A Machine Learning Approach to Mapping Streaming Workloads to Dynamic Multicore Processors

Published at LCTES 2016

Authors: Paul-Jules Micolet (University of Edinburgh, UK)

Presenter: Wangjiaxuan Xin

March 9, 2023

UNIVERSITY OF NORTH CAROLINA CHARLOTTE

Introduction

In recent years a shift has occurred towards heterogeneity(e.g. ARM big.LITTLE) and reconfigurability.

Dynamic Multicore Processors (DMPs) bridge the gap between fully reconfigurable processors and homogeneous multicore systems

Dynamic multicore processors allow cores to compose (or fuse) together into larger logical cores to accelerate each single thread.

Some Features For DMPs

In this paper a dynamic multicore processor allows cores to compose their execution resources, register files and private L1 caches to create logical processors to accelerate a single thread.

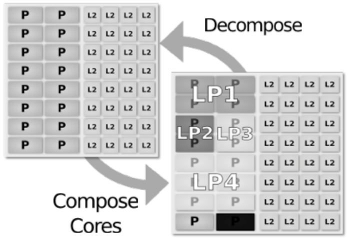


Figure 1: High-level view of a dynamic multicore processor considered in this paper.

Motivation

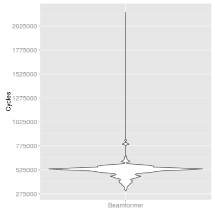


Figure 2: Distribution of the runtime for Beamformer resulting from an exhaustively exploration of the hardware/software co-design space. The application has been partitioned into different number of threads and core compositions.

This shows that finding the right combination of thread mapping and core composition is critical since a wrong choice often leads to the sub-optimal performance (best around 275,000 cycles and majority around 525,000 cycles, Benchmark: Beamformer).

UNIVERSITY OF NORTH CAROLINA CHARLOTTE

Overview

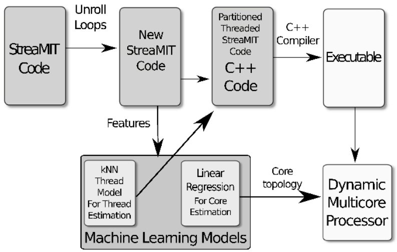

StreaMIT Code
Unroll Loops
New StreaMIT Code
Partitioned Threaded StreaMIT Code
C++ Code
C++ Compiler
Executable
Features
kNN Thread Model For Thread Estimation
Linear Regression For Core Estimation
Machine Learning Models
Core topology
Dynamic Multicore Processor

Figure 3: Description of the workflow. Two distinct machine learning models are used to predict the optimal thread partitioning and core composition based on static code features.

Design Space

Use 16 cores and assign core 0 to the main thread and for runtime management. This leaves 15 cores available for each application. We restrict each core to running only a single thread (no preemptive scheduling) which leads to a possible number of threads between 1 and 15:

Parameters	Values
# of cores in the processor	16
# threads per application	1-15
# cores per thread	1-15
# sampled core compositions	100
# our sampled space	1316
# total sample space	32762

Table 1: Design space considered per application.

Sample Space

The best point found in the sample space of 1,316 points is at least within 5% of the real best in the exhaustive space with 95% confidence[1].

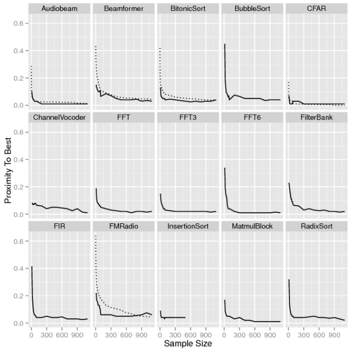


Figure 4: Statistical (plain line) and actual proximity (dotted line)to best performance using a subset of the sample space.

Software Design: Thread Partitioning

We can estimate the optimal number of threads for a benchmark independently of the hardware composition.

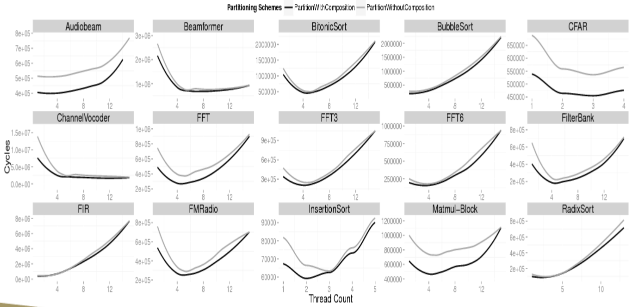


Figure 5: Performance as a function of the number of threads. The performance metric is number of cycles. Each benchmark has the performance measured with cores composed and with threads mapped to a single core.

Hardware Design: Core Composition

For each of the threaded versions we ran the benchmark using on average 100 different compositions. Curves represent the density distribution for different core compositions with modifying # of thread.

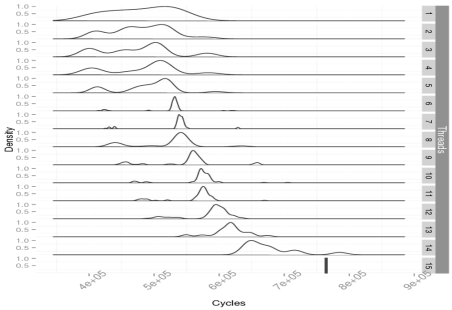


Figure 6: Distribution of Audiobeam performance when modifying the amount of threads and compositions.

Loop Unrolling

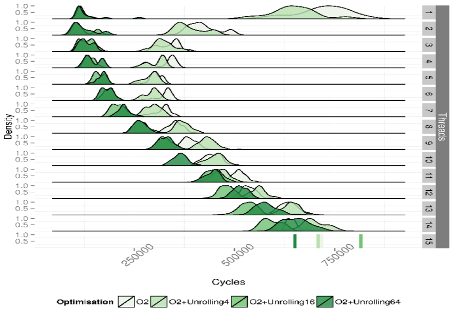


Figure 7: Distribution of FMRadio performance with modifying the amount of threads, core composition and unrolling factor.

Co-Design Space Best Results

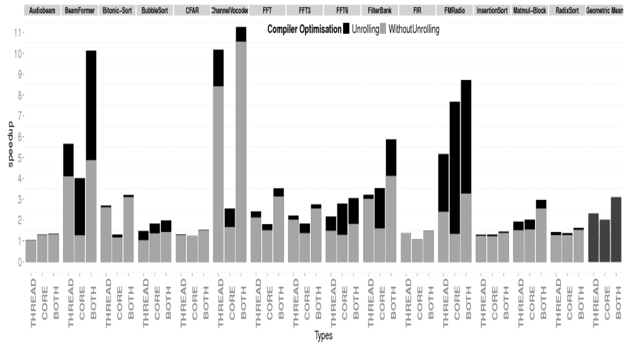
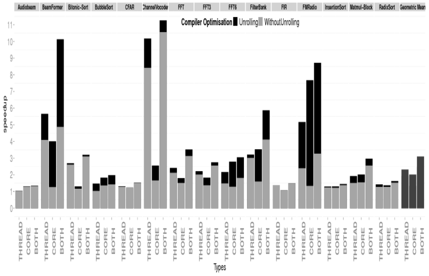


Figure 8: Speedup obtained by choosing best core composition, best thread number and the combination of both optimisations. The baseline for the speedup measurement is single core, single thread execution using O2 compiler optimisations. Higher is better.

UNIVERSITY OF NORTH CAROLINA CHARLOTTE

Co-Design Space Best Results



without unrolling, finding the correct number of threads gives a speedup of 1.92 compared to 1.33 when using only core composition.

Whilst finding the optimal thread mapping is better than the best composition, the best performance is always obtained through a combination of both optimizations.

There is a 3x benefit (overall) by automating the partitioning of both the software (threads) and hardware (cores).

Features Extraction

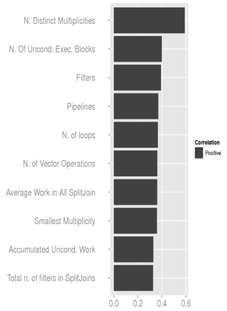


Figure 9: The ten highest correlating features with the best number of threads for 1000 synthetic benchmarks.

In StreamIt the term multiplicity references the number of times a filter will have to execute in a time slice when the graph is in a steady state.

Unconditionally executed blocks represent sets of operations in a filter that will always execute.

UNIVERSITY OF NORTH CAROLINA CHARLOTTE

Features Extraction

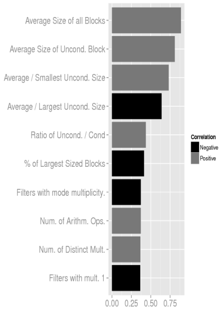


Figure 10: The ten highest correlating features with the optimal number of cores.

Overall there are no features distinct to StreamIt, such as pipelines or splitjoins that correlate highly with the optimal number of cores.

We can thus infer that the optimal number of cores is independent of the structure of a StreamIt program. Instead, it is more dependent on the amount of computation.

UNIVERSITY OF NORTH CAROLINA CHARLOTTE

K-Nearest Neighbor Model for Thread Partitioning

Given a new application to predict, the kNN classifier determines the k closest generated applications in terms of the features. The distance between the features is measured using the Euclidean for each application. K=7 in this study.

Once the set of k nearest neighbors has been identified, the model simply averages the best number of threads for each of the k nearest neighbors to make a prediction.

Linear Regression Model for Core Composition

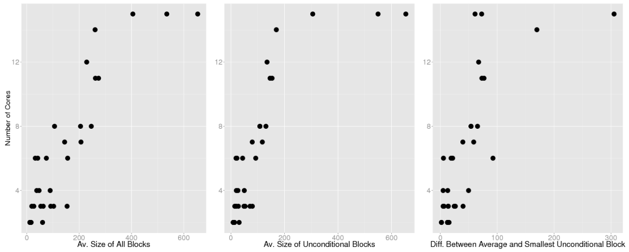


Figure 11: Optimal number of cores in relation to the three highest correlating features. The maximum number of cores plateaus on the right hand side as this is the maximum possible amount.

Evaluation

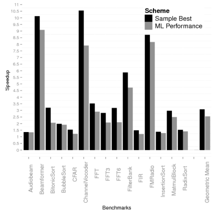


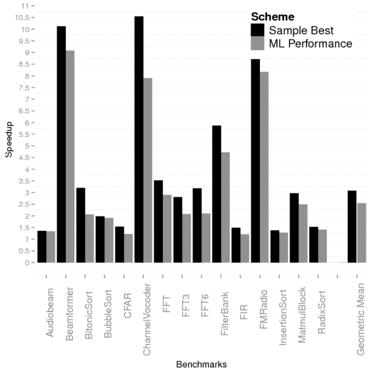
Figure 12: Performance of our machine learning model against the best execution from random sampling. The baseline for the speedup measurement is single core, single thread execution using O2 compiler optimisations. Higher is better.

Average speedup for ML: 2.6

16% smaller than the average of the best found (3.1).

At least within 16% of the total best.

Results Comparison



Q & A

1.If fine-grained composition can allow more optimisation, why haven't we used that?

Types of DMPs such as WidGET and Sharing Architecture are fine-grained level of composition, where cores can be created out of arithmetic logical units, floating point units and memory units and other different components on the processor. This requires **high complexity** for the optimization problem.

2.Is K-fold cross validation more useful than Leave-one-out cross-validation?

Actually LOOCV is a special type of K-fold cross validation, where **k = # of samples**.But deciding which one is better is a trade-off between bias and variance, as well as computation resources available and the cost of mis-classification. That depends on the specific problem. LOOCV is computationally expensive compared with K-fold cross-validation. LOOCV is typically for **small data-sets** and K-fold for relatively **large data-sets**.

Q & A

3.Is there any other streaming language that you are aware of which can be used instead of StreamIt which can produce similar or better results?

The choice of language or framework will depend on the specific requirements of the application, such as performance, scalability, ease of use, and available resources. For example, the functional programming language for signal processing and control applications, Signal. It supports both continuous-time and discrete-time signal processing, making it a good fit for a wide range of streaming applications.

4.What is the importance and effect of predicting the optimal number of threads before predicting optimal core composition?

The model needs to determine the Thread Level Parallelism (TLP) first, using extrated features and uses that information for thread partitioning, and for deciding the best core composition by finding the Instruction Level Parallelism in each thread.

UNIVERSITY OF NORTH CAROLINA CHARLOTTE

Q & A

5.How important is unrolling of the loops and what is the general trend observed while performing it?

Unrolling may increase the degree of parallelism which is advantageous to a wider fused processor. Unrolling generally increases the speed-up for applications.

6.Briefly explain the loop unrolling.

Loop unrolling is a compiler optimization technique that involves duplicating the body of a loop multiple times to reduce the overhead of loop control and increase instruction-level parallelism.

Q & A

7.What is the prevalence of StreamIt in the real world? How popular is it? Is it likely the only or one of the only languages of its type to be used in the future? If so, why?

StreamIt is a programming language specifically designed for programming stream-processing applications. Hard to quantify its popularity. While StreamIt was one of the earliest stream processing programming languages, there have been other languages and frameworks that have emerged in recent years, such as Apache Flink, Apache Kafka Streams, and Apache Spark Streaming.

8.Do we have any real-world applications that have successfully utilized Dynamic multi core processors?

Scientific simulations: Many scientific simulations require intensive computation and can benefit from the parallel processing capabilities of dynamic multi-core processors. For example, simulations in fields such as physics, chemistry, and engineering can be accelerated by using dynamic multi-core processors.

Q & A

9.Is dynamic processor mentioned here is efficient than others like GPU?

The most efficient architecture for a given application depends on a variety of factors such as the nature of the computation, the amount of data, and the available hardware resources.

Dynamic multi-core processors are generally more versatile than GPUs and can handle a wider range of computations, including those that are not well-suited for parallel processing(Recursive algorithms). Additionally, dynamic multi-core processors can dynamically allocate resources to different tasks, which can help to optimize processing efficiency and reduce wasted resources.

However, for tasks that require a high degree of parallelism, such as machine learning and graphics rendering, **GPUs may be more efficient** due to their highly parallelized architecture and specialized hardware.

Q & A

10.Why do streaming programming languages treat programs as graphs?

Streaming applications often involve a large number of data-processing tasks that need to be performed in a particular order. This order can be represented as a graph, where the tasks are the nodes, and the dependencies between tasks are the edges.

In a streaming program, data is continuously processed as it arrives, rather than being processed all at once like in batch processing. This means that the program needs to be able to handle data in real-time and adapt to changes in the data stream. Using a graph to represent the program allows for greater flexibilityin handling data as it flows through the program, as the graph can be dynamically modified to adapt to changes in the data stream.

End-to-end Deep Learning of Optimization Heuristics

Published at PACT 2017

Authors: Chris Cummins (University of Edinburgh, UK)

Presenter: Wangjiaxuan Xin

March 9, 2023

UNIVERSITY OF NORTH CAROLINA
CHARLOTTE

Introduction

Deep neural networks should be able to automatically extract features from source code.

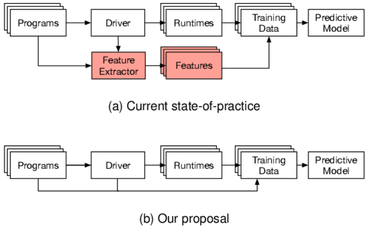


Figure 1: Building a predictive model. The model is originally

Figure 1: Building a predictive model. The model is originally trained on performance data and features extracted from the source code and the runtime behavior. We propose bypassing feature extraction, instead learning directly over raw program source code.

Current Limitations

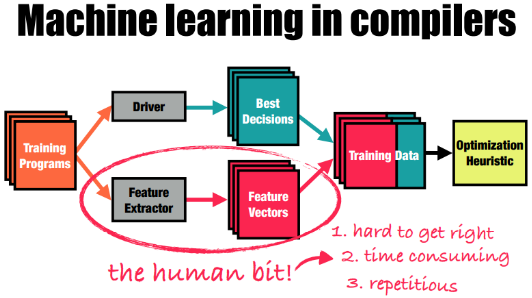


Figure 2: Current State of Practice.

Language Model

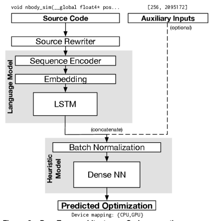


Figure 2: DeepTune architecture. Code properties are ex-

Figure 3: DeepTune architecture. Code properties are extracted from source code by the language model. They are fed, together with optional auxiliary inputs, to the heuristic model to produce the final prediction.

Auxiliary Input: Provide dynamic information which **cannot** be statically determined from the program code.

Embedding: Given a vocabulary size V and embedding dimensionality D, an embedding matrix $\mathbf{W_E} \in \mathbb{R}^{V \times D}$ is learned during training, so that an integer encoded sequences of tokens $\mathbf{t} \in \mathbb{N}^{L}$ is mapped to the matrix $\mathbf{T} \in \mathbb{R}^{L \times D}$. Here the embedding dimensionality D = 64.

Heuristic Model

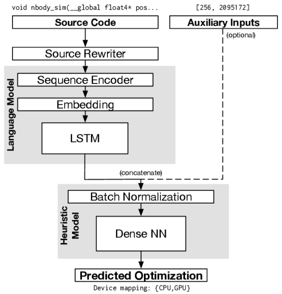


Figure 2: DeepTune architecture. Code properties are ex-

LSTMs: Two-layer implementation receives a sequence of embedding vectors, and returns a single output vector, characterizing the entire sequence.

Two-layer FNN: The first layer consists of 32 neurons with ReLU and the second layer consists of a single neuron for each possible heuristic decision, with Sigmoid Activation Function.

Training the Network

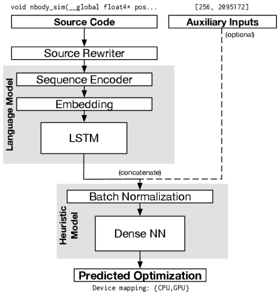


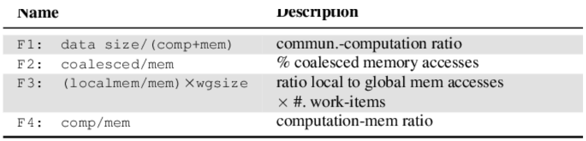
Figure 2: DeepTune architecture. Code properties are ex-

Training Methods: Stochastic Gradient Descent(SGD), using the Adam optimizer.

Loss Function:

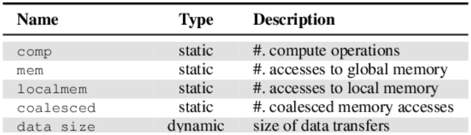
$$\Theta = \underset{\Theta}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} L(X_i, \Theta)$$

OpenCL Heterogeneous Mapping



Name		Description
F1:	data size/(comp+mem)	commun.-computation ratio
F2:	coalesced/mem	% coalesced memory accesses
F3:	(localmem/mem)×wgsize	ratio local to global mem accesses × #. work-items
F4:	comp/mem	computation-mem ratio

(a) Feature values



Name	Type	Description
comp	static	#. compute operations
mem	static	#. accesses to global memory
localmem	static	#. accesses to local memory
coalesced	static	#. coalesced memory accesses
data size	dynamic	size of data transfers

Figure 4: Features used by Grewe et al. to predict heterogeneous device mappings for OpenCL kernels.

OpenCL Thread Coarsening Factor

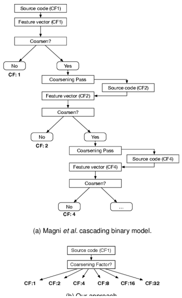


Figure 5: Predicting coarsening factor (CF) of OpenCL kernels. Magni et al. reduce the multilabel classification problem to a series of binary decisions,by iteratively applying the optimization and computing new feature vectors.

Thread Coarsening: Optimization for parallel programs in which the operations of two or more threads are fused together.

SOTA: They implement an iterative heuristic which determines whether a given program would benefit from coarsening. If yes, then the program is coarsened, and the process repeats, allowing further coarsening.

Neural Network Configuration

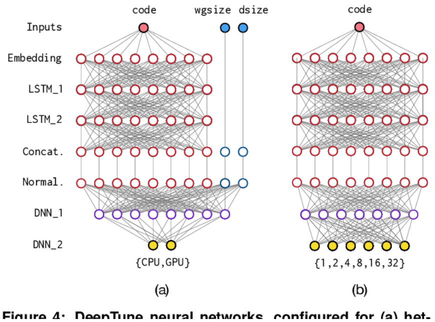
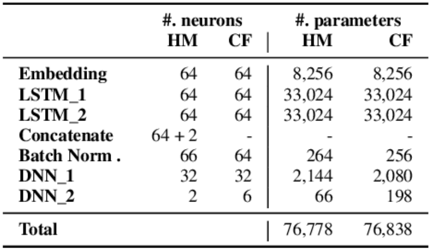


Figure 6: DeepTune neural networks, configured for (a) heterogeneous mapping, and (b) thread coarsening factor. The design stays almost the same regardless of the optimization problem. The only changes are the extra input for (a) and the number of nodes in the output layer.

Parameters Comparison



	#. neurons		#. parameters	
	HM	CF	HM	CF
Embedding	64	64	8,256	8,256
LSTM_1	64	64	33,024	33,024
LSTM_2	64	64	33,024	33,024
Concatenate	64 + 2	-	-	-
Batch Norm .	66	64	264	256
DNN_1	32	32	2,144	2,080
DNN_2	2	6	66	198
Total			76,778	76,838

Figure 7: The size and number of parameters of the DeepTune components of Figure 4, configured for heterogeneous mapping (HM) and coarsening factor (CF).

Two Models' Similarity

The only difference between our network design is the auxiliary inputs for Case Study A and the different number of optimization decisions.

UNIVERSITY OF NORTH CAROLINA CHARLOTTE

Performance Evaluation

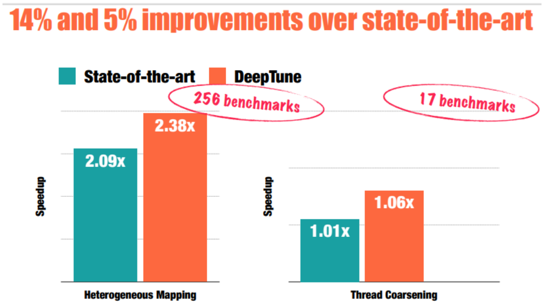


Figure 8: Experiments Results Evaluation in two tasks[1].

Transfer Learning

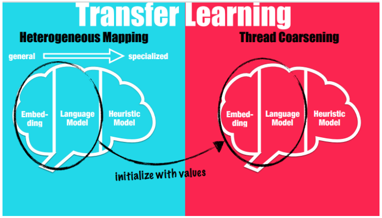


Figure 9: Transfer Learning Approach[2]

Extract the language model, including the **Embedding**, **LSTM_1**, and **LSTM_2 layers**, trained for the heterogeneous mapping task and transfer it over to the new task of thread coarsening.

Since DeepTune keeps the same design for both optimization problems, this is as simple as copying the learned weights of the three layers. Then train the model as normal.

Transfer Learning Results

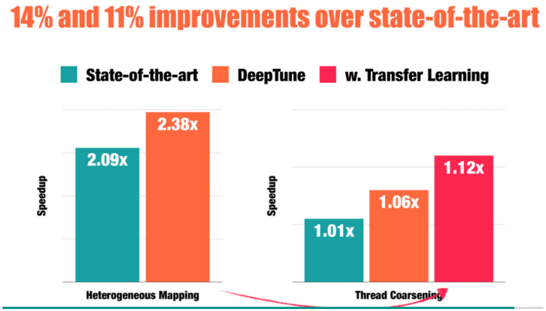
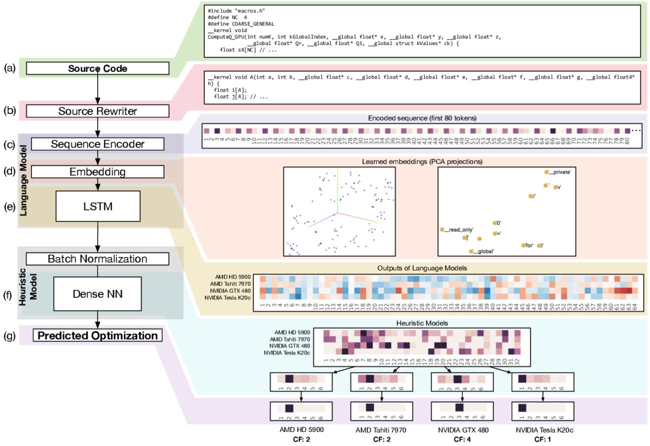


Figure 10: Transfer Laearning Results[3].

Internal State Visualization



```
#include "macros.h"
#define NC  4
#define COARSE_GENERAL
__kernel void
ComputeQ_GPU(int numK, int kGlobalIndex, __global float* x, __global float* y, __global float* z,
             __global float* Qr, __global float* Qi, __global struct kValues* ck) {
    float sX[NC] // ...
```

```
__kernel void A(int a, int b, __global float* c, __global float* d, __global float* e, __global float* f, __global float* g, __global float4*
h) {
  float i[4];
  float j[4]; // ...
```

Figure 9: Visualizing the internal state of DeepTune when predicting coarsening factor for Parboil's mriQ benchmark on four