
for(i=0;i<32;i=i+1)
  value = A[i];
  for(a=0;a<4;a=a+1)
    B[4*i+a] = value;
```

Listing 5: Tile-based classes.

5 Evaluation of the classification

In this work we presented a new algorithm classification. In this section, we briefly compare the classification to existing work and discuss a number of difficult and ambiguous classification problems. Prior to that, we evaluate the presented algorithm classification on three important aspects: 1) completeness, 2) abstraction level or granularity, and 3), understandability:

completeness: The presented classification is not proven to be complete. Nevertheless, we have shown that a significant amount of example algorithms and code snippets can be classified under the presented classification. Larger algorithms or complete applications can be split up to enable the classification of separate smaller algorithms. Furthermore, the classification's modularity enables the support for any number and type of input and output data structures.

abstraction level: The classification uses parameters to achieve a very fine-grained classification which includes a high level of detail. Apart from parameters added to the data structures itself, we also add parameters for neighbourhood-sizes and tile-sizes.

understandability: In order to create an understandable and fine-grained classification at the same time, we introduced a classification grammar with a limited vocabulary. The data structure patterns used in the vocabulary are very limited, decreasing the threshold to familiarize with the classification. Additionally, the vocabulary is closely related to jargon used in the domains of image processing and computer vision, which suggests ease of adaptability.

5.1 Comparison with existing work

If we compare our algorithm classification with existing classification as discussed in section 2, we find a number of differences. First of all, we find that our classification is much more fine grained, especially if we compare against *patterns* as presented in [2] and [13]. But even if we compare our classification to the more finer-grained classifications ([3], [4], [8], [10], [15], and [16]), we find that we provide much more detail. This level of detail is achieved through the introduction of parameters, which provides a manner to distinguish different sizes and dimensions from each other - if needed. For example, we can distinguish a large neighbourhood from a small neighbourhood, and we can identify between the use of 1D and 2D tiles.

Apart from the finer granularity, we also improve upon existing classifications by introducing a modular classification. With modularity, we do not have to define individual classes, but can construct them using a given grammar and a limited vocabulary. This contributes towards the completeness of the classification, making it possible to support any number and combination of input and output data structures. Furthermore, our classification is based on a well-defined grammar and vocabulary, while many existing classifications are introduced less formally.

5.2 Classification difficulties

In this section we discuss a number of difficult and ambiguous classification problems. These discussions can be seen as responses to 'how should I classify this algorithm'-problems.

classification ambiguities: Algorithms can sometimes be classified in multiple ways. An example of this is MATRIX MULTIPLICATION. We can express the algorithm as: one row of the first matrix *and* all elements of the second matrix *produces* one row of the output matrix. This is expressed as: 'AxB|tile(1xB) $\wedge$ BxC|full $\rightarrow$ AxC|tile(1xC)'. The resulting class is conservative, as it does not capture all parallelism. We can solve this by expressing the algorithm at a lower level: the classified code is now executed in a loop over the number of rows (A) of the input and output matrices. The classification of the inner part becomes: one row *and* one column *produces* one output element. This is expressed as: 'B|full $\wedge$ BxC|tile(Bx1) $\rightarrow$ C|element'.

class combinations: In some cases multiple algorithm classes might be combined to create a single class. An 8x8 PIXELIZATION on a 32x32 image for example can be classified as a sequence of 32x32|tile(8x8) $\rightarrow$ 4x4|element and 4x4|element $\rightarrow$ 32x32|tile(8x8) or as the single class 32x32|tile(8x8) $\rightarrow$ 32x32|tile(8x8). Which one of these solutions to select depends on the implementation details and whether or not the intermediate result is required.

classes as upper-bounds: Although classes pose restrictions on data accesses, it does not mean that all accesses have to be performed for an algorithm to be classified under a certain class. In this sense, classes can be seen as upper-bounds. For example, an algorithm might still be classified as 8|element $\rightarrow$ 8|element, even if it does not produce a result for the 5th element in the output data structure. Another example is a neighbourhood-based class which uses only a subset of the neighbourhood for each operation $f()$.

less restricting classes: Irregular algorithms or algorithms with limited or no parallelism can be classified under less restricting classes, such as 'S|full $\rightarrow$ S|full'. In this way, a classification can be found, but the parallelism expressed by the class might be lower than the actual parallelism found in the algorithm.

6 Applications of the algorithm classification

In this section, we briefly discuss two applications of the presented algorithm classification. We discuss the applications of the classification to address both the challenges of programming and performance prediction for multi-core and many-core processors. An overview of the two approaches taken is shown in figure 2.

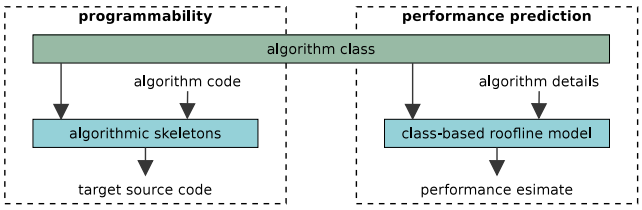

Figure 2: Using algorithm classes to address programmability challenges and to enable performance prediction.

6.1 Parallel programming using the classification

A solution for the programmability challenge might be found in the use of algorithmic skeletons [7]. It has been applied to multi-core and many-core processors in other work, but, although they show promising results, their tools have not been widely adopted by programmers yet. In contrast to existing work, we focus our research on the algorithm classification itself. The result is the development of the 'Bones' source-to-source compiler. This compiler is based on algorithmic skeletons and uses the presented algorithm classification. The work is ongoing and currently generates high-performance CUDA and OpenCL code.

6.2 Performance prediction using the classification

We aim to develop a method to be able to choose a suitable processor for a given set of applications without requiring to port any source code to the target processor candidates. Thus, we need a method to enable performance prediction based on either the original source code or on a textual description of the application. To achieve this, we have developed a specific performance model for a specific algorithm class. Each of these models can be seen as a refinement of the roofline model [18]. The new model, named the 'boat hull model' [14], is based on the roofline model, but gives much tighter bounds on performance as it captures class information. It is currently applicable to GPUs and CPUs and gives very valuable predictions, even though no source code is required.

7 Conclusions and future work

In this work we introduced a new modular and parameterisable algorithm classification. In this section, we briefly discuss future work and conclude the current work.

7.1 Future work

As part of our work on the algorithm classification we plan to continue in two directions:

1. We aim to further fine-tune the presented algorithm classification. To do so, we plan to validate the classification against the PolyBench benchmark suite [17]. Additionally, we plan to validate the classification against a set of real-life applications.

2. We aim to automatically extract class information from source code. We plan to develop a tool to analyze C-code and to annotate the source with identified classes. The goal is similar to the work performed in [16] for a different classification.

7.2 Conclusions

In this work, we introduced a new algorithm classification which is designed for programmers and tools to capture and reason about parallel algorithms. We have shown a number of example classes, algorithms and code snippets which illustrate the presented algorithm classification. We have furthermore evaluated the classification on completeness, granularity and understandability.

The presented classification is initially designed to be used to address two different challenges: 1) the challenge of parallel programming, and 2), performance prediction of parallel and heterogeneous systems. The use of the classification for these and other purposes is left for future work.

The algorithm classification we presented in this work uses a limited vocabulary and a well-defined grammar, creating a modular classification. Additionally, the classification is parameterisable. Both the modularity and the parameterisability of the algorithm classification make it possible to achieve a very fine-grained but widely applicable classification while maintaining understandability. Compared to existing work, our classification provides much more detail, but also improves on completeness and understandability. It is furthermore formally defined and illustrated through numerous examples.

References

[1] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: A Language and Compiler for Algorithmic Choice. In *PLDI '09: ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2009.

[2] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A View of the Parallel Computing Landscape. *Communications of the ACM*, 52:56–67, October 2009.

[3] W. Caarls, P. Jonker, and H. Corporaal. Algorithmic Skeletons for Stream Programming in Embedded Heterogeneous Parallel Image Processing Applications. In *IPDPS '06: 20th International Parallel and Distributed Processing Symposium*. IEEE, 2006.

[4] D. K. G. Campbell. Towards the Classification of Algorithmic Skeletons. Technical Report YCS 276, University of York, 1996.

[5] L. Carrington, M. M. Tikir, C. Olschanowsky, M. Laurenzano, J. Peraza, A. Snavely, and S. Poole. An Idiom-finding Tool for Increasing Productivity of Accelerators. In *ICS '11: International Conference on Supercomputing*. ACM, 2011.

[6] B. Catanzaro, A. Fox, K. Keutzer, D. Patterson, B.-Y. Su, M. Snir, K. Olukotun, P. Hanrahan, and H. Chafi. Ubiquitous Parallel Computing from Berkeley, Illinois, and Stanford. *IEEE Micro*, 30:41–55, March 2010.

[7] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1991.

[8] J. Enmyren and C. W. Kessler. SkePU: a Multi-backend Skeleton Programming Library for Multi-GPU Systems. In *HLPP '10: Fourth International Workshop on High-level Parallel Programming and Applications*. ACM, 2010.

[9] S. H. Fuller and L. I. Millett. Computing Performance: Game Over or Next Level? *IEEE Computer*, 44:31–38, January 2011.

[10] D. Goswami, A. Singh, and B. R. Preiss. From Design Patterns to Parallel Architectural Skeletons. *Elsevier Journal of Parallel and Distributed Computation*, 62:669–695, April 2002.

[11] R. Jongerius, P. Stanley-Marbell, and H. Corporaal. Quantifying the Common Computational Problems in Contemporary Applications. In *IISWC '11: IEEE International Symposium on Workload Characterization*. IEEE, 2011.

[12] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the Future of Parallel Computing. *IEEE Micro*, 31:7–17, September 2011.

[13] K. Keutzer and T. Mattson. A Design Pattern Language for Engineering (Parallel) Software. In *Intel Technology Journal*, 2010.

[14] C. Nugteren and H. Corporaal. The Boat Hull Model: Adapting the Roofline Model to Enable Performance Prediction for Parallel Computing. In *PPoPP '12: 17th Symposium on Principles and Practice of Parallel Programming*. ACM, 2012.

[15] C. Nugteren, H. Corporaal, and B. Mesman. Skeleton-based Automatic Parallelization of Image Processing Algorithms for GPUs. In *SAMOS XI: International Conference on Embedded Computer Systems*. IEEE, 2011.

[16] C. Olschanowsky, A. Snavely, M. Meswani, and L. Carrington. PIR: PMaC's Idiom Recognizer. In *ICPPW '10: 39th International Conference on Parallel Processing Workshops*. IEEE, 2010.

[17] L.-N. Pouchet. PolyBench: The Polyhedral Benchmark Suite. http://www.cse.ohio-state.edu/~pouchet/software/polybench/.

[18] S. Williams, A. Waterman, and D. Patterson. Roofline: an Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52:65–76, April 2009.