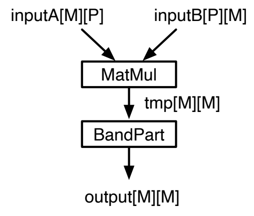
# Dead Iteration Elimination

Cedric Bastoul
Qualcomm France S.A.R.L.
France

Maxime Schmitt
Qualcomm France S.A.R.L.
France

Benoît Meister
Qualcomm Technologies, Inc.
USA

Chandan Reddy
Qualcomm France S.A.R.L.
France



**(a)** High-Level Subgraph

```
// MatMul Operator
for (i = 0; i < M; i++)
  for (j = 0; j < M; j++) {
    tmp[i][j] = 0.;
    for (k = 0; k < P; k++)
      tmp[i][j] += inputA[i][k] * inputB[k][j];
  }
// BandPart Operator (set to upper triangular)
for (i = 0; i < M; i++)
  for (j = i; j < M; j++)
    output[i][j] = tmp[i][j];
```

**(b)** Fused Operator Code Corresponding to **(a)**

```
// Fused MatMul-BandPart Operator after DIE
for (i = 0; i < M; i++)
  for ( j = i ; j < M; j++) {
    tmp[i][j] = 0.;
    for (k = 0; k < P; k++)
      tmp[i][j] += inputA[i][k] * inputB[k][j];
  }
for (i = 0; i < M; i++)
  for (j = i; j < M; j++)
    output[i][j] = tmp[i][j];
```

**(c)** Fused Operator Code After DIE

**Figure 1.** Goal: Automatically Generate **(c)** From **(b)** and Information From **(a)** to Remove Dead Iterations

## Abstract

Dead code elimination (DCE) is a compiler technique that aims at increasing the program performance and reducing the executable size by removing program parts which do not contribute to the program output. Usual DCE implementations identify instructions not producing live-out data and remove them entirely. However, it may happen that only some dynamic executions of the instruction are dead. This situation notably arises when an instruction is enclosed inside a loop and some iterations of that loop do not contribute to the program output.

In this paper we present dead iteration elimination (DIE), a technique to identify and to remove those iterations. DIE relies on polyhedral analysis to compute the required data space and the iterations which contributed them. It enables the safe removal of parts of the iteration space, possibly up to complete statement removal, in a complementary way to DCE. It also makes it possible to consider required output specification which opens new applications such as automatic specialization, sparsification or subsampling.

## 1 Introduction

There exists a number of situations where only a subset of the dynamic executions of an instruction actually contributes to a program output. In such cases, dead code elimination (DCE) techniques fail at removing them because they address only complete instruction removal. The problem may be particularly prominent when applications are built from a limited set of pre-defined high-level operators, such as in artificial intelligence and deep learning frameworks. In this context, the output of some operators may not be used entirely by subsequent operators. An illustration is shown in Fig. 1**(a)** where the composition of MatMul and BandPart TensorFlow-like operators filters only upper triangular elements of a matrix-multiplication output. The compound operator code built from the concatenation of each operator code is shown in Fig. 1**(b)**. It features many "dead iterations", i.e. loop iterations not contributing to the program output. In this paper we present a polyhedral compilation approach called dead iteration elimination (DIE) to remove them and generate the code in Fig. 1**(c)**.[1] DIE enables a number of advances:

1. We achieve finer-grain optimization w.r.t. DCE by acting at the loop iteration level rather than at full statement level, allowing the composition of AI/DL operators with auto-removal of non-pertinent computation, reducing the need for specialized custom operators;
2. We apply full statement removal in dead iteration space situations that could not be identified by DCE, in a complementary way;
3. We enable new applications when a specification of the output dataspace exists, e.g., specialization, sparsification or subsampling of AI/DL operators which inactivate parts of their original computation space.
4. We provide static analysis information exposing potential programming mistakes during software development, e.g., ill-formed loops or out-of-bound accesses.

## 2 Dead Iteration Analysis and Removal

DIE's flow is depicted in Fig. 3. It takes as input the program code as well as an optional specification of the desired output data, i.e., which parts of the data space the code should actually compute. This specification can be used to specialize the code (e.g., to compute only a given data tile) and/or to inject regular sparsity or subsampling information (e.g., to compute only half the data according to a checkerboard

[1]In this example, the temporary tensor tmp may be removed as well using specific analysis oustide the scope of this paper.

**Algorithm 1** Dead Iteration Space Analysis

```
Input: Augmented & inverted data-dependence graph (see step (c))
Output: Dead iteration space for each node
1: for each vertex $v$, initialize $R_{v,t}$ the required data space for the tensor $t$: do
2:     Entry vertex: $R_{entry,t}$ = required data space of each output tensor $t$
3:     Other vertices: empty spaces
4: end for
5: for each vertex $v$ (that has iteration domain $\mathcal{D}_v$, writes tensor $w$ with function $f_w$, and reads tensors $r_i$
   with function $f_{r_i}$) according to the order defined in step (c) do
6:     Compute the vertex potential contribution space for $w$: $\mathcal{P}_{v,w} = \cup_{i \in predecessor(v)} \mathcal{R}_{i,w}$
7:     Compute the vertex contributing space for $w$: $C_{v,w} = Preimage(\mathcal{P}_{v,w}, f_w) \cap \mathcal{D}_v$
8:     Compute the required data spaces for $v$: $\{\mathcal{R}_{v,r_i} = Image(C_{v,w}, f_{r_i})\}$
9:     Compute the dead iteration space for $v$: $Dead_v = \mathcal{D}_v - C_{v,w}$
10: end for
```

**Figure 2.** Dead Iteration Elimination Analysis Algorithm

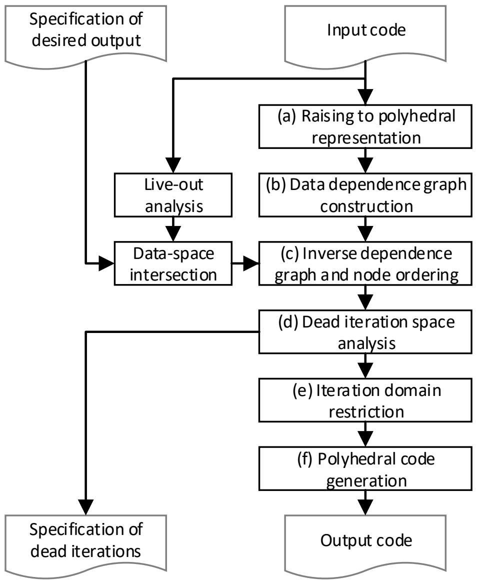

Specification of desired output
Input code
(a) Raising to polyhedral representation
Live-out analysis
(b) Data dependence graph construction
Data-space intersection
(c) Inverse dependence graph and node ordering
(d) Dead iteration space analysis
(e) Iteration domain restriction
(f) Polyhedral code generation
Specification of dead iterations
Output code

**Figure 3.** Dead Iteration Elimination Processing Flow

layout). That specification is combined with an automatic live-out analysis to form the *required data space*. Iterations not contributing directly or indirectly to a write on that required data space are dead. DIE follows a 6-step process to generate both an output code cleared from dead iterations and a specification of the dead iteration space for further analysis (e.g., to generate compiler warnings).

The principle of DIE is to back-propagate constraints on the required data spaces along the data dependence graph from nodes producing the output data to nodes reading the input data. We eliminate cycles by collapsing each strongly connected component into a single node, over-approximating its requirements as the union of all its input domains. Handling he special case of self-dependency with distance zero, e.g., += operator, by removing the cycle is sufficient to handle the class of programs we are addressing without introducing an over-approximation. Those constraints on data translate to constraints on the iteration space that we restrict to remove dead iterations. The consecutive steps are as follows:

**(a) Raising to polyhedral representation** achieves extraction of polyhedral representation, including iteration domains, data access functions, and scheduling of the original program as offered by polyhedral compilers [5, 9, 14].

**(b) Data dependence graph construction** builds the data dependence graph (DDG) [4] and its strongly connected component graph (enabling a convenient node ordering during the next step). Each strongly connected component corresponds to a compound statement writing/reading all data references written/read in the original nodes. In the following, they are considered as single nodes.

**(c) Inverse dependence graph and node ordering** finds a convenient node traversal ordering for backward propagation of constraints. First, it builds an inverse dependence graph by (1) inverting all edges of the graph computed in the previous step, (2) adding an "entry" node with edges from that node to all nodes writing to a required output, and (3) adding an "exit" node with edges from all nodes reading an input data to that node. Finally it computes a topological order for that graph, not taking into account self-dependencies.

**(d) Dead iteration space analysis** is the core of the process and is detailed in Fig. 3(right). Without loss of generality and for clarity reasons, we suppose each node writes only one tensor, as it can easily be generalized. Starting from the required data space specified for each output tensor, the algorithms walks back the DDG up to the statements reading the input data, and uses polyhedral operations such as image or preimage by the access functions [12] to move from data space to iteration space and conversely.

For each node, we first compute the potential contribution space, i.e. the parts of the required data space that node may contribute to. Then we compute the contributing space, i.e., the parts of the iteration space actually contributing. Next we compute the required data space, i.e., the parts of the data space that node requires. Finally we compute the dead iteration space as the difference between the iteration domain and the contributing space.

**(e) Iteration domain restriction** replaces each statement iteration domain with the difference between that iteration domain and the dead iteration space for that statement.

**(f) Polyhedral code generation** finally produces the output code without dead iterations from the modified representation, using polyhedral code generation techniques [1].

## 3 Related Work

Dead code elimination is a classical optimization present since early compilers [6]. Modern solutions are based on static single assignment form, identifying instructions not producing live-out data to remove them entirely [2]. Partial dead code removal is possible by moving that code to branches where it actually contributes to the output while it is dead in others [7]. Differently, we propose to reason and to remove partial dead code at the loop iteration level, with the support of polyhedral techniques. Sharing data-space analysis, Feautrier addressed the dual problem for a single reference [3] and overlapped tiling computes parts of a tiled iteration space contributing to tiles [8]. Our work differs as the analysis targets full kernels or functions and applies to conservative and efficient iteration removal.

Verdoolaege suggested a dead code elimination approach by iteratively applying dataflow analysis [13]. As this approach may add only isolated iterations at each step, it is not practical unless the number of iterations is small and it is not applicable when the number of iterations is parametric. Verdoolaege suggests to use widening to overcome those limitations, removing full loops only (not supporting, e.g., the example in Fig. 1). Differently, our approach reasons on spaces for excellent scaling and support of parametric loops: it applies to subsets of iteration domains, enabling sparsification and subsampling scenarios.

Dead code elimination based on polyhedral representations also appear in implemented forms in source code of Polly [5] where it is described as related to PPCG's approach [13], or of AlphaZ [15] in the context of systems of affine recurrence equations to ensure that AlphaZ generates a C code that covers only the useful data/iterations of a program. The implementation nature of those related work makes it difficult to make a precise comparison, instead our paper explicits a practical approach to achieve this task.

## 4 Conclusion

This paper presents an approach to dead iteration elimination, leveraging polyhedral analysis to generate a code where loop iterations not contributing to the output are removed. Our technique supports desired output data specification, enabling a number of applications including code specialization, sparsification and subsampling. A notable application in the context of large language models based on the transformer architecture [11] is the removal of all iterations not participating to the computation of the output logits (only the last row of the logit matrix being useful in most scenarios), strongly reducing the cost of the last decoder layer (about 3% overall time reduction of prompt processing for LLaMa 3.1 8B [10]). Ongoing work aims at removing dead output dependencies.

## References

[1] Cédric Bastoul. 2004. Code Generation in the Polyhedral Model Is Easier Than You Think. In *PACT'13 IEEE International Conf. on Parallel Architecture and Compilation Techniques*. Juan-les-Pins, France, 7–16.

[2] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (oct 1991).

[3] P. Feautrier. 1988. Parametric Integer Programming. *RAIRO Recherche Opérationnelle* 22, 3 (1988), 243–268.

[4] P. Feautrier. 1991. Dataflow Analysis of Scalar and Array References. *International Journal of Parallel Programming* 20, 1 (Feb. 1991), 23–53.

[5] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Grösslinger, and Louis-Noël Pouchet. 2011. Polly-Polyhedral optimization in LLVM. In *IMPACT 2011 First International Workshop on Polyhedral Compilation Techniques*. Chamonix, France.

[6] Kenneth W. Kennedy. 1973. *Global dead computation elimination*. Technical Report. Tech. Rep. SETL Newsl. 111, Courant Institute of Mathematical Sciences, New York Univ., New York, N.Y.

[7] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. 1994. Partial dead code elimination. *ACM Sigplan Notices* 29, 6 (1994), 147–158.

[8] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. 2007. Effective Automatic Parallelization of Stencil Computations. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California) *(PLDI '07)*. 235–244.

[9] Benoît Meister, Nicolas Vasilache, David Wohlford, Muthu Manikandan Baskaran, Allen Leung, and Richard Lethin. 2011. R-Stream Compiler. In *Encyclopedia of Parallel Computing*. 1756–1765. Now Qualcomm® Polyhedral Mapper[2]

[10] Llama Team. 2024. The Llama 3 Herd of Models. arXiv:2407.21783

[11] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st Intl. Conf. on Neural Information Processing Systems* (Long Beach, California). 6000–6010.

[12] Sven Verdoolaege. 2010. *isl*: An Integer Set Library for the Polyhedral Model. In *ICMS 2010, Third International Congress on Mathematical Software*, Vol. 6327. Kobe, Japan, 299–302.

[13] Sven Verdoolaege. 2015. *PENCIL support in pet and PPCG*. Technical Report RT-0457. INRIA Paris-Rocquencourt.

[14] Sven Verdoolaege and Tobias Grosser. 2012. Polyhedral Extraction Tool. In *IMPACT'12 International Workshop on Polyhedral Compilation Techniques*. Paris, France.

[15] Tomofumi Yuki, Gautam Gupta, DaeGon Kim, Tanveer Pathan, and Sanjay Rajopadhye. 2013. AlphaZ: A System for Design Space Exploration in the Polyhedral Model. In *Languages and Compilers for Parallel Computing*. Springer, 17–31.

[2]Qualcomm Polyhedral Mapper is a product of Qualcomm Technologies, Inc. and/or its subsidiaries.