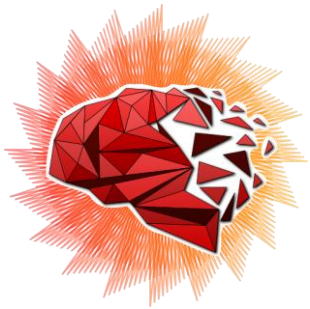
Orchestrating a brighter world NEC



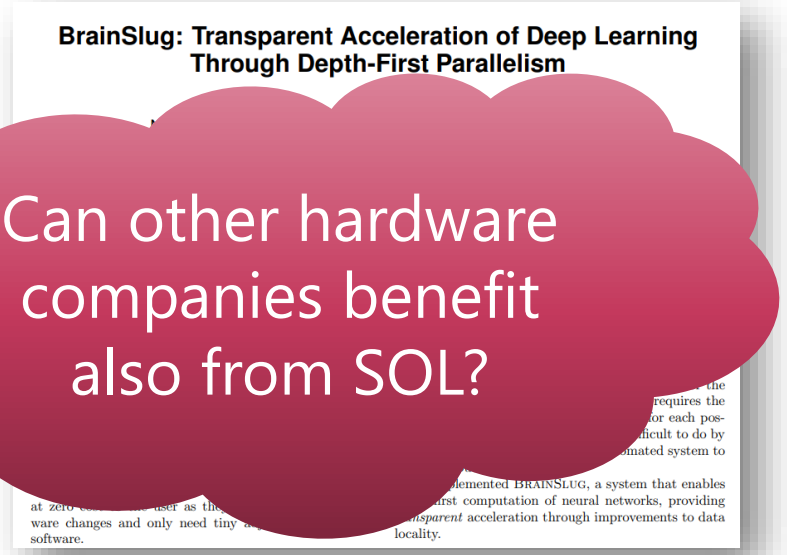
Facilitate high-performance hardware integration into AI Frameworks with the NEC SOL AI compiler

Nicolas Weber <nicolas.weber@neclab.eu>
NEC Laboratories Europe

IWAPT'25 @ IPDPS Milano

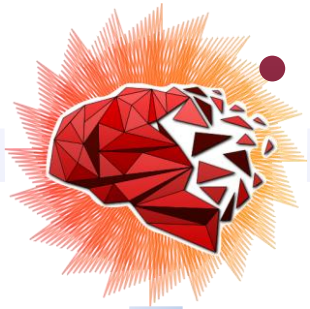
„NEC SOL? Never heard of her."

- NEC SOL AI compiler
  - Started as RnD project end of 2017
  - Aimed for improving NEC AI development and products


BrainSlug: Transparent Acceleration of Deep Learning Through Depth-First Parallelism

Can other hardware companies benefit also from SOL?

https://arxiv.org/abs/1804.08378



NEC

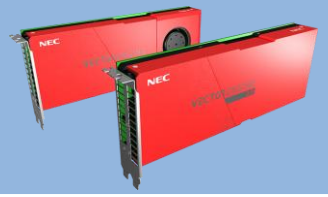
EU Projects

BRAINE

DESIRE6G

NEC

HPC Customers



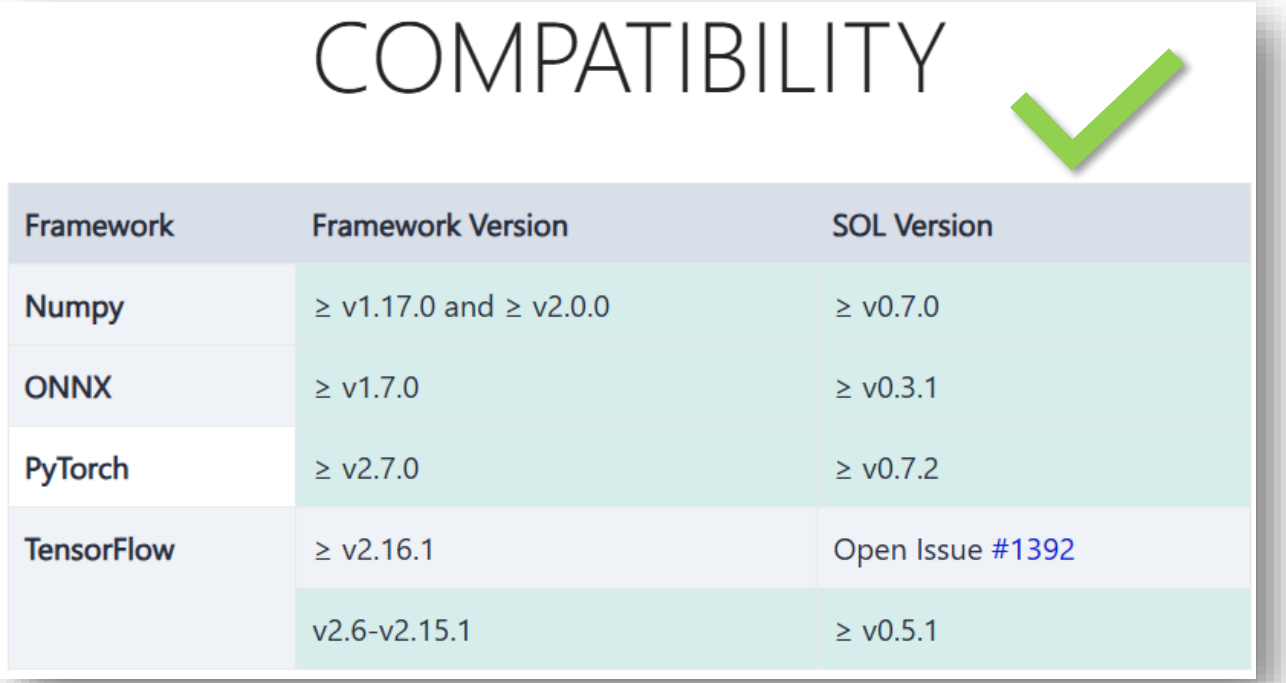
NEC SOL as compiler for AI hardware vendors?

- How do you convince hardware vendors to invest into a proprietary solution and not just take open-source software, e.g., TVM, OpenVino, OpenXLA, …?

- Vendors need to compete with 15+ years of software development at NVIDIA
  - Framework support: **PyTorch**, **TensorFlow** (**+Keras**), **Numpy**, **ONNX**, JAX, …

```
import sol
model = sol.optimize(model)
sol.device.set('ve', 0)
```

  - Execution Modes: **Inference**/**Deployment** and **Training**
  - Need to be cutting edge
  - Great performance

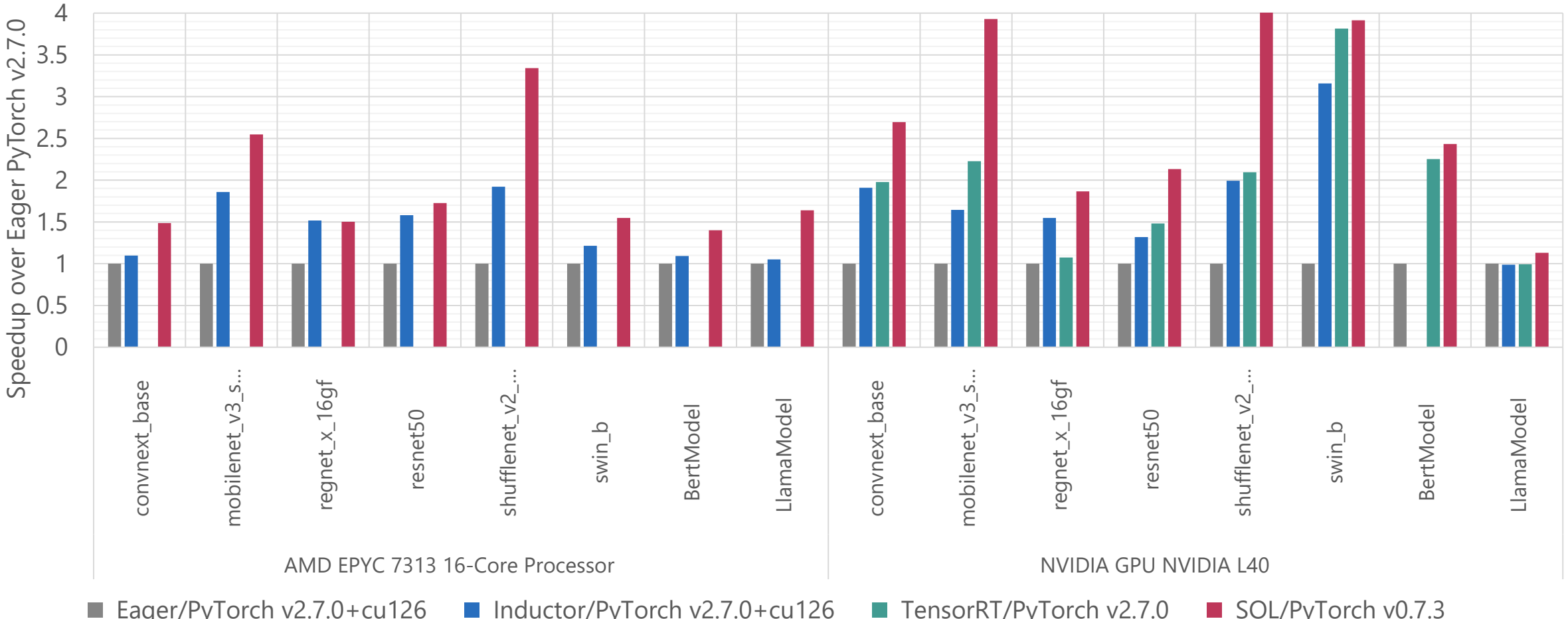


Framework	Framework Version	SOL Version
Numpy	≥ v1.17.0 and ≥ v2.0.0	≥ v0.7.0
ONNX	≥ v1.7.0	≥ v0.3.1
PyTorch	≥ v2.7.0	≥ v0.7.2
TensorFlow	≥ v2.16.1	Open Issue #1392
	v2.6-v2.15.1	≥ v0.5.1

How fast is SOL?

Benchmark Config:

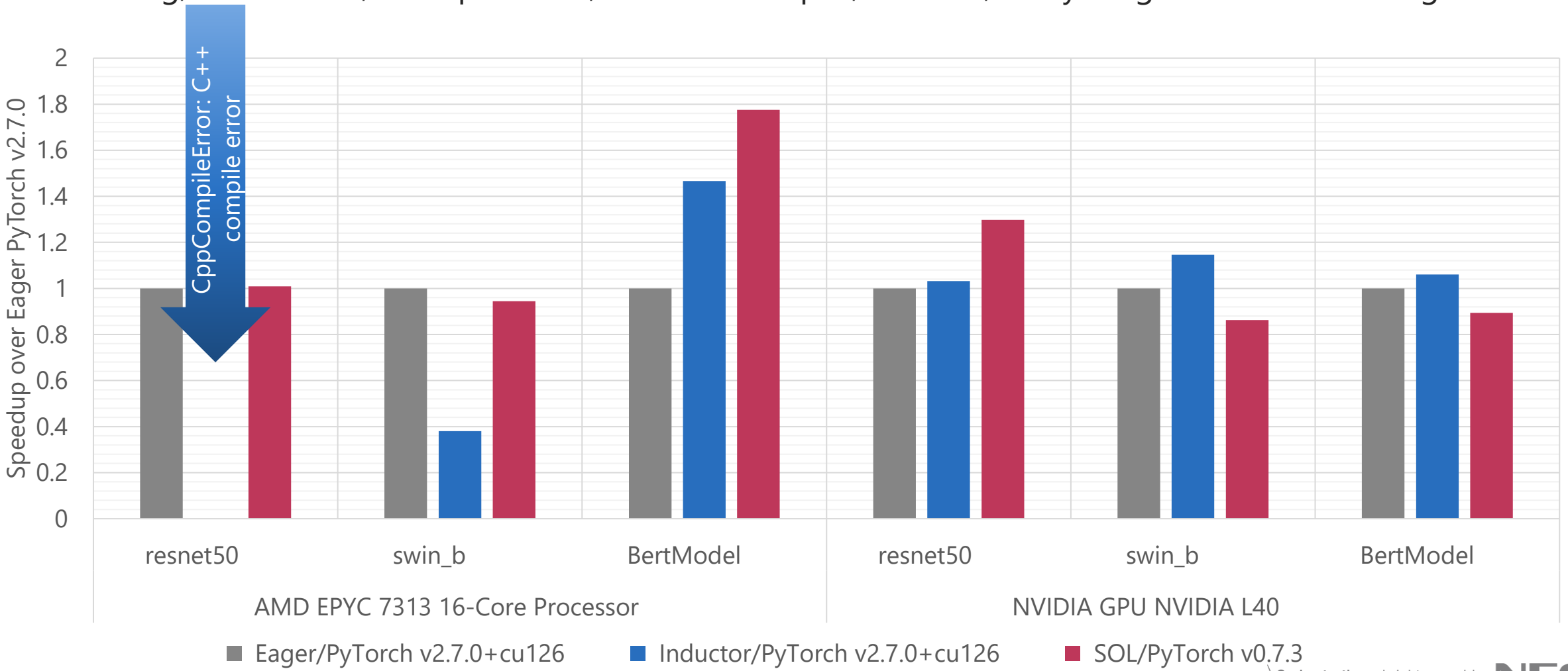
- Inference, BatchSize 1, FP32 precision, no CUDA Graphs, no TF32, everything else default settings.



How fast is SOL?

Benchmark Config:

- Training, BatchSize 8, FP32 precision, no CUDA Graphs, no TF32, everything else default settings



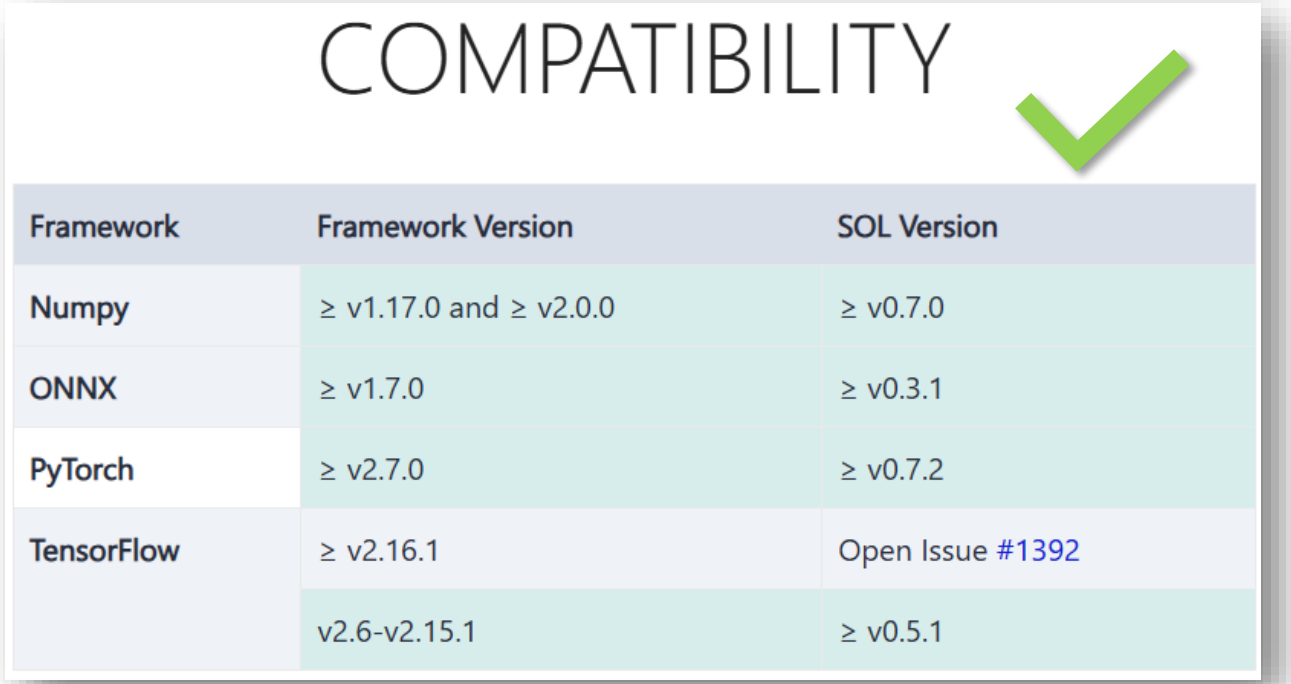
NEC SOL as compiler for AI hardware vendors?

- How do you convince hardware vendors to invest into a proprietary solution and not just take open-source software, e.g., TVM, OpenVino, OpenXLA, …?

- Vendors need to compete with 15+ years of software development at NVIDIA
  - Framework support: **PyTorch**, **TensorFlow** (**+Keras**), **Numpy**, **ONNX**, JAX, …

```
import sol
model = sol.optimize(model)
sol.device.set('ve', 0)
```

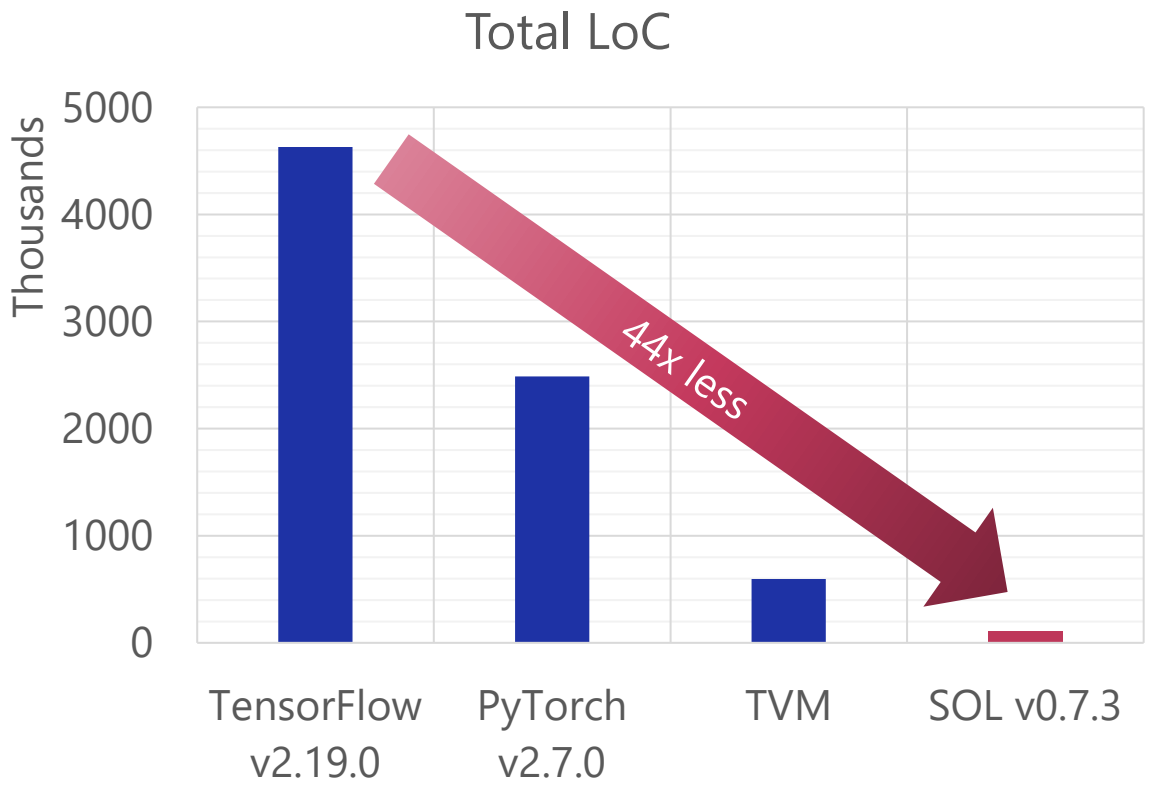
  - Execution Modes: **Inference**/**Deployment** and **Training**
  - Need to be cutting edge
  - Great performance
- Service and Maintainability
  - OSS has no obligation to fix bugs or help developers

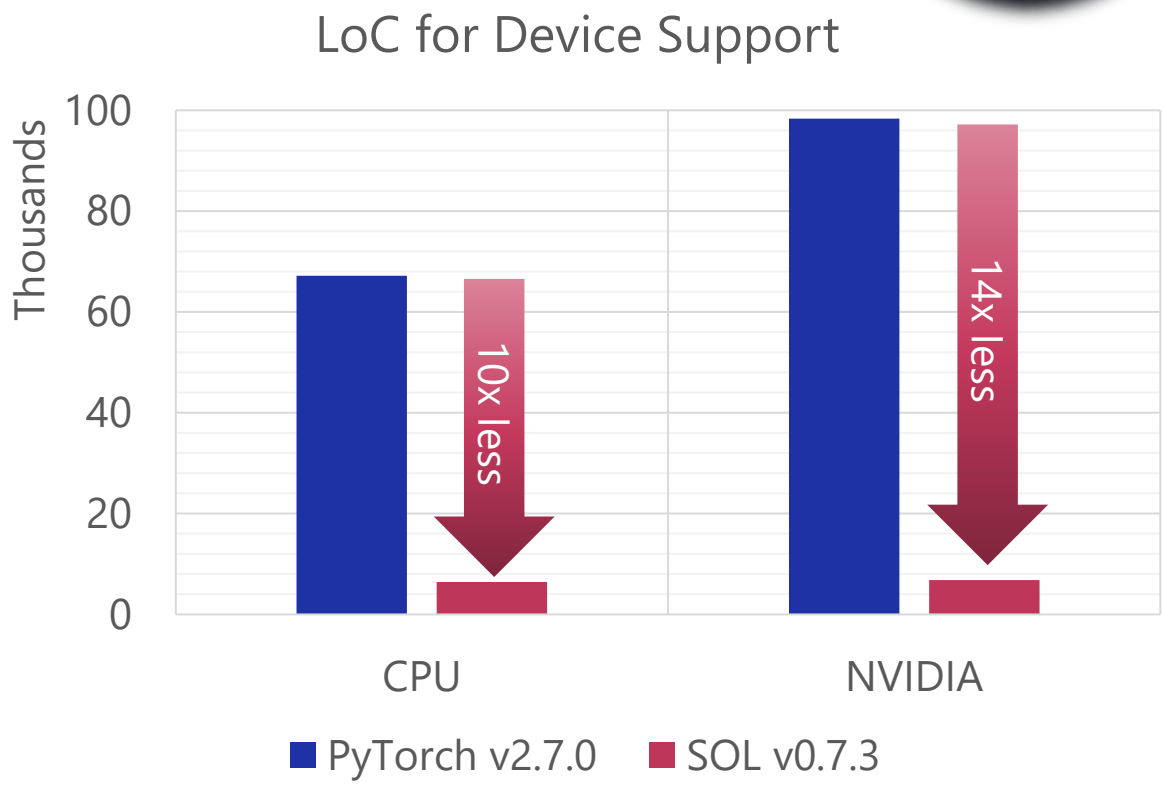


Framework	Framework Version	SOL Version
Numpy	≥ v1.17.0 and ≥ v2.0.0	≥ v0.7.0
ONNX	≥ v1.7.0	≥ v0.3.1
PyTorch	≥ v2.7.0	≥ v0.7.2
TensorFlow	≥ v2.16.1	Open Issue #1392
	v2.6-v2.15.1	≥ v0.5.1

SOL is designed for easy maintenance

- SOL dev team is **very small**, most of the time it was just this poor guy





*LoC counted with cloc

SOL Infrastructure

- Convert NN into SOL High Level IR
- integrate into AI framework runtime system

High level optimizations:
- Static evaluation
- Dead code elimination
- Deduplication
- …

Device specific optimizations:
- Auto upcasting (e.g. SX-Aurora supports only 32/64Bit dtypes for vectorization)

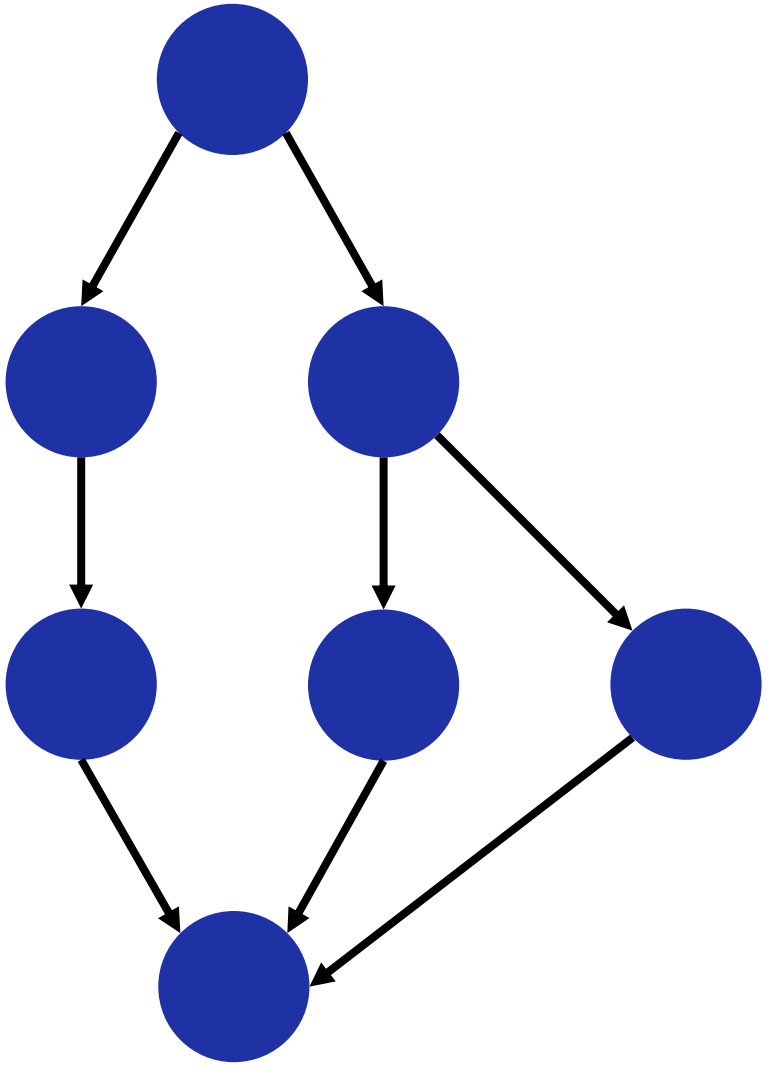
Frameworks: PyTor | Numpy+JAX | API

Core: API | Runtime

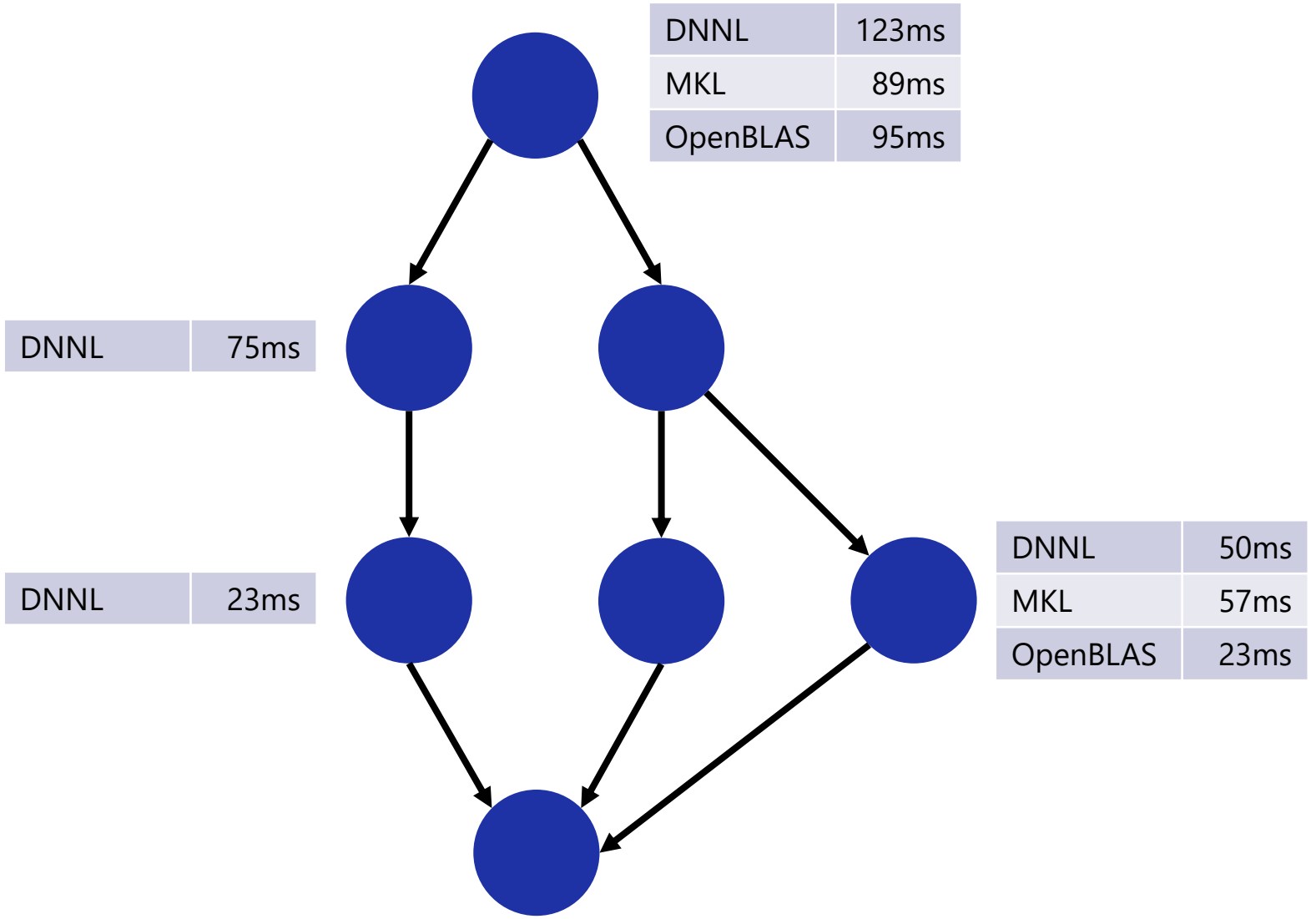
Devices: X86 CPUs | NVIDIA GPUs | NEC SX-Aurora | API

Backends:
- ISPC
- API
- Opt. Libraries: GCC, MKL, CUDA, CUDNN, VEASL, DNNL, AOCLBLAS, CUBLAS, CUBLASLT, NCC, CUDA, CUFFT, CURAND, VEBLAS, VEDNN, API
- DFP CodeGen: ISPC, NCC, CUDA, API

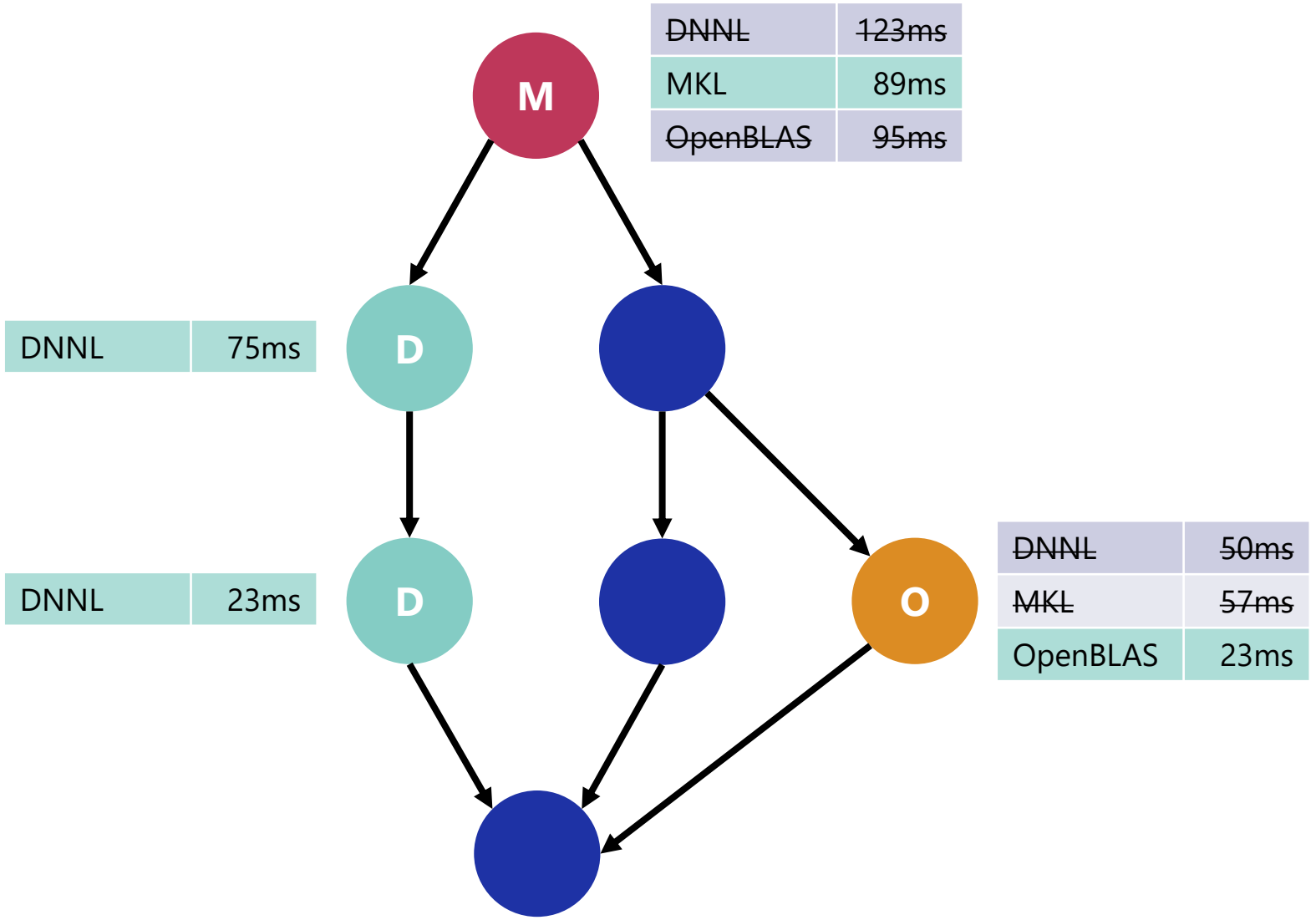
Auto-Tuning



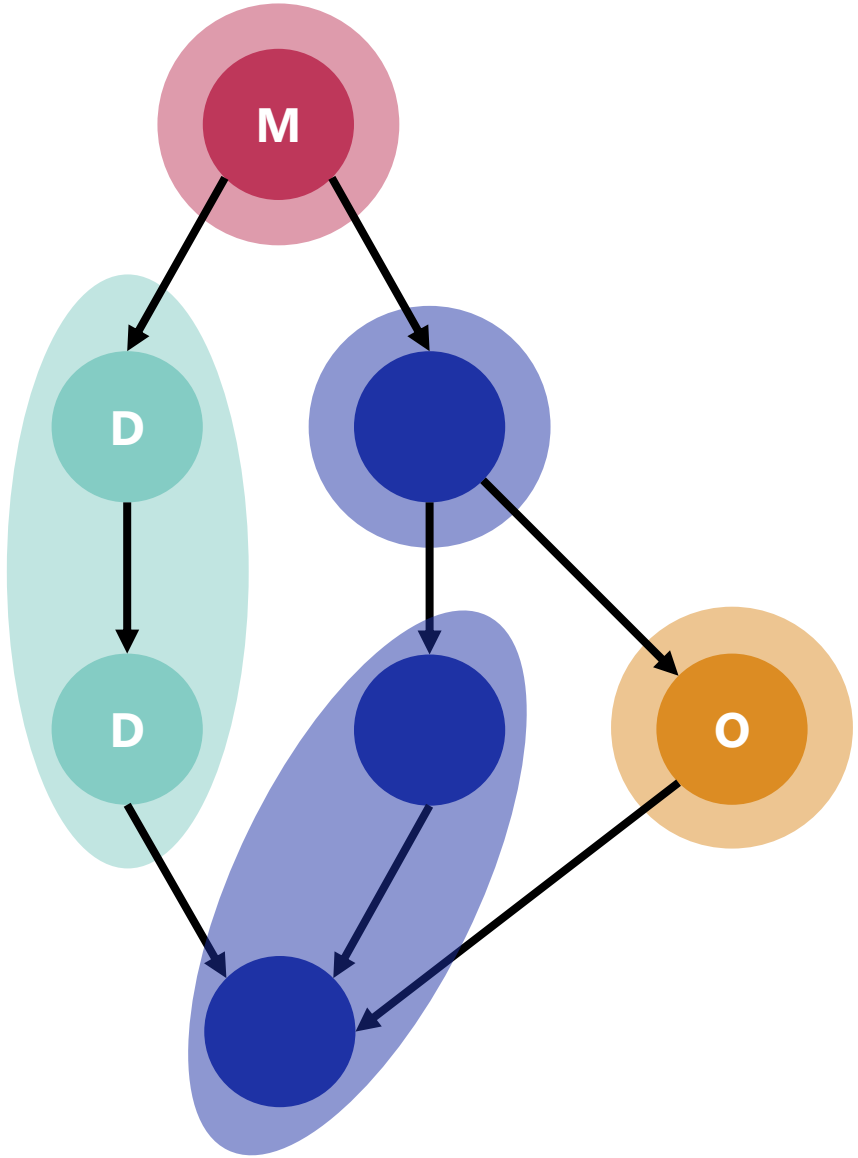
Auto-Tuning



Auto-Tuning

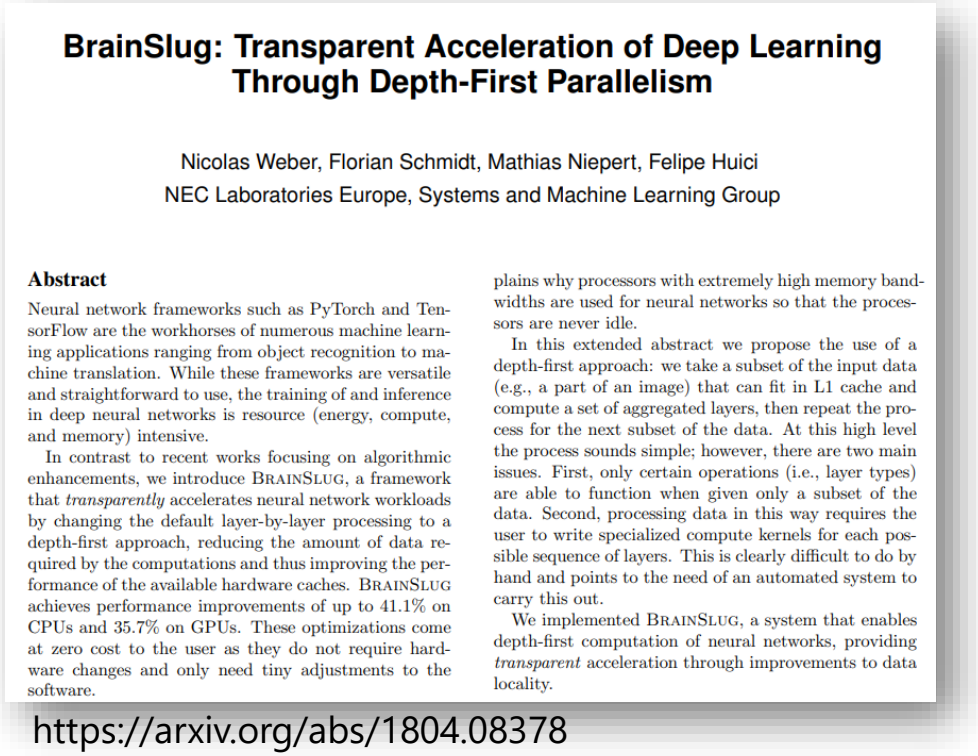


Auto-Tuning



DFP: Depth-First Parallelism

- DFP is our "work-horse" for the **~90% of layers**, that take up **~10% of execution time** (if done correctly!)



BrainSlug: Transparent Acceleration of Deep Learning Through Depth-First Parallelism

Nicolas Weber, Florian Schmidt, Mathias Niepert, Felipe Huici
NEC Laboratories Europe, Systems and Machine Learning Group

Abstract

Neural network frameworks such as PyTorch and TensorFlow are the workhorses of numerous machine learning applications ranging from object recognition to machine translation. While these frameworks are versatile and straightforward to use, the training of and inference in deep neural networks is resource (energy, compute, and memory) intensive.

In contrast to recent works focusing on algorithmic enhancements, we introduce BrainSlug, a framework that *transparently* accelerates neural network workloads by changing the default layer-by-layer processing to a depth-first approach, reducing the amount of data required by the computations and thus improving the performance of the available hardware caches. BrainSlug achieves performance improvements of up to 41.1% on CPUs and 35.7% on GPUs. These optimizations come at zero cost to the user as they do not require hardware changes and only need tiny adjustments to the software.

plains why processors with extremely high memory bandwidths are used for neural networks so that the processors are never idle.

In this extended abstract we propose the use of a depth-first approach: we take a subset of the input data (e.g., a part of an image) that can fit in L1 cache and compute a set of aggregated layers, then repeat the process for the next subset of the data. At this high level the process sounds simple; however, there are two main issues. First, only certain operations (i.e., layer types) are able to function when given only a subset of the data. Second, processing data in this way requires the user to write specialized compute kernels for each possible sequence of layers. This is clearly difficult to do by hand and points to the need of an automated system to carry this out.

We implemented BrainSlug, a system that enables depth-first computation of neural networks, providing *transparent* acceleration through improvements to data locality.

https://arxiv.org/abs/1804.08378

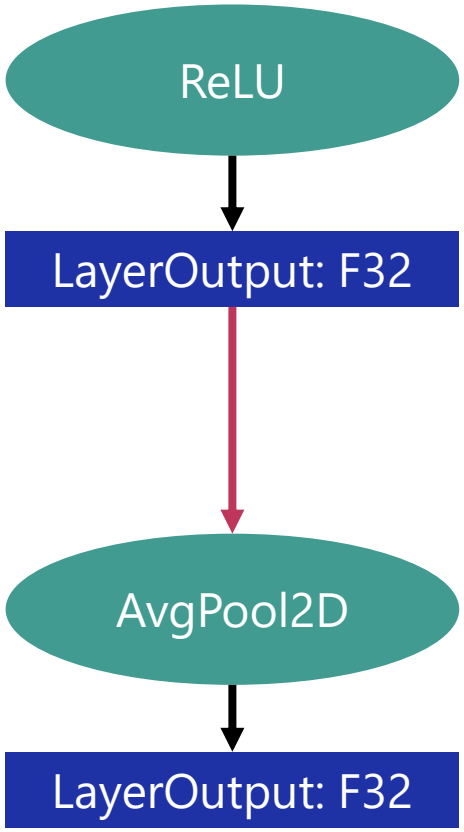
- For more literature, search for "nicolas weber nec laboratories europe" on patents.google.com

DFP: Simple example

```
def forward(self, a):
    b = torch.relu(a)
    c = torch.nn.functional.avg_pool2d(b, 2, stride=(2,1))
    d = torch.exp(c)
    return d
```

DFP: Init Loop Primitives

Dim	Size	Stride
0	1	150528
1	3	50176
2	224	224
3	224	1

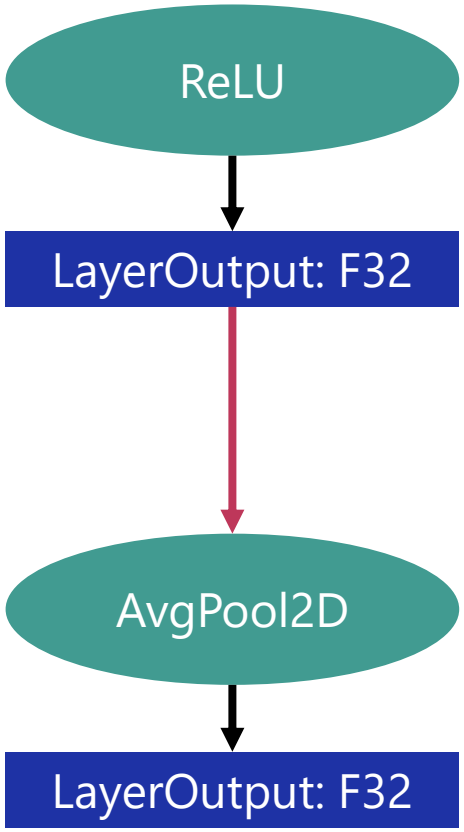


Dim	Size	Stride
0	1	150528
1	3	50176
2	224	224
3	224	1

Dim	Size	Stride
0	1	150528
1	3	50176
2	112	224
3	112	1

DFP: Init Loop Primitives

Dim	Key	Size	Stride
0	None3	1	150528
1	None2	3	50176
2	None1	224	224
3	None0	224	1

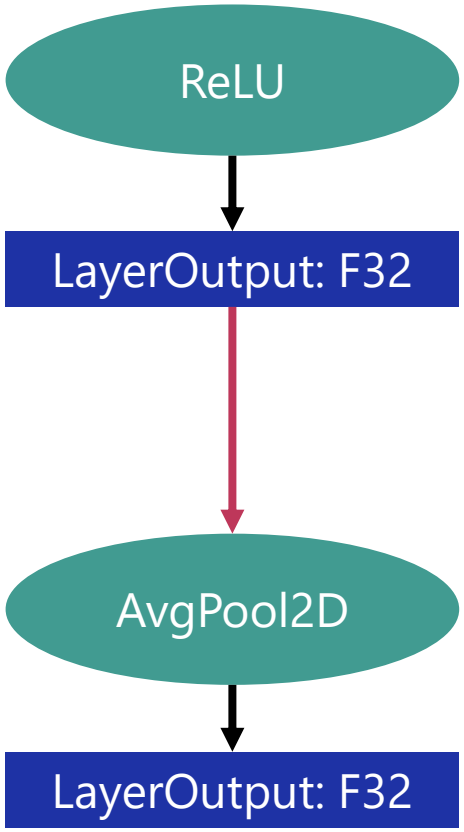


Dim	Key	Size	Stride
0	Batch	1	150528
1	Channels	3	50176
2	InPixelY	224	224
3	InPixelX	224	1

Dim	Key	Size	Stride
0	Batch	1	150528
1	Channels	3	50176
2	OutPixelY	112	224
3	OutPixelX	112	1

DFP: Init Loop Primitives

Dim	Key	Size	Stride	Primitive
0	None3	1	150528	Linear
1	None2	3	50176	Linear
2	None1	224	224	Linear
3	None0	224	1	Linear

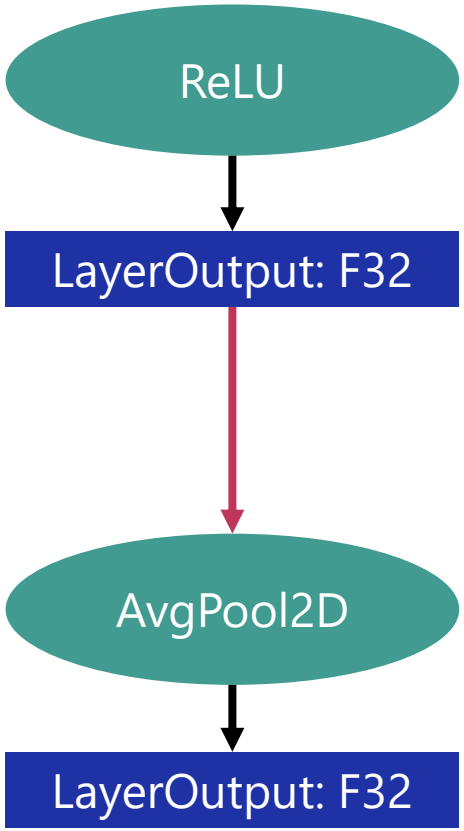


Dim	Key	Size	Stride	Primitive
0	Batch	1	150528	Linear
1	Channels	3	50176	Linear
2	InPixelY	224	224	Pooling
3	InPixelX	224	1	Pooling

Dim	Key	Size	Stride	Primitive
0	Batch	1	150528	Linear
1	Channels	3	50176	Linear
2	OutPixelY	112	224	Linear
3	OutPixelX	112	1	Linear

DFP: Init Loops

Dim	Key	Size	Stride	Primitive	Loop
0	None3	1	150528	Linear	L4
1	None2	3	50176	Linear	L5
2	None1	224	224	Linear	L6
3	None0	224	1	Linear	L7

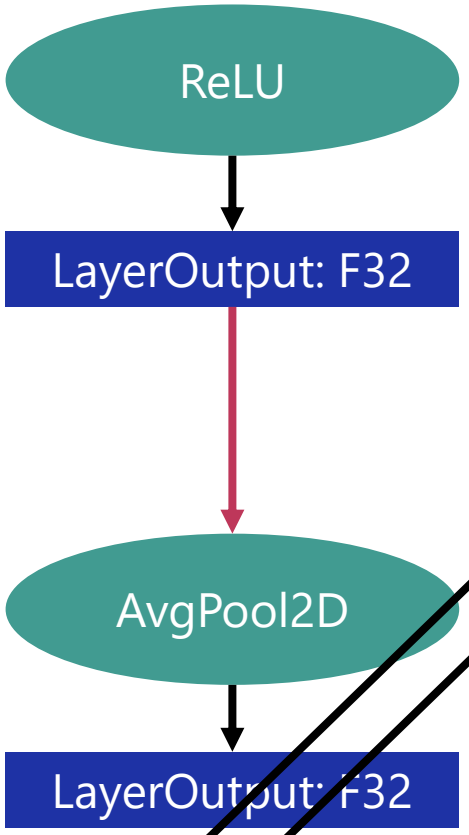


Dim	Key	Size	Stride	Primitive
0	Batch	1	150528	Linear
1	Channels	3	50176	Linear
2	InPixelY	224	224	Pooling
3	InPixelX	224	1	Pooling

Dim	Key	Size	Stride	Primitive	Loop
0	Batch	1	150528	Linear	L0
1	Channels	3	50176	Linear	L1
2	OutPixelY	112	224	Linear	L2
3	OutPixelX	112	1	Linear	L3

DFP: Init Loops

Dim	Key	Size	Stride	Primitive	Loop
0	None3	1	150528	Linear	L4
1	None2	3	50176	Linear	L5
2	None1	224	224	Linear	L6
3	None0	224	1	Linear	L7

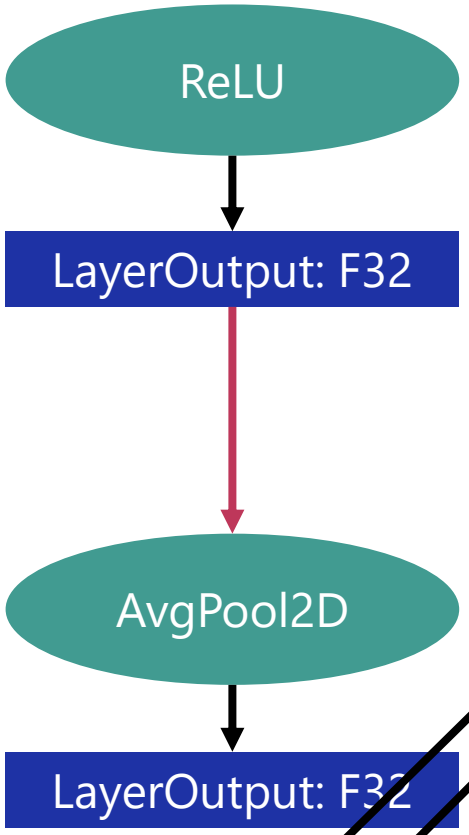


Dim	Key	Size	Stride	Primitive	Loop
0	Batch	1	150528	Linear	L0
1	Channels	3	50176	Linear	L1
2	InPixelY	224	224	Pooling	
3	InPixelX	224	1	Pooling	

Dim	Key	Size	Stride	Primitive	Loop
0	Batch	1	150528	Linear	L0
1	Channels	3	50176	Linear	L1
2	OutPixelY	112	224	Linear	L2
3	OutPixelX	112	1	Linear	L3

DFP: Init Loops

Dim	Key	Size	Stride	Primitive	Loop
0	None3	1	150528	Linear	L4
1	None2	3	50176	Linear	L5
2	None1	224	224	Linear	L6
3	None0	224	1	Linear	L7

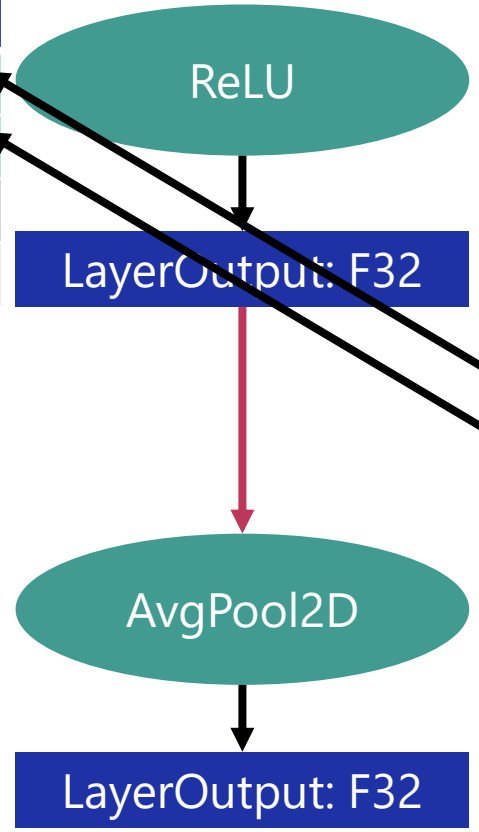


Dim	Key	Size	Stride	Primitive	Loop
0	Batch	1	150528	Linear	L0
1	Channels	3	50176	Linear	L1
2	InPixelY	224	224	Pooling	2*L2 + L8
3	InPixelX	224	1	Pooling	2*L3 + L9

Dim	Key	Size	Stride	Primitive	Loop
0	Batch	1	150528	Linear	L0
1	Channels	3	50176	Linear	L1
2	OutPixelY	112	224	Linear	L2
3	OutPixelX	112	1	Linear	L3

DFP: Init Loops

Dim	Key	Size	Stride	Primitive	Loop
0	None3	1	150528	Linear	L0
1	None2	3	50176	Linear	L1
2	None1	224	224	Linear	L6
3	None0	224	1	Linear	L7

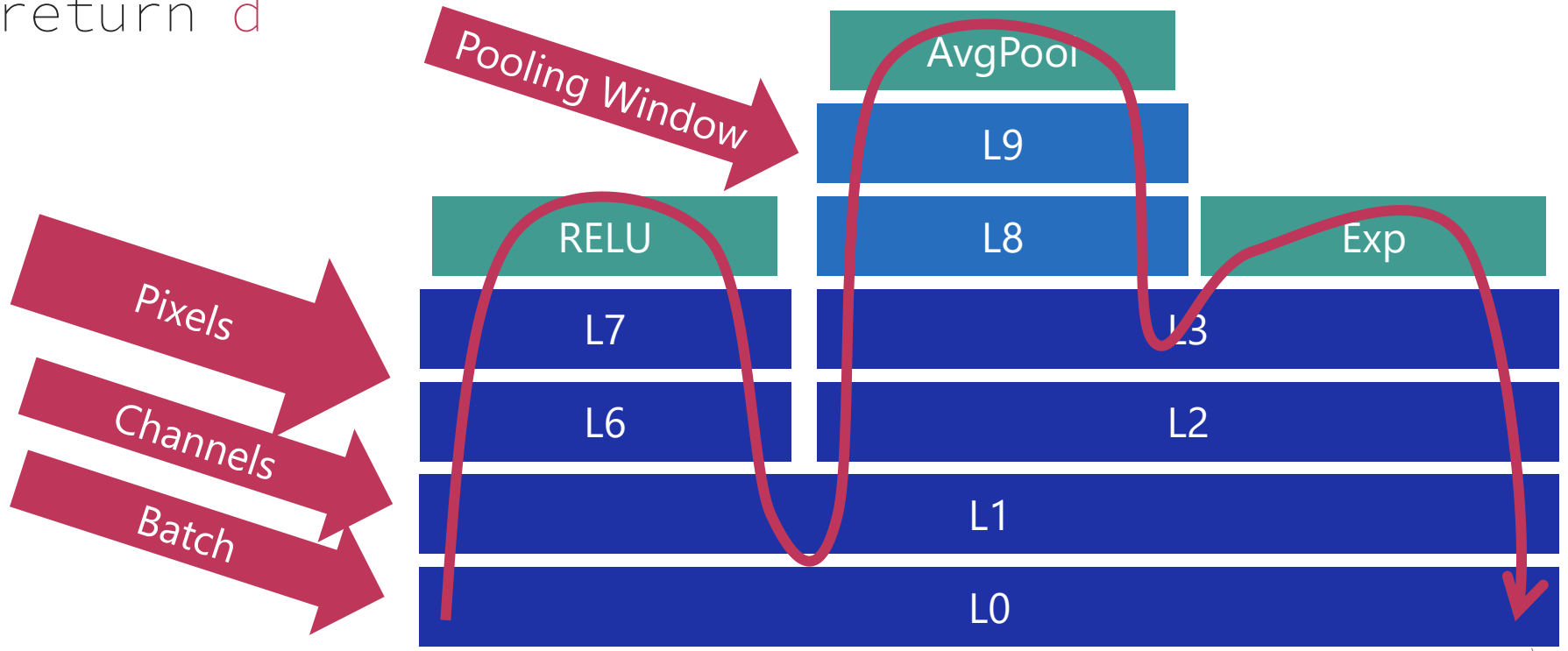


Dim	Key	Size	Stride	Primitive	Loop
0	Batch	1	150528	Linear	L0
1	Channels	3	50176	Linear	L1
2	InPixelY	224	224	Pooling	2*L2 + L8
3	InPixelX	224	1	Pooling	2*L3 + L9

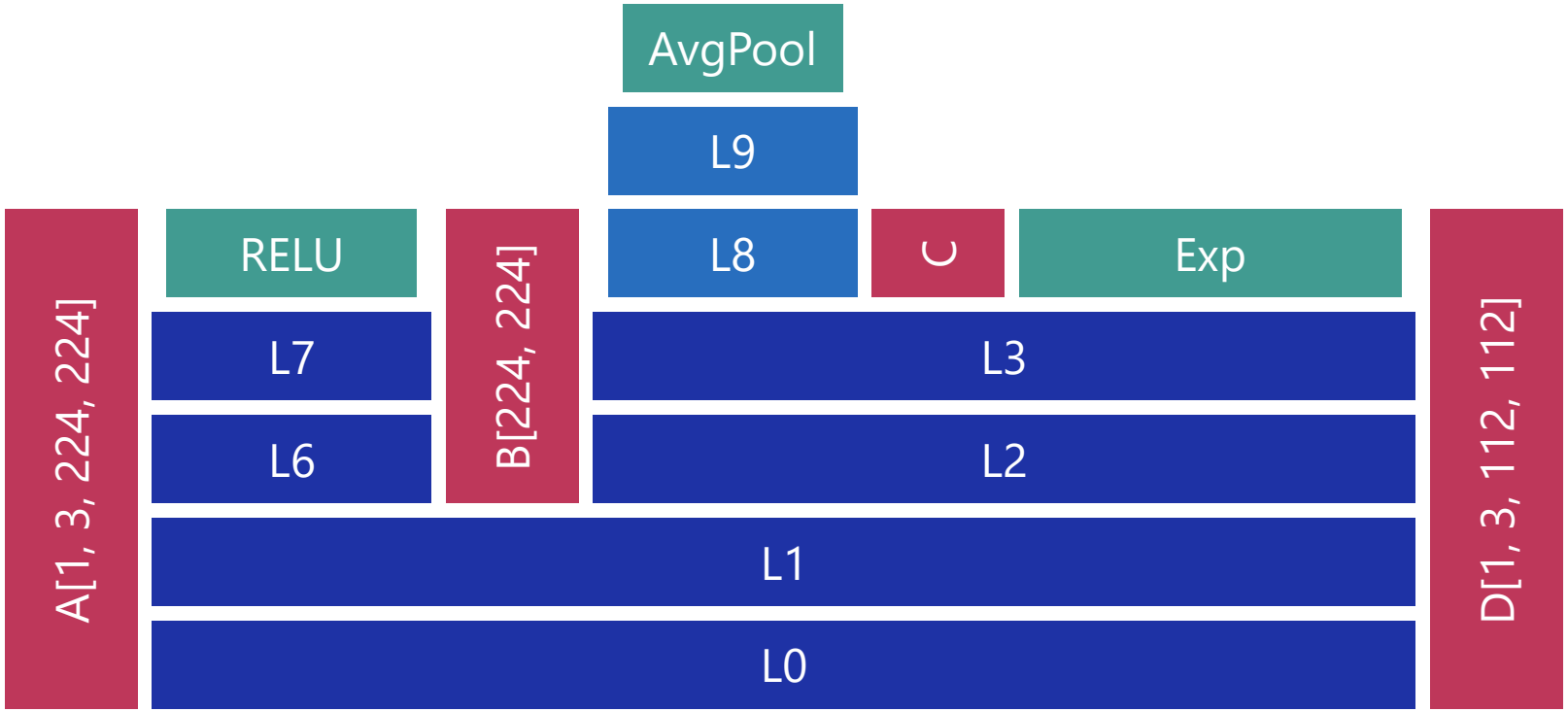
Dim	Key	Size	Stride	Primitive	Loop
0	Batch	1	150528	Linear	L0
1	Channels	3	50176	Linear	L1
2	OutPixelY	112	224	Linear	L2
3	OutPixelX	112	1	Linear	L3

DFP: Naïve loop structure

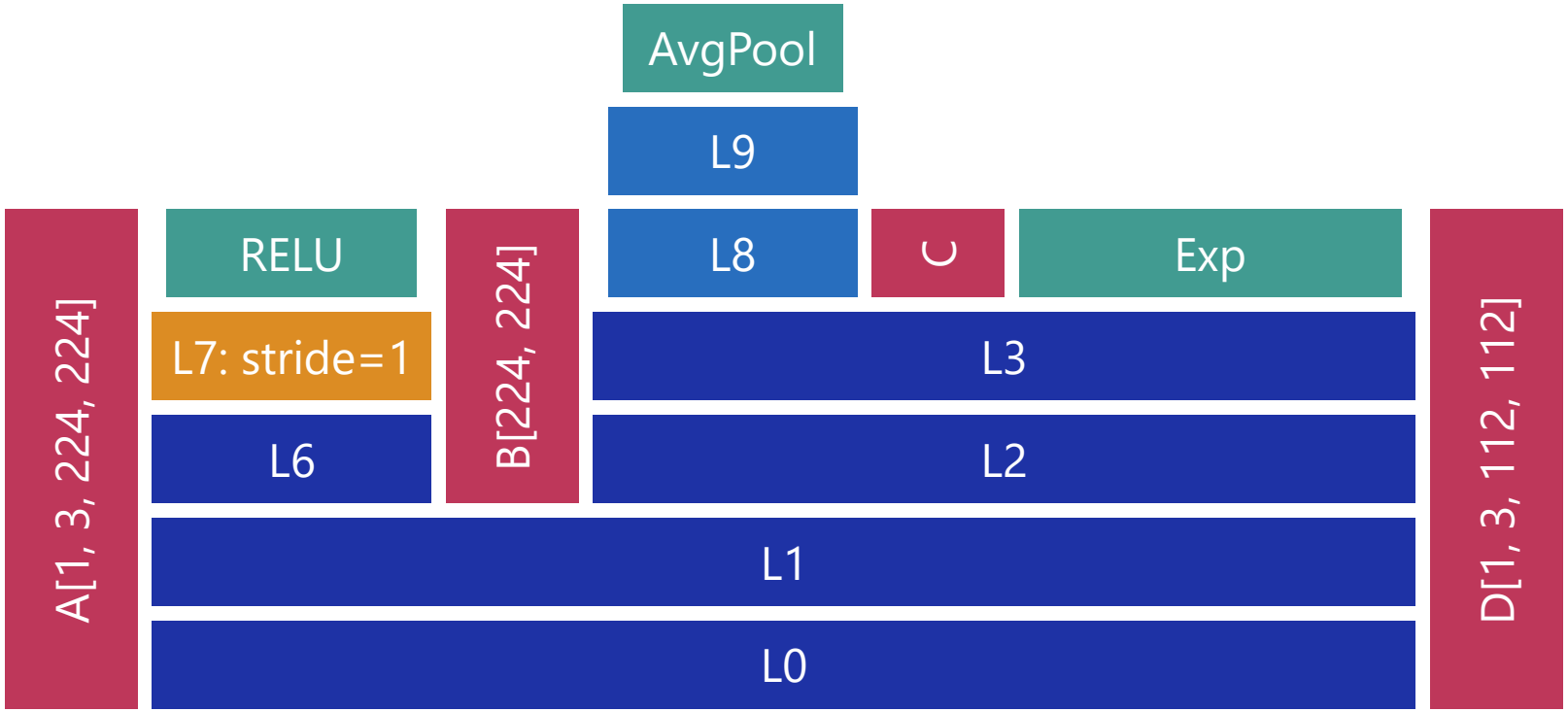
```
def forward(self, a):
        b = torch.relu(a)
        c = torch.nn.functional.avg_pool2d(b, 2, stride=(2,1))
        d = torch.exp(c)
        return d
```



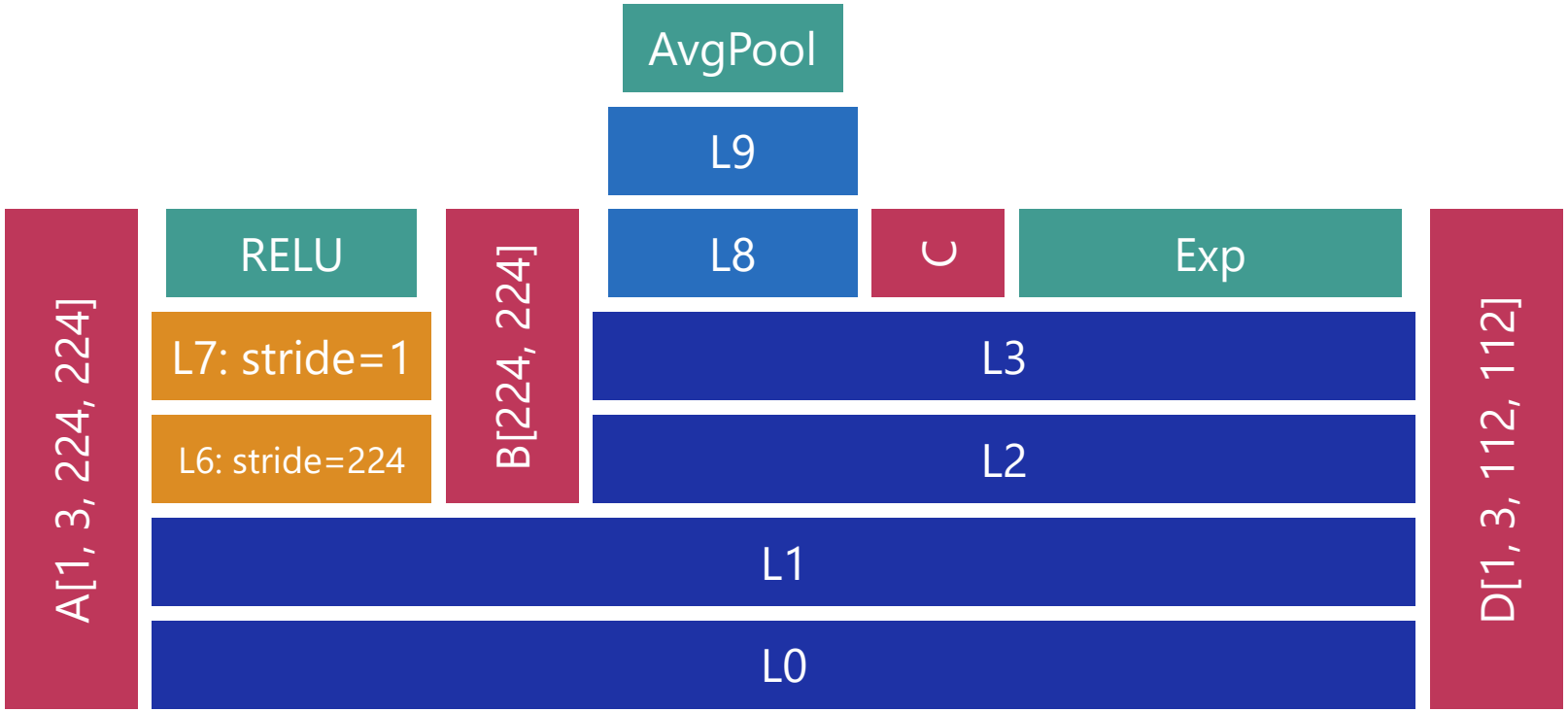
DFP: Find minimal tensors for intermediate results



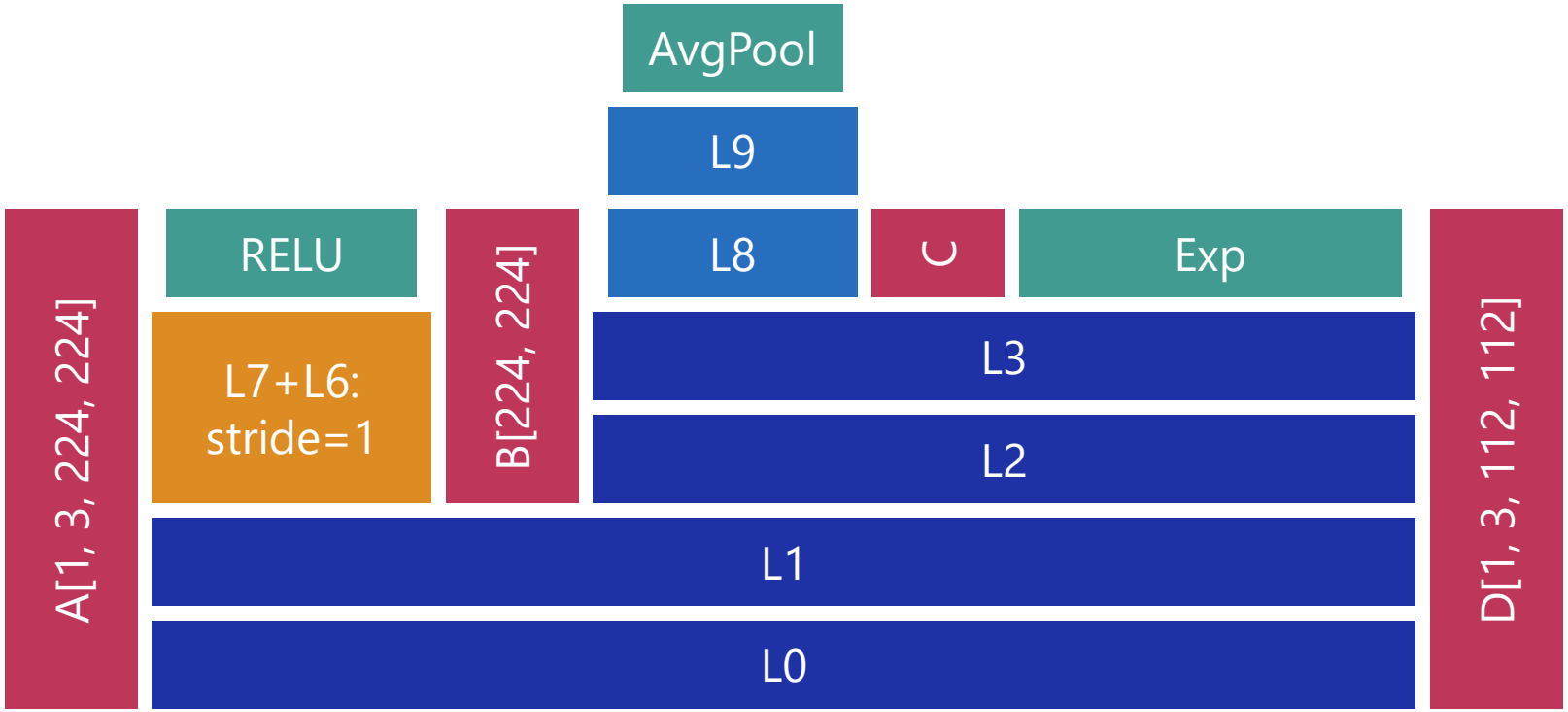
DFP: Find optimal vectorizable loops



DFP: Find optimal vectorizable loops

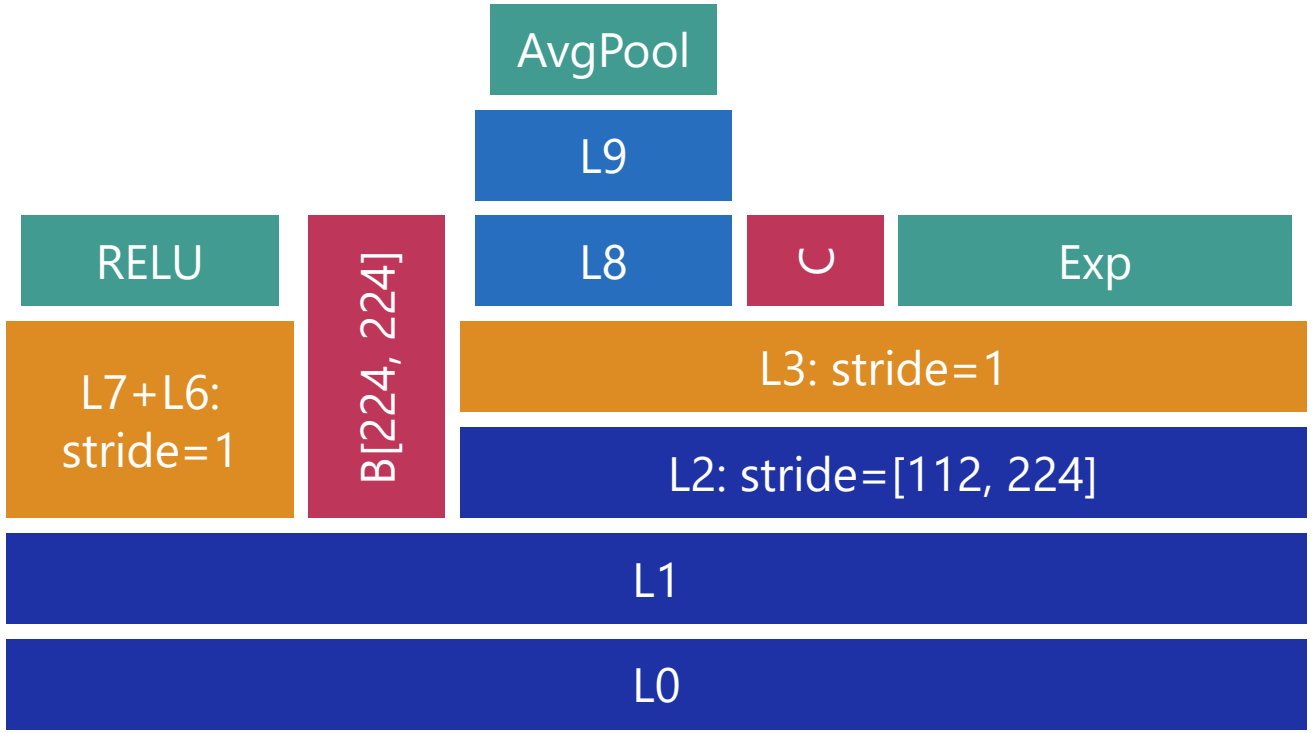


DFP: Find optimal vectorizable loops



DFP: Find optimal vectorizable loops

```
c = torch.nn.functional.avg_pool2d(b, 2, stride=(2,1))
```



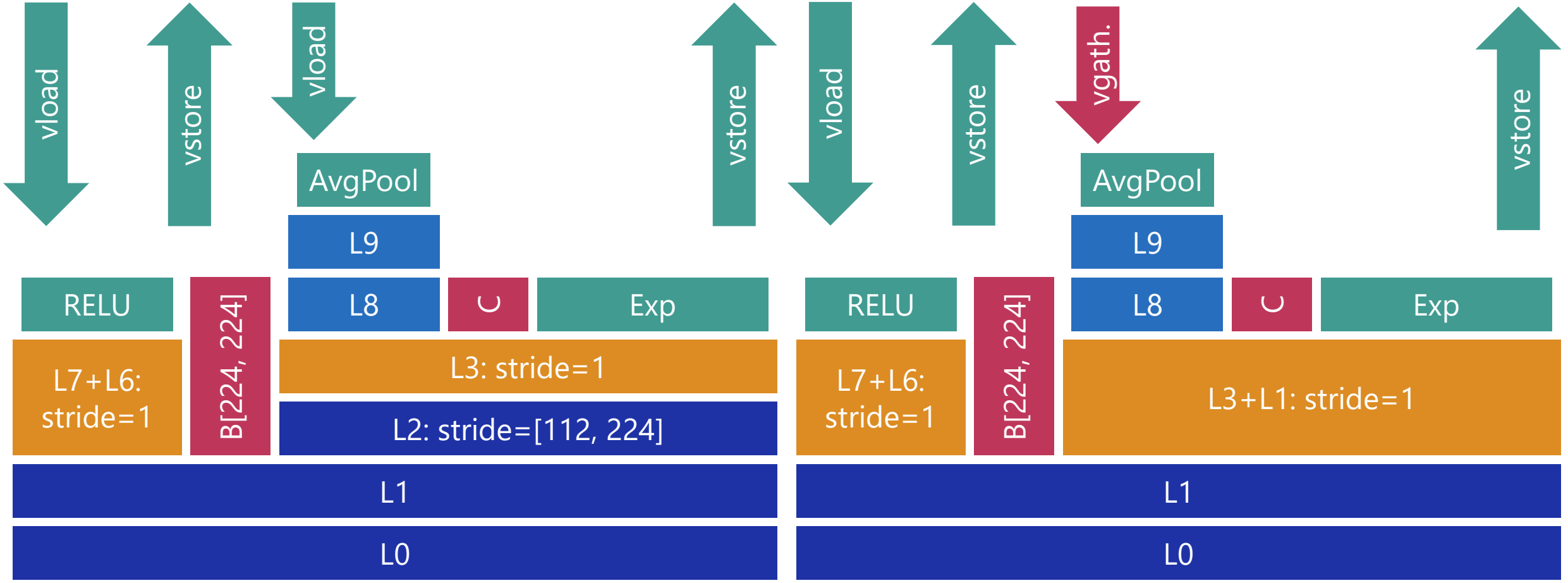
Available Vector Lengths:

- X86: [8 (AVX2), 16 (AVX512)]
- NVIDIA GPUs: [1, 2, 4, 16, 32, 64, 128, 256]
- NEC SX-Aurora: [256, 512]

Utilization:

- X86: 16: >100%
- NVIDIA GPUs: 128: 87.5%
- NEC SX-Aurora: 256: 43.75%

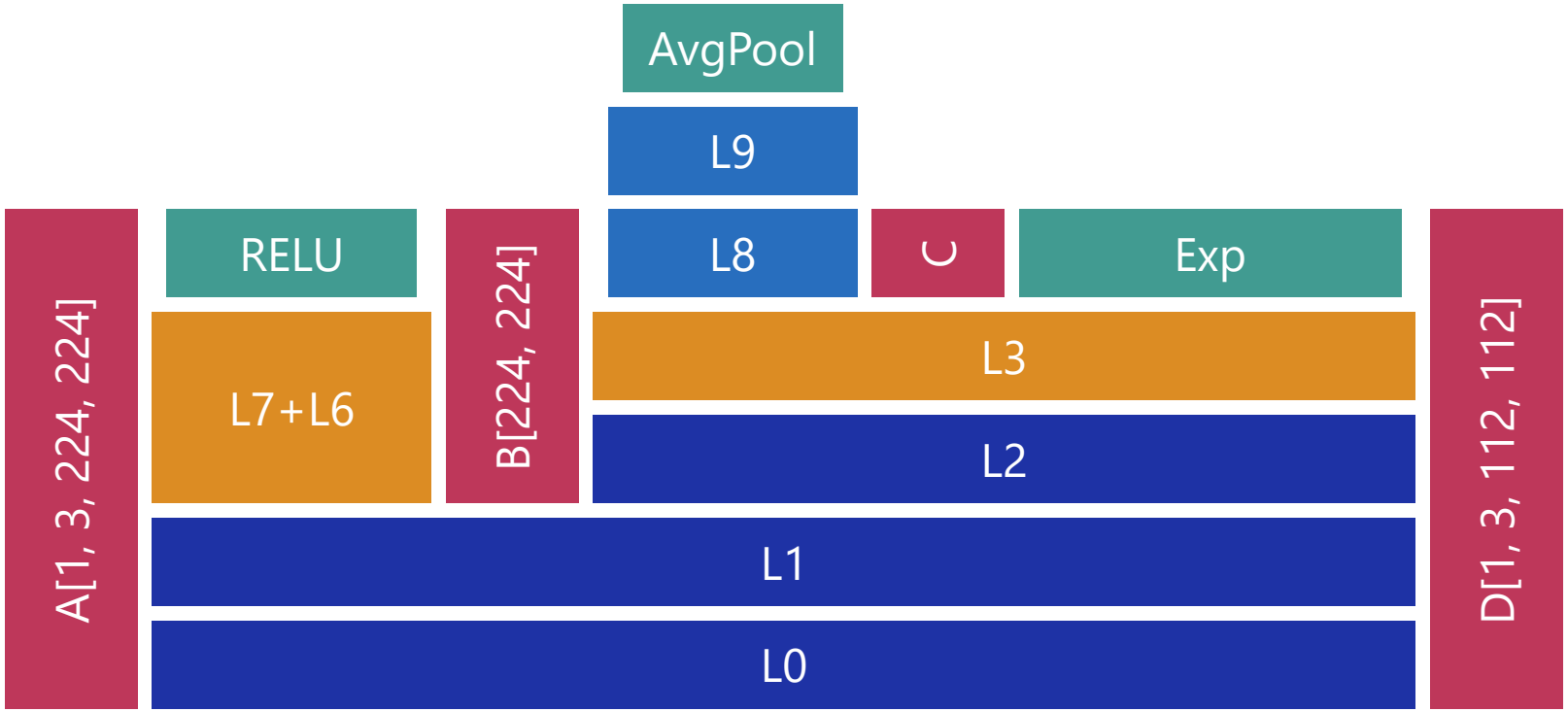
DFP: Different vectorization dependent on vector length



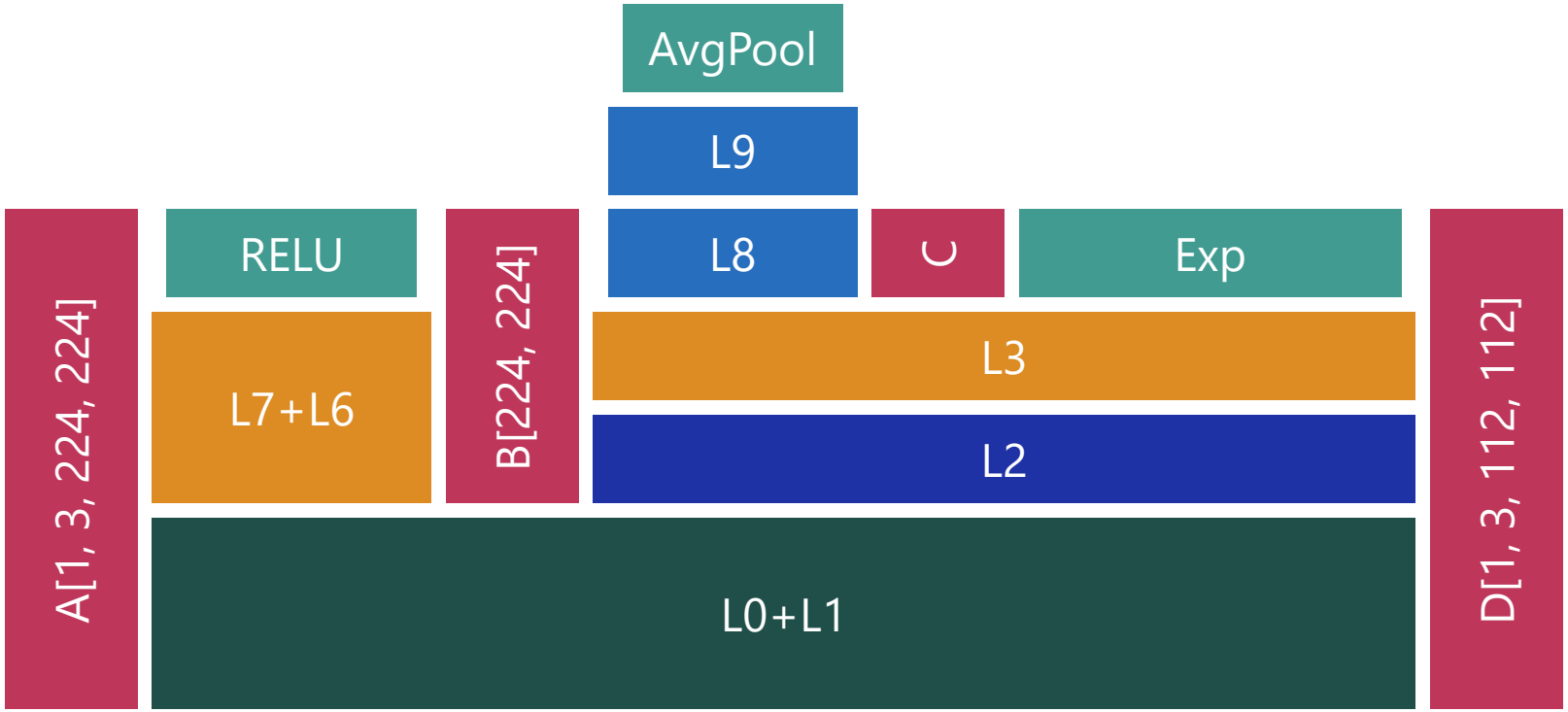
X86 and NVIDIA

NEC SX-Aurora

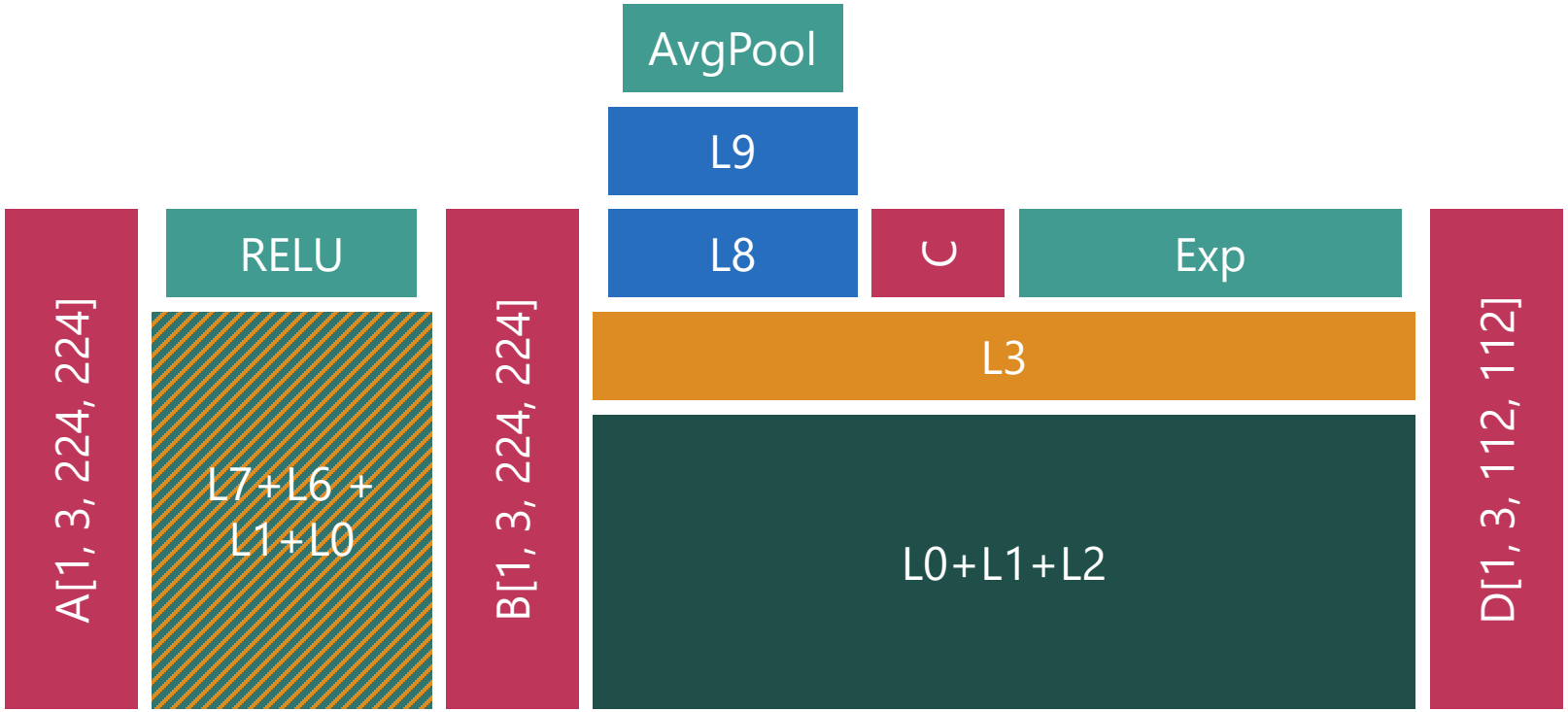
DFP: Find optimal parallelization



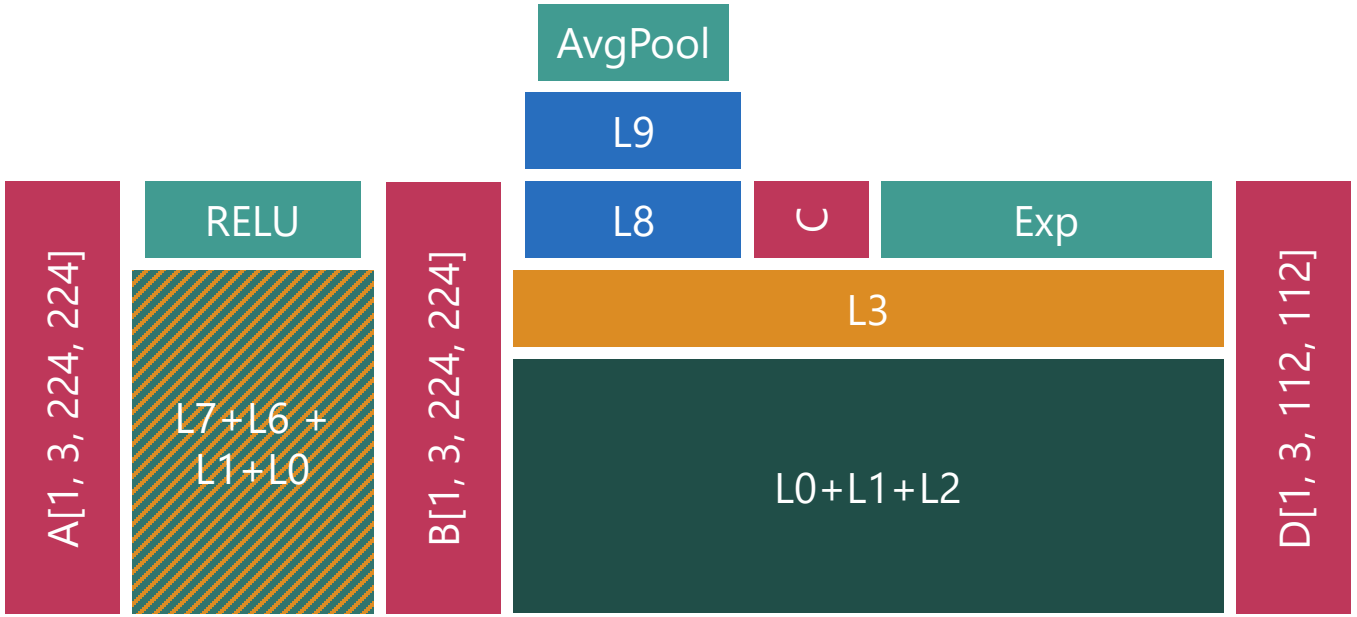
DFP: Find optimal parallelization



DFP: Find optimal parallelization



DFP: DFP-IR Generation




```
elementwise(L0+L1+L6+L7):
   R0 = vld(A, …)
   R1 = 0
   R2 = max(R0, R1)
   vst(B, …) = R2

cores(L0+L1+L2):
   vector(L3):
      R3 = 0
      for(L8):
         for(L9):
            R4 = vld(B, …)
            R3 = R3 + R4
      R5 = R3 / area(…)
      R6 = exp(R5)
      vst(C, …) = R6
```

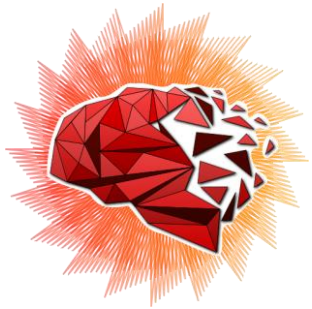
DFP: DFP-IR to Source Code

```
elementwise(L0+L1+L6+L7):
   R0 = vld(A, …)
   R1 = 0
   R2 = max(R0, R1)
   vst(B, …) = R2

cores(L0+L1+L2):
   vector(L3):
      R3 = 0
      for(L8):
         for(L9):
            R4 = vld(B, …)
            R3 = R3 + R4
      R5 = R3 / area(…)
      R6 = exp(R5)
      vst(C, …) = R6
```

```
__global__ void ...(...) {
    int L0L1L6L7 = blockIdx.x * blockSize + threadIdx.x;
    if(L0L1L6L7 < 150528) {
        float R0 = A[...];
        float R1 = 0;
        float R2 = max(R0, R1);
        B[...] = R2;
    }
}

__global__ void ...(...) {
    int L0L1L2 = blockIdx.x …;
    for(int L3 = threadIdx.x; L3 < 112; L3 += blockSize) {
        float R3 = 0;
        #pragma unroll
        for(int L8 = 0; L8 < 2; L8++) {
            #pragma unroll
            for(int L9 = 0; L9 < 2; L9++) {
                float R4 = B[...];
                R3 = R3 + R4;
            }
        }
        float R5 = R3 / area(...);
        float R6 = exp(R5);
        C[...] = R6;
    }
}
```



\Orchestrating a brighter world NEC

Thanks for listening!

nicolas.weber@neclab.eu