Diesel: DSL for Linear Algebra and Neural Net Computations on GPUs

Venmugil Elango
NVIDIA
USA
velango@nvidia.com

Norm Rubin
NVIDIA
USA
nrubin@nvidia.com

Mahesh Ravishankar
NVIDIA
USA
mravishankar@nvidia.com

Hariharan Sandanagobalane
NVIDIA
USA
haris@nvidia.com

Vinod Grover
NVIDIA
USA
vgrover@nvidia.com

Abstract

We present a domain specific language compiler, Diesel, for basic linear algebra and neural network computations, that accepts input expressions in an intuitive form and generates high performing code for GPUs. The current trend is to represent a neural network as a computation DAG, where each node in the DAG corresponds to a single operation such as matrix-matrix multiplication, and map the individual operations to hand tuned library functions provided by standard libraries such as CuBLAS and CuDNN. While this method takes advantage of readily available optimized library codes to achieve good performance for individual operations, it is not possible to optimize across operations. As opposed to this, given a computation composed of several operations, Diesel generates (a set) of efficient device functions, where the code is optimized for the computation as a whole, using polyhedral compilation techniques. In addition, there are cases where the code needs to be specialized for specific problem sizes to achieve optimal performance. While standard libraries are written for parametric problem sizes (where problem sizes are provided at runtime), Diesel can accept problem sizes at compile time and generate specialized codes. Experimental results show that the performance achieved by Diesel generated code for individual operations are comparable to the highly tuned versions provided by standard libraries, while for composite computations, Diesel outperforms manually written versions.

MAPL'18, June 18, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5834-7/18/06…$15.00
https://doi.org/10.1145/3211346.3211354

CCS Concepts • **Computing methodologies** → **Parallel computing methodologies**; *Neural networks*;

Keywords Domain specific language (DSL) compiler, Polyhedral model, Linear algebra, Neural Networks.

ACM Reference Format:
Venmugil Elango, Norm Rubin, Mahesh Ravishankar, Hariharan Sandanagobalane, and Vinod Grover. 2018. Diesel: DSL for Linear Algebra and Neural Net Computations on GPUs. In *Proceedings of 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL'18).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3211346.3211354

1 Introduction

GPUs play a major role in scientific computations, and in neural network training and inference. Regular computational structure of scientific computing codes makes GPGPUs a good fit to perform these computations with high efficiency and low energy. Matrix-matrix multiplication, and many other linear algebra computations form the basis for several scientific computation and neural network algorithms. Hence, having efficient implementations of basic linear algebra computations for different GPU architectures is critical for the overall performance. With evolving architectures, high performing linear algebra kernels have to be written and tuned for every GPU architecture. Current trend is to manually write these kernels for each individual architecture in GPU's assembly-level language (SASS), and hand tune them for the best performance. This involves significant and continuous manual effort.

In addition, scientific computation codes are generally composed of various sub-computations in different combinations. For example, a part of training a neural network involves performing a matrix-multiplication, followed by adding a bias, and applying an activation function. Libraries such as CuBLAS and CuDNN provide efficient implementations for each of these individual sub-computations. In order to take advantage of these libraries, the programmer needs to build his/her code by calling these library functions one after the other, with intermediate results moved in and out of

global memory between different calls. Several of these unnecessary data movements can be eliminated if the libraries provided functions with different sub-computations composed into a single kernel. There are exponentially many possible combinations of sub-computational sequences to be optimized, making it impractical to manually write and hand-tune all possible combinations. Further, there are several use cases (especially in ML), where it is important to specialize implementation of these kernels for specific problem sizes. This heavily motivates the need to automate the process of generating efficient kernels for these computations.

Diesel is a domain specific language compiler that automatically generates efficient code for computations involving basic linear algebra and neural net operations. Diesel takes the user expressions in a simple language (whose grammar is shown in Sec. 3), and generates an efficient CUDA code corresponding to the input expressions. Diesel also exposes these functionalities as a set of API's allowing it to be integrated with other tools. The key contributions include

- a user-friendly interface that allows users to express a sequence of BLAS and neural net computations using an intuitive grammar,
- auto-generate efficient GPU device functions (or kernels) for different operations with minimal user interaction,
- generate efficient GPU kernels for different operations, whose performances are comparable to standard libraries,
- allow easy tuning of parameters, such as tilesizes, manually or through an autotuner, and automatically produce kernels corresponding to different configurations,
- produce efficient fused kernels for a sequence of operations.

Diesel internally uses polyhedral machineries to perform its code optimizations, so that different optimizations are encoded abstractly in a mathematical form. This allows Diesel to apply similar transformations on a wide range of problems, and reuse the effort across different architectures. As opposed to the current trend, this eliminates the redundant work of having to optimize the kernels for each architecture separately.

Background on polyhedral model is summarized in Sec. 2. Various optimizations, and specifics of code generation perfomed by Diesel are detailed in Sec. 3. Experimental results comparing the performance of Diesel generated code against different manually optimized versions are presented in Sec. 4.

2 Background

This section provides background on GPU architecture and the polyhedral model.

2.1 GPU Architecture and CUDA Programming Model

A GPGPU is typically used to accelerate data parallel codes. The architecture of GPU is composed of several streaming multiprocessors (SMs). Each SM has several processors (SIMD units) that share a single instruction unit. The processors within an SM communicate through a fast on-chip *shared memory*, while the different SMs communicate through a slower off-chip DRAM, also called *global memory*. Each multiprocessor unit also has a fixed number of registers. Programming GPGPUs is enabled through CUDA programming model. CUDA is an extension of C programming language, that allows users to define device functions, called *kernels*, that can be executed on a GPU. The CUDA programming model abstracts the processor space as a grid of *thread blocks* (that are mapped to multiprocessors in the GPU device), where each thread block is a grid of *threads* (that are mapped to SIMD units within a multiprocessor). Threads in a thread block are divided into SIMD groups called *warps*. Different threads within a warp are referred to as *lanes*. Lanes in a warp are executed in a lockstep fashion. If the lanes in a warp execute different code paths, only those that follow the same path can be executed simultaneously and a penalty is incurred. This execution model is called Single Instruction Multiple Thread (SIMT) model.

2.2 Polyhedral Model

Diesel uses polyhedral model to find efficient schedules and perform high-level optimizations. Polyhedral compilation techniques are applicable for a sub-class of functions called *Static Control Parts* (SCoPs). A part of a program is a SCoP if it only consists of `for`-loops with constant strides, and `if`-conditions, whose conditional expressions, and loop bound expressions are affine functions of program parameters and surrounding loop induction variables. Further, the array access functions of the statements should be expressible as affine functions of program parameters and surrounding loop induction variables. Linear algebra computations typically fall into this category. Fig. 1 shows the SCoP that corresponds to an implementation of a standard matrix-matrix multiplication. Polyhedral model captures the execution of SCoPs in a compact form as sets and maps.

Notation We use ISL's terminology [9] to describe sets and maps. Using ISL notation, sets are denotated by: `[n] -> {S[i]: 0<=i<n}`. The LHS of `->` in this example is an optional list of parameters needed to define the set. `S[i]` models a set with one dimension `i`, and the space in which the set resides is named `S`. Presburger formulae are used on the RHS of ":" to model the points belonging to the set. These sets are disjunctions of conjunctions of Presburger formulae, thereby modeling unions of convex and strided integer sets.

Maps are denoted by: `[n] -> {S1[i]->S2[j]: 0<=i,j<n and j=i+1}`. Relationship between domain and range is also

specified using Presburger formulae. The example relates a one-dimensional set S1[i] to the set S2[j] such that the points in the domain and range are related by the expression $j = i + 1$.

```
for (i=0; i<N; i++)
 for (j=0; j<N; j++)
  for (k=0; k<N; k++)
   S1: C[i][j] += A[i][k] * B[k][j];
```

Figure 1. Matrix-matrix multiplication.

Iteration domain Iteration domain of a statement S represents the set of statement instances of S that are executed at runtime. Iteration domains for statements within a SCoP can be statically derived. For instance, iteration domain of the statement S1 in Fig. 1 is given by the set $\mathcal{D}_{S1}$:= `[N] -> {S1[i, j, k] : 0 <= i,j,k < N}`.

Access function Access functions map statement instances to the memory locations from where data is read or written to. Access functions are represented by affine maps from iteration domain to data sets. In our example, read from array A is given by the map $\mathcal{F}^R_{S1}$:= `[N] -> {S1[i, j, k] -> A[i, k] : 0 <= i,j,k < N}`.

Schedule The order in which the statement instances are executed in a program is given by a schedule. An affine schedule ϕ_S for a statement S is an affine map that assigns a multi-dimensional integer value to each point in the iteration domain of S. A statement instance s_1 of S is executed before an instance s_2 iff $\phi_S(s_1) \prec \phi_S(s_2)$, i.e. , the value $\vec{v_1}$ assigned by ϕ_S to s_1 is lexicographically smaller than the value $\vec{v_2}$ assigned to s_2. The affine schedule corresponding to the execution order provided by the code segment in Fig. 1 for statement $S1$ is given by `[N] -> {S1[i, j, k] -> [i, j, k] : 0 <= i,j,k < N}`.

Data dependencies There exists a data dependence between two statement instances if they both access the same memory location. Affine data dependencies are characterized by affine maps that map the source iteration domain points to their targets. In our example in Fig. 1, value written to array C by the statement S1 at a timestamp [i,j,k] is later read during [i,j,k+1]. Hence, there is a RAW data dependence between the points in the iteration domain of S1, which is denoted by $\mathcal{P}_{S1}$:= `[N] -> {S1[i, j, k] -> S1[i, j, k + 1] : 0 <= i,j < N and 0 <= k < N-1}`.

Optimizing a SCoP mainly involves determining an efficient schedule for execution of its statement instances under a given objective such as maximizing parallelism, minimizing data movement, etc. Data dependencies play an important role in this by constraining the possible schedules that can be derived. Any valid schedule ϕ_S for a statement S must respect RAW, WAW and WAR dependencies, i.e. , if an iteration vector $\vec{v_1}$ has a RAW dependence on vector $\vec{v_2}$, then $\phi_S(v_1) \prec \phi_S(v_2)$ (where $\prec$ denotes lexicographical order). Further, read-after-read (RAR) dependencies indicate reuse of data. Any schedule ϕ_S that aims to improve data locality would try to reduce the dependence distances, $\delta_S = \phi_S(\vec{v_2}) - \phi_S(\vec{v_1})$, where $\mathcal{P}_S(\vec{v_1}) = \vec{v_2}$.

3 Diesel Compiler

Diesel is composed of three parts – 1) a *frontend* that parses the input program, and converts the program into its polyhedral representation; 2) *scheduler* that computes an efficient computation schedule corresponding to the input program for the target GPU; 3) a *code generator* that generates device functions from the schedule computed by the scheduler. Each of these parts are detailed in the following sub-sections.

3.1 Frontend

Frontend of Diesel accepts input program from the user and converts it into an equivalent polyhedral intermediate representation (IR), which is then passed to the scheduler for optimization. Diesel supports an intuitive syntax to represent linear algebra expressions. Fig. 2 shows an example input Diesel code that performs a matrix-matrix multiplication (GEMM), followed by a matrix addition. The user defines the input matrices, along with its dimensions, using the keyword Matrix. Output and intermeditate matrices need not be explicitly defined. Dimenions of the output and intermediate matrices are automatically derived by Diesel using domain specific knowledge as shown later. The keyword Params is used to define parameters such as problem sizes. Finally, a call to CodeGen is made with the datatype of computation, and a set of live-out matrices. Live-out matrices are the ones that will be stored to the global memory at the end of the computation. The complete grammar is provided in Fig. 3.

```
Params M, K, N;
Matrix A(M,K);
Matrix B(K,N);
Matrix C(M,N);
D = A * B;
E = D + C;
CodeGen(FLOAT, {E});
```

Figure 2. Example input program.

In addition, Diesel library also exposes a set of C APIs allowing other libraries to directly link to diesel to generate optimized GPU kernels. This is shown in Fig. 4.

3.1.1 Constructing DAG from Input Program

In order to convert the input program into its polyhedral representation, Diesel first constructs an expression DAG. Fig. 5 shows the DAG corresponding to the program in Fig. 2.

```
dsl           : decls stmts codegen;

decls         : /*empty*/ | decls decl;
decl          : 'Params' param_list ';'
              | inp_mat ';';

param_list    : VAR | param_list ',' VAR;
inp_mat       : mat_decl mat_dim
              | mat_decl mat_param
mat_decl      : 'Matrix' VAR
mat_dim       : '(' NUM ',' NUM ')'
mat_param     : '(' VAR ',' VAR ')';

stmts         : /*empty*/ | stmts stmt;
stmt          : VAR '=' expr ';';
expr          : VAR
              | expr '+' expr
              | expr '-' expr
              | expr '*' expr
              | expr TRANS | '(' expr ')'
              | pointwise '(' expr ')';
pointwise     : RELU | SIGMOID | TANH;

codegen       : /*empty*/
              | 'CodeGen' '(' type ','
                '{' out_mat_list '}'')' ';';
type          : 'INT' | 'HALF'
              | 'FLOAT' | 'DOUBLE';
out_mat_list  : VAR | out_mat_list ',' VAR;
```

Figure 3. Input grammar for Diesel.

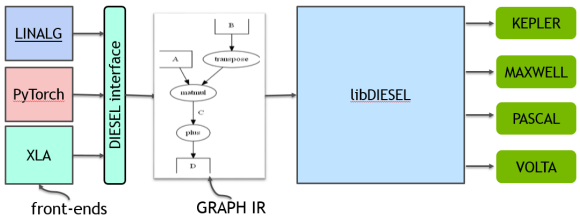


Figure 4. Overview of Diesel's compilation process.

Nodes represent input matrices, and operations. Dependencies are captured by the edges between nodes. The primary purpose of building an expression DAG is to propagate necessary information between the nodes, in order to extract the polyhedral representation of the computation. To assist this process, each node is associated with a polyhedron, called *dataset*, and an affine map, called *datamap*. As detailed later, datasets and datamaps are used to derive inter-statement data dependencies, and data layout transformations (DLT), respectively.

Notation In the remainder of this section, to refer to the node N in a DAG corresponding to an expression e in the input program, we use the notation $N[e]$. When the label of the node is irrelevant or clear from the context, we simply use $[e]$ to refer to the node corresponding to e. For instance, $N_1[(N_2[e])^T]$ indicates that the node N_2 corresponds to the expression e, while N_1 corresponds to e^T.

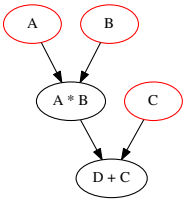


Figure 5. DAG for the expression sequence `D=A*B; E=D+C;`. Nodes colored in red correspond to input matrices; the remaining nodes represent operations.

Datasets Each node has an associated polyhedron called *dataset*, that contains the information about the space needed to hold its data. For instance, a node corresponding to an input matrix `Matrix A(M,K)` has an associated dataset `[M,K] -> {A[i,j] : 0 <= i < M and 0 <= j < K}`, which is a two-dimensional set with each dimension constrained by the matrix size. Dimensionality and sizes of the datasets for input nodes are obtained from declarations specified by the user. Datasizes of remaining nodes in a DAG are derived using the rules shown in Table 1.

Table 1. Rules to propagate data size information in the expression DAG. Subscripts of each node represent sizes of the node's dataset.

$[e_1]_{m,n} + [e_2]_{m,n}$	$\rightarrow$	$[e_1 + e_2]_{m,n}$
$[e_1]_{m,n} - [e_2]_{m,n}$	$\rightarrow$	$[e_1 - e_2]_{m,n}$
$[e_1]_{m,p} * [e_2]_{p,n}$	$\rightarrow$	$[e_1 * e_2]_{m,n}$
$[e_1]^T_{m,n}$	$\rightarrow$	$[{e_1}^T]_{n,m}$
$\texttt{ReLU}([e_1]_{m,n})$	$\rightarrow$	$[\texttt{ReLU}(e_1)]_{m,n}$
$\texttt{Sigmoid}([e_1]_{m,n})$	$\rightarrow$	$[\texttt{Sigmoid}(e_1)]_{m,n}$
$\texttt{Tanh}([e_1]_{m,n})$	$\rightarrow$	$[\texttt{Tanh}(e_1)]_{m,n}$

Datamaps Datamaps are affine functions that capture data-layout information of datasets. For instance, consider the expression `C = A^T * B;`. The expression `A^T` semantically indicates that the elements of matrix `A` are just reordered to

their transposed locations, without any change in their values. Hence, both expressions `A` and `A^T` can share the same physical memory space. This information is captured using the datamap `{AT[i,j] -> A[j,i] : }`, representing transpose DLT. In Diesel, datamap, $\mathcal{M}$, of all types of nodes, except a transpose (`^T`) node, is an identity map from its dataset to itself. The rule to build datamap for a transpose node N_1 is: $\mathcal{M}\left(N_1[(N_2[e_1]^T)]\right) := \mathcal{M}(N_2) \circ$ `{M_1[i,j]->M_2[j,i]:}`, where, `M_1[i,j]` and `M_2[j,i]` are datasets of nodes N_1 and N_2, respectively.

In neural net computations, performing GEMM followed by *bias addition* is a common operation, where the output matrix M from GEMM is added with a bias vector v as follows: $M_{i,j} = M_{i,j} + v_j$. In Diesel, this computation is represented as shown in Fig. 6. The temporary matrix `T` in Fig. 6 is of size 128×128, while `V` is of size 1×128. In order to make the sizes of the two operands of `+` operator compatible, Diesel internally "broadcasts" `V` into a matrix of size 128×128. Hence, the computation in Fig. 6 becomes `(A * B) + Bcast(V)` [1]. The broadcast semantics of Diesel is same as numpy's broadcast operation [2]. To prevent redundant replication of the elements of `V`, Diesel uses datamaps to represent broadcast. Datamap construction rule for broadcast node is: $\mathcal{M}\left(N_1[\text{Bcast}(N_2[e_1]_{m2,n2})]_{m1,n1}\right) := \mathcal{M}(N_2) \circ R$, where $R =$ `{M_1[i,j]->M_2[0,j]:}` when $m2 = 1 \wedge n1 = n2$, and $R =$ `{M_1[i,j]->M_2[i,0]:}` when $m1 = m2 \wedge n2 = 1$. In the final code generated by Diesel, elements of `V` are not replicated. Instead, the datamap makes sure that the access functions are correctly updated to point to the correct locations in the original matrix `V`.

```
Matrix A(128,128);
Matrix B(128,128);
Matrix V(1,128);
T = A * B;
C = T + V;
CodeGen(FLOAT, {C});
```

Figure 6. Matrix multiplication followed by bias addition. Matrices `T` and `V` are of different sizes. Diesel internally performs "broadcast" of `V` to make the sizes compatible.

3.1.2 Converting DAG to Polyhedral IR

Extracting the polyhedral IR of the input program involves building iteration domains, access functions, and data dependence relations corresponding to the expressions. Iteration domain and data dependence information is required to find a valid schedule for the computation, while access functions are needed for data layout transformations, and memory promotions.

[1]Note that the Bcast operator is not exposed to the user; instead Diesel automatically takes care of inserting broadcasts when applicable.
[2]https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html

Iteration domains As detailed in Sec. 2, iteration domains mathematically capture the operation instances of expressions. Iteration domains of various operations in Diesel are constructed using domain specific knowledge, as shown by the rules in Table 2. Iteration domains of different nodes live in different set-spaces, and are identified by their space names.

Table 2. Iteration domains for different node types. m, n, and p are integer constants. Set-spaces (denoted as `D_x` in the table) are unique for each DAG node, where `x` corresponds to the unique node-id.

Node	*Iteration domain ($\mathcal{D}$)*
$N_1[e_1 \oplus e_2]_{m,n}$ $\oplus \in \{+,-\}$	`[m,n] -> {D_1[i,j]:0<=i<m and 0<=j<n}`
$N_1[[e_1]_{m,p} * [e_2]_{p,n}]_{m,n}$	`[m,n,p] -> {D_1[i,j,k]:0<=i<m and 0<=j<n and 0<=k<p}`
$N_1[\text{OP}(e_1)]_{m,n}$ OP $\in$ {`ReLU`, `Sigmoid`, `Tanh`}	`[m,n] -> {D_1[i,j]:0<=i<m and 0<=j<n}`

Access functions Each Diesel operation is associated with two access maps – read, and write access maps. Read maps capture read information of an operation by relating its iteration domain points to datasets of its operands, while write maps are used to capture write information by relating iteration domain points to its own dataset. Aliases between datasets for the operands are resolved with the help of datamaps. Rules to build read accesses are shown in Table 3. Write access maps for all the operations except multiplication (`*`) are identity maps from their iteration domains to their datasets. Write access map for `*` is given by $N_1[e_1 * e_2] :=$ `{D_1[i, j, k] -> M_1[i, j]:}`.

Data dependencies Intra-node read-after-write (RAW) data dependencies can be directly constructed based on domain specific knowledge as follows: For any node N other than multiplication node, $\mathcal{P}_N := \emptyset$; for multiplication (`*`), $\mathcal{P}_N :=$ `{D[i,j,k] -> D[i,j,k+1] :}`. In order to build inter-node dependences, we first define *last-writer* map of a node as a relation from the subset of its iteration domain that writes the final values to its dataset. Last-writer map for a node N can either be built using the domain specific knowledge, or it can be derived from write access map $\mathcal{F}_N^W$ and intra-node data dependence $\mathcal{P}_N$ as follows: Last-writer set, $LS_N = (\text{Range}(\mathcal{P}_N) - \text{Domain}(\mathcal{P}_N))$; and last-writer map,

Table 3. Rules to build read access functions.

Node	*Access relations,* $\mathcal{F}^R$
$N_1[N_2[e_1] \oplus N_3[e_2]]$ $\oplus \in \{+,-\}$	$(\mathcal{M}(N_2) \circ M_1) \cup (\mathcal{M}(N_3) \circ M_2)$ where, M_1 = {D_1[i,j]->M_2[i,j]:}; M_2 = {D_1[i,j]->M_3[i,j]:}.
$N_1[N_2[e_1] * N_3[e_2]]$	$(\mathcal{M}(N_2) \circ M_1) \cup$ $(\mathcal{M}(N_3) \circ M_2) \cup M_3$ where, M_1 = {D_1[i,j,k]->M_2[i,k]:}; M_2 = {D_1[i,j,k]->M_3[k,j]:}; M_3 = {D_1[i,j,k]->M_1[i,j]:}.
$N_1[\mathrm{OP}(N_2[e_1])]_{m,n}$ OP $\in$ {ReLU, Sigmoid, Tanh}	$\mathcal{M}(N_2) \circ M_1$ where, M_1 = {D_1[i,j]->M_2[i,j]:}.

LM_N, is obtained by intersecting the domain of $\mathcal{F}_N^W$ with LS_N, i.e. , $LM_N = \mathcal{F}_N^W \cap (LS_N \times \mathcal{S}_N)$.

Once we have the last-writer map, the complete RAW data-dependence map for the whole DAG can be obtained from the edges as follows: for each edge (N_1, N_2) in the DAG, $\mathcal{P} := \mathcal{P} \cup \mathcal{P}_{N_1} \cup \mathcal{P}_{N_2} \cup \left(\left(\mathcal{F}_{N_2}^R\right)^{-1} \circ LM_{N_1}\right)$, where, $\mathcal{F}_N^R$ is the read access map for node N.

Read-after-read (RAR) dependencies indicate data reuse between points in iteration domain. This information is taken into account while computing schedules, to improve data locality. Given an union of read access relations $\mathcal{F}^R$ of all the operations in the DAG, RAR dependencies,$\mathcal{R}_\mathcal{R}$, can be computed as follows: $\mathcal{R}_\mathcal{R} := (\mathcal{F}^R)^{-1} \circ \mathcal{F}^R$.

3.2 Scheduler

Diesel uses ISL's implementation of Pluto scheduling algorithm [1] to obtain an initial schedule. Pluto's cost function tries to maximize data reuse, and thus generates an outer-parallel/inner-sequential schedule. Multiple loop nests are fused only if fusing them doesn't lead to loss of parallelism. In order to generate an SPMD code for GPUs, it is essential that at least one outermost loop of the computed schedule is parallel. Fig. 7 shows the initial schedule computed for the statement sequence `E=A+B; F=C+D; G=E*F`. We obtain a sequence of two loop nests, where the first loop nest computes the expression `C+D`, while the second loop nest computes the addition `A+B` and multiplication `E*F` in a fused fashion.

Once an initial affine schedule is obtained, we apply a sequence of transformations/optimizations, each of which adds further improvement to the schedule. These optimizations involve *tiling*, *memory promotion*, and *software pipelining*.

```
for (i=0; i<1024; i++)
 for (j=0; j<1024; j++)
   Temp1[i][j] = C[i][j] + D[i][j];

for (i=0; i<1024; i++)
 for (j=0; j<1024; j++)
  for (k=0; k<1024; k++) {
   if (j==0)
    Temp2[i][k] = A[i][k] + B[i][k];
   E[i][j] += Temp1[i][k] * Temp2[k][j];
 }
```

Figure 7. Initial schedule for `E=A+B; F=C+D; G=E*F`.

They are detailed in subsequent subsections. We use matrix-matrix multiplication (GEMM) as a running example (See Fig. 1).

3.2.1 Tiling

Tiling serves the dual purpose of improving data locality, and enabling distribution of workload to thread and threadblocks. Diesel performs a three-level tiling on the initial schedule leading to three sets of tile-loops, and one set of point-loops. Different iterations of the first-set of parallel tile-loops are distributed among different threadblocks. Similarly, second and third set of parallel tile-loops are distributed among different warps and lanes, respectively. Sequential tile-loops, and point-loops are computed sequentially by the threads. Diesel uses its default tilesizes for tiling. The user can override these values by providing a configuration file to Diesel. (Details of the configuration file are not shown for brevity.) In case of GEMM, outer `i` and `j` loops carry no dependencies and are parallel, while the inner `k` loop is sequential.

In order to aid in efficient memory promotion as detailed later, Diesel generally distributes the parallel loops in a block-cyclic fashion. (We omit the details of this optimization, but in summary this is done by tiling the parallel loop that needs to be distributed in block-cyclic fashion once more, and assigning tiles from different cycles to corresponding compute units.) Device function for our running example obtained after tiling and block-cyclic distribution is shown in Fig. 8.

3.2.2 Memory Promotion

Threads within a threadblock can share data through *shared memory*, whose latency is lower than global memory. Unlike CPUs, a GPU's shared memory is software managed. In addition, any data that is reused multiple times within a thread can be reused in registers. The `NVCC` compiler takes care of automatically promoting any private array in a CUDA code to registers, provided that these private arrays are amenable to promotion [4]. In order to utilize these fast memories, Diesel (i) analyzes the tiled schedule to determine the arrays that are reused between threads (for promotion to shared memory), and within threads (for promotion to registers); (ii)

```
int bid_y = blockIdx.y;
int bid_x = blockIdx.x;
int tid = threadIdx.y*blockDim.x+threadIdx.x;
int wid = (tid >> 5); // warp id
int lid = (tid & 31); // lane id
int wid_y = wid / 2;
int wid_x = wid % 2;
int lid_y = lid / 8;
int lid_x = lid % 8;

int idx1 = 128*bid_y + 32*wid_y + 4*lid_y;
int idx2 = 128*bid_x + 64*wid_x + 4*lid_x;
for (k=0; k<64; k++)
 for (kk=0; kk<16; kk++) {
  int z_idx = 16*k+kk;
  for (i=0; i<2; ++i)
   for (j=0; j<2; ++j)
    for (ii=0; ii<4; ii++)
     for (jj=0; jj<4; jj++) {
      int y_idx = idx1 + 16*i + ii;
      int x_idx = idx2 + 32*j + jj;
      C[y_idx][x_idx] += A[y_idx][z_idx] *
                         B[z_idx][x_idx];
    }
 }
```

Figure 8. Code segment showing the device function for GEMM after tiling and loop-distribution for the problem size $1024 \times 1024 \times 1024$. Tiling is applied thrice, with tilesizes $(128, 128, 16)^T$, $(32, 64, 16)^T$, and $(8, 8, 1)^T$. Gridsize is <<< $8, 8, 1$ >>>, and threadblock size is <<< $16, 16, 1$ >>>. Each threadblock collectively computes a 128×128 sub-array of C. 32 Lanes in a warp are arranged as 4×8 two-dimensional block. Lanes in a single warp collectively compute a 32×64 sub-array. A single thread computes two blocks of 4×4 sub-arrays, separated by strides 16 and 32 along y and x dimensions, respectively.

determines sizes of promoted arrays; (iii) constructs schedules to copy data from slow to fast memory and inserts them at appropriate locations in the original schedule; and finally (iv) updates all reused global array accesses into accesses to promoted arrays.

In order to determine whether an array access $\mathcal{F}$ has reuse between threads under a schedule ϕ, Diesel check if the parallel-schedule-to-array map $\mathcal{F}_\phi := (\mathcal{F} \circ \phi^{-1}_{|par})$ is non-injective, where $\phi_{|par}$ is ϕ restricted to parallel dimensions, with sequential dimensions projected out. If $\mathcal{F}_\phi$ is non-injective, it implies that the same array element is accessed more than once along parallel dimensions, thus implying reuse between threads. Note that since the parallel dimensions do not carry any data dependencies, only read-only (i.e. input) arrays have reuses along parallel dimensions. Further, in the actual hardware since all the operations are performed in registers, all input operands are loaded to registers prior to the computation, and all computed values are temporarily placed in registers before they are written out. Hence, Diesel marks all the arrays for promotion to private memory. This also allows Diesel to explicitly do vector loads/stores to/from registers even if the array has no intra-thread reuse.

Once the arrays are marked for promotion, Diesel analyzes the constraints of those accesses to determine bounds of promoted array sizes. At this point, the loop depth at which the data is reused is also determined so that the load/store operations can be inserted at appropriate locations in the schedule. Once the reuse depths are determined, Diesel updates the original schedule with copy operations. Global-to-shared memory copy schedules are marked to be distributed between threads in a block-cyclic fashion, so that the global memory accesses are coalesed, and shared memory bank conflicts are eliminated, while also allowing the loads to be vectorized. Finally, Diesel replaces global accesses functions with accesses to promoted arrays.

During memory promotion, Diesel performs checks on array accesses using simple heuristics to determine if there exist any potential bank conflicts in the promoted arrays. For example, in our GEMM code (Fig. 8), neighboring threads access elements of array `A` along the slow varying dimension, which once promoted would lead to shared memory bank conflicts. To prevent this, the data is transposed while it is being copied to shared memory. In addition, arrays are padded as needed to eliminate any bank conflicts that might arise due to this transposition during copy from global to shared memory. Fig. 9 illustrates these optimizations.

3.2.3 Software Pipelining

As seen in Fig. 9, the device functions generated for the domain follow the general pattern: `load_to_shared(); sync(); compute(); sync(); repeat`. It is possible to overlap memory and compute operations to hide the data transfer latency. Diesel achieves this by allocating an additional buffer for the data, and prefetching the data needed for the next iteration asynchronously, thus overlapping the data transfer with computation. This leads to the pattern `async_prefetch(); compute(); sync(); repeat`, where there is no `sync()` between prefetch and compute. While this reduces latency, it also increases the shared memory requirement, thus reducing the device occupancy which might be detrimental to the performance. In order to overcome this, Diesel performs a partial overlap as follows: load from global to shared is split into two phases – read from global memory to registers, and write from registers to shared memory. As reads from global memory have higher latency, phase one alone is overlapped with computation, eliminating the requirement for additional shared memory. This provides us the pattern: `async_prefetch_to_reg(); compute(); sync(); write_to_shared(); sync(); repeat`. Effect of applying this transformation on GEMM is shown in Fig. 10.

```
__shared__  float As[16][128+2], Bs[16][128];
float Ap[1][8], Bp[1][8], Cp[8][8] = {0};

for (int k=0; k<64; k++) {
 // GlbToShr(src, dest, dim1, dim2)
 GlbToShr(&(A[128*bid_y][16*k]),
          As, 16, 128);
 GlbToShr(&(B[16*k][128*bid_x]),
          Bs, 16, 128);
 __syncthreads();

 for (int kk=0; kk<16; kk++) {
  // ShrToReg(src, dest, dim1, dim2,
  //          cycles1, cycles2,
  //          stride1, stride2)
  ShrToReg(&(As[kk][32*wid_y+4*lid_y]),
           Ap, 1, 4, 1, 2, 1, 16);
  ShrToReg(&(Bs[kk][64*wid_x+4*lid_x]),
           Bp, 1, 4, 1, 2, 1, 32);

  //Compute
  for (int i=0; i<8; i++)
   for (int j=0; j<8; j++)
    Cp[i][j] += Ap[0][i] * Bp[0][j];
 }
 __syncthreads();
}

// RegToGlb(src, dest, dim1, dim2,
//          cycles1, cycles2,
//          stride1, stride2)
RegToGlb(Cp, &(C[128*bid_y][128*bid_x]),
         4, 4, 2, 2, 16, 32);
```

Figure 9. Code segment of GEMM device function after memory promotion. To prevent bank conflicts, array A is transposed on-the-fly while it is being copied from global to shared memory, and the shared array As has been padded.

3.3 Code Generation

The final schedule, after application of several transformations is represented in the form of a *schedule tree* [2]. A device function is created corresponding to each high-level loop nest. Grid sizes and threadblock sizes are derived by analyzing the bounds of parallel schedule dimensions. During code generation, the loops that are marked for distribution are replaced with appropriate CUDA variables (such as `blockIdx.y`, etc), and the remaining loops and statements are printed within the body of the device function.

4 Experimental Results

CuBLAS is the widely used standard library to perform linear algebra computations on GPUs. In this section, we compare the performance of Diesel generated code against CuBLAS. We used CuBLAS 9.0 and CuDNN 7.1 as baselines, and NVCC

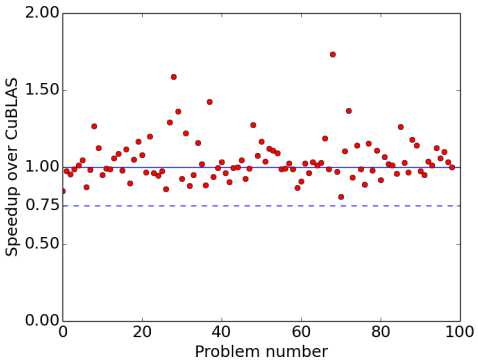
```
__shared__  float As[16][128+2], Bs[16][128];
float Ap[1][8], Bp[1][8], Cp[8][8] = {0};
float Abuf[8], Bbuf[8];

GlbToReg(&(A[128*bid_y][0]), Abuf, 16, 128);
GlbToReg(&(B[0][128*bid_x]), Bbuf, 16, 128);
RegToShr(Abuf, As); RegToShr(Bbuf, Bs);
__syncthreads();
for (int k=0; k<64; k++) {
 GlbToReg(&(A[128*bid_y][16*(k+1)]),
          Abuf, 16, 128);
 GlbToReg(&(B[16*(k+1)][128*bid_x]),
          Bbuf, 16, 128);

 for (int kk=0; kk<16; kk++) {
  ShrToReg(&(As[kk][32*wid_y+4*lid_y]),
           Ap, 1, 4, 1, 2, 1, 16);
  ShrToReg(&(Bs[kk][64*wid_x+4*lid_x]),
           Bp, 1, 4, 1, 2, 1, 32);

  for (int i=0; i<8; i++)
   for (int j=0; j<8; j++)
    Cp[i][j] += Ap[0][i] * Bp[0][j];
 }
 __syncthreads();
 RegToShr(Abuf, As); RegToShr(Bbuf, Bs);
 __syncthreads();
}

RegToGlb(Cp, &(C[128*bid_y][128*bid_x]),
         4, 4, 2, 2, 16, 32);
```

Figure 10. Code segment of GEMM device function after pipelining.

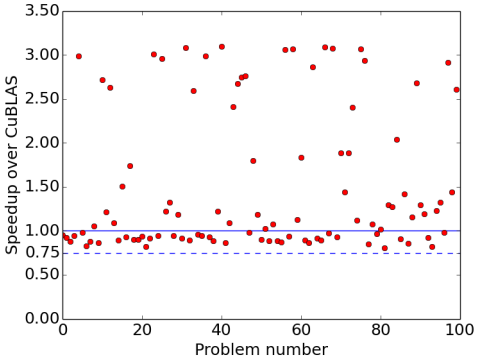
9.0 to compile Diesel generated CUDA kernels for our experiments. Experiments were performed on Pascal (sm_60) and Volta (sm_70) GPUs. We chose 100 randomly generated problem sizes within the range [32, 16384] (that provide a mix of square and rectangular matrices) as datasets for our experiments. The problem sizes were rounded to their nearest multiple of 32 when they were < 128, and to their nearest multiple of 128 otherwise [3]. The running times of the kernels were averaged over multiple runs to minimize any variations due to disturbances.

[3] CuBLAS chooses different kernels at runtime depending on the problem size. Each of these kernels is optimized for a specific tilesize. At runtime, if the problem size is not a multiple of those tilesizes, it either pads the matrices with zeros during load to shared memory, or splits the computation into two phases: the highly optimized main kernel computes part of the problem that is a multiple of tilesize, and the remaining parts are handled by a residual kernel. Since the amount of work performed by the residual kernel is not significant, these kernels are not generally highly optimized. We take a similar approach with Diesel, where Diesel generates code optimized for a set of tilesizes, and uses a simpler kernel to execute the residual part. Hence, we restricted the problem sizes to multiples of those tilesizes to directly compare the performances of main kernels.

SGEMM CuBLAS's single-precision matrix-matrix multiplication kernel (`cublasSgemm`) is a highly optimized function hand-tuned to provide close-to-peak performance on Nvidia's GPUs. Fig. 11(a) shows the speedup of Diesel generated version against CuBLAS on a Pascal GPU. As shown in the figure, Diesel generated code achieves at least a speedup of 0.8, with the highest speedup being 1.73. Out of 100 problem sizes, 52 of them achieve a speedup of ≥ 1. We see a similar trend on Volta (Fig. 11(b)), where Diesel achieves upto 3.1× speedup, with 56 problem sizes achieving speedup ≥ 1.



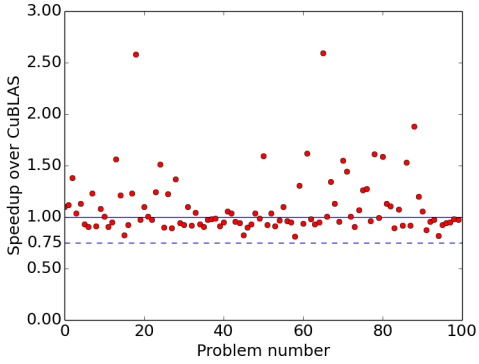
(a) SGEMM on Pascal



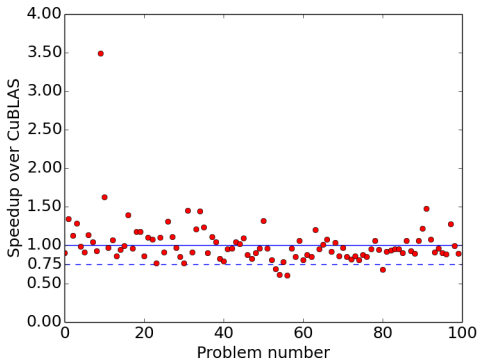
(b) SGEMM on Volta

Figure 11. Speedup of Diesel generated SGEMM computation over CuBLAS on Pascal (sm_60) and Volta (sm_70) GPUs.

DGEMM Fig. 12(a) shows the speedup of Diesel generated code for double-precision matrix-matrix multiplication computation against CuBLAS on a Pascal GPU. We see similar trend as that of SGEMM, where Diesel generated code achieves at least a speedup of 0.8. The highest achieved speedup is 2.59, and 47 out of 100 problem sizes have a speedup of ≥ 1. DGEMM performance on Volta is shown in Fig. 12(b), where Diesel achieves a highest speedup of 3.5×. Ninety-six problems achieve a performance of > 75, and 38 problems achieve ≥ 1 speedup.



(a) DGEMM on Pascal



(b) DGEMM on Volta

Figure 12. Speedup of Diesel generated DGEMM computation over CuBLAS on Pascal (sm_60) and Volta (sm_70) GPUs.

GEMM + bias + ReLU It is common in neural network computation to perform matrix-matrix multiplication, followed by addition of bias and application of activation functions. Typically, various pacakges use CuBLAS and CuDNN to perform these computation one-by-one while transfering intermediate matrices/tensors to global memory between different function calls. We used Diesel to generate a fused code that performs these three operations within a single kernel, by reusing intermediate values directly in registers. As baseline, we wrote the same computation as a sequence of `cublasSgemm`, `cudnnAddTensor`, `cudnnActivationForward` calls. Fig. 13 shows the speedup obtained by Diesel generated version against the baseline. As it can be seen, Diesel consistently performs on par or better than baseline. More than 90% of the time, Diesel generated code runs faster than the baseline version.

5 Related Work

CuBLAS [3] and CuDNN [5] have been the mainstream libraries for linear algebra and neural network computations, respectively, on GPUs. These kernels are handwritten directly in GPU's assembly language (SASS). This involves

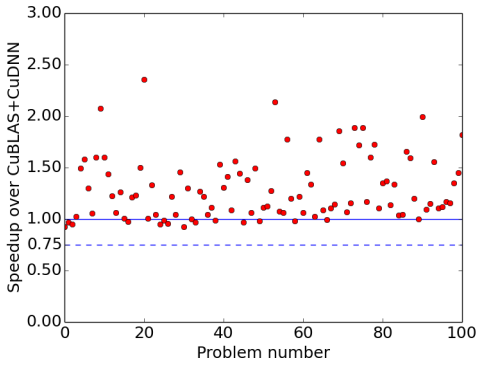


Figure 13. Speedup of Diesel generated neural net code over CuBLAS+CuDNN on Pascal (sm_60) GPU.

significant amount of manual effort. In addition, these kernels need to be rewritten for each GPU architecture, leading to a continuous effort. In contrast, Diesel provides a fully automatic solution, where different optimizations are encoded and applied in an abstract form with the help of well developed polyhedral machineries. In addition, Diesel allows the code to be tuned using an auto-tuner thus eliminating the manual effort. OpenAI [7] takes a similar approach, where GEMM kernels are written in SASS, and hand tuned for individual architectures.

PPCG [10] is a polyhedral compiler for CUDA programs that transforms a regular C program into a CUDA code. Similar to Diesel, PPCG internally uses polyhedral transformations to optimize the code. However, PPCG has been written to handle general purpose affine computations. While Diesel's optimization passes have been tuned to generate highly efficient codes for specific class of computations from linear algebra domain, thus allowing Diesel to achieve better performance for such problems, PPCG performs common optimizations for a wider domain while ignoring any domain specific optimizations. Further, PPCG expects equivalent C programs as inputs, whereas Diesel allows users to express operations with simple expressions and uses domain specific knowledge to convert them to CUDA programs.

In the similar vein to Diesel, CUTLASS [6] strives to ease the manual effort of having to write optimized kernels in SASS. CUTLASS is a templated C++ library that decomposes the problems from linear algebra domain into fundamental components, and exposes them to the users as C++ template classes, allowing programmers to easily customize and specialize them within their own CUDA kernels. While this significantly reduces user's effort in writing efficient kernels, the library itself needs to be hand optimized for different architectures.

Facebook's Tensor Comprehension (TC) [8] is a mathematical language to express machine learning (ML) computations. TC internally uses polyhedral libraries to optimize the code. TC mainly focuses on achieving good performance for the network as a whole, as opposed to optimizing each individual operations.

6 Conclusion

We presented a domain specific language compiler, Diesel, to generate efficient CUDA kernels for linear algebra operations. Diesel accepts basic linear algebra and neural net expressions in an intuitive form, and generates a corresponding efficient CUDA code. Diesel internally uses machineries from polyhedral techniques to perform optimizations in an abstract form, thus eliminating the need to manually optimize each kernel. Further, several domain specific optimizations are performed to achieve high performance. The experimental results show that the performance of the kernels generated by Diesel for individual operations are comparable to the performance provided by standard libraries. Further, when provided with a sequence of operations, Diesel generates an efficient fused kernel thus eliminating redundant data transfers, and achieving better performance.

Acknowledgments

We would like to thank Duane Merrill and Andrew Kerr of NVIDIA for their useful comments and suggestions.

References

[1] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Program Optimization System. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[2] Tobias Grosser, Sven Verdoolaege, and Albert Cohen. 2015. Polyhedral AST Generation Is More Than Scanning Polyhedra. *ACM Trans. Program. Lang. Syst.* 37, 4, Article 12 (July 2015), 50 pages. https://doi.org/10.1145/2743016

[3] NVIDIA. 2018. CuBLAS: Dense Linear Algebra on GPUs. https://developer.nvidia.com/cublas

[4] NVIDIA. 2018. CUDA programming guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#device-memory-accesses

[5] NVIDIA. 2018. CuDNN: GPU Accelerated Deep Learning. https://developer.nvidia.com/cudnn

[6] NVIDIA. 2018. CUTLASS: Fast Linear Algebra in CUDA C++. https://github.com/NVIDIA/cutlass

[7] OpenAI. 2018. OpenAI: Open single and half precision GEMM implementations. https://github.com/openai/openai-gemm

[8] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *ArXiv e-prints* (Feb. 2018). arXiv:cs.PL/1802.04730

[9] Sven Verdoolaege. 2010. isl: An Integer Set Library for the Polyhedral Model. In *Mathematical Software – ICMS 2010*, Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 299–302.

[10] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral Parallel Code Generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4, Article 54 (Jan. 2013), 23 pages. https://doi.org/10.1145/2400682.2400713