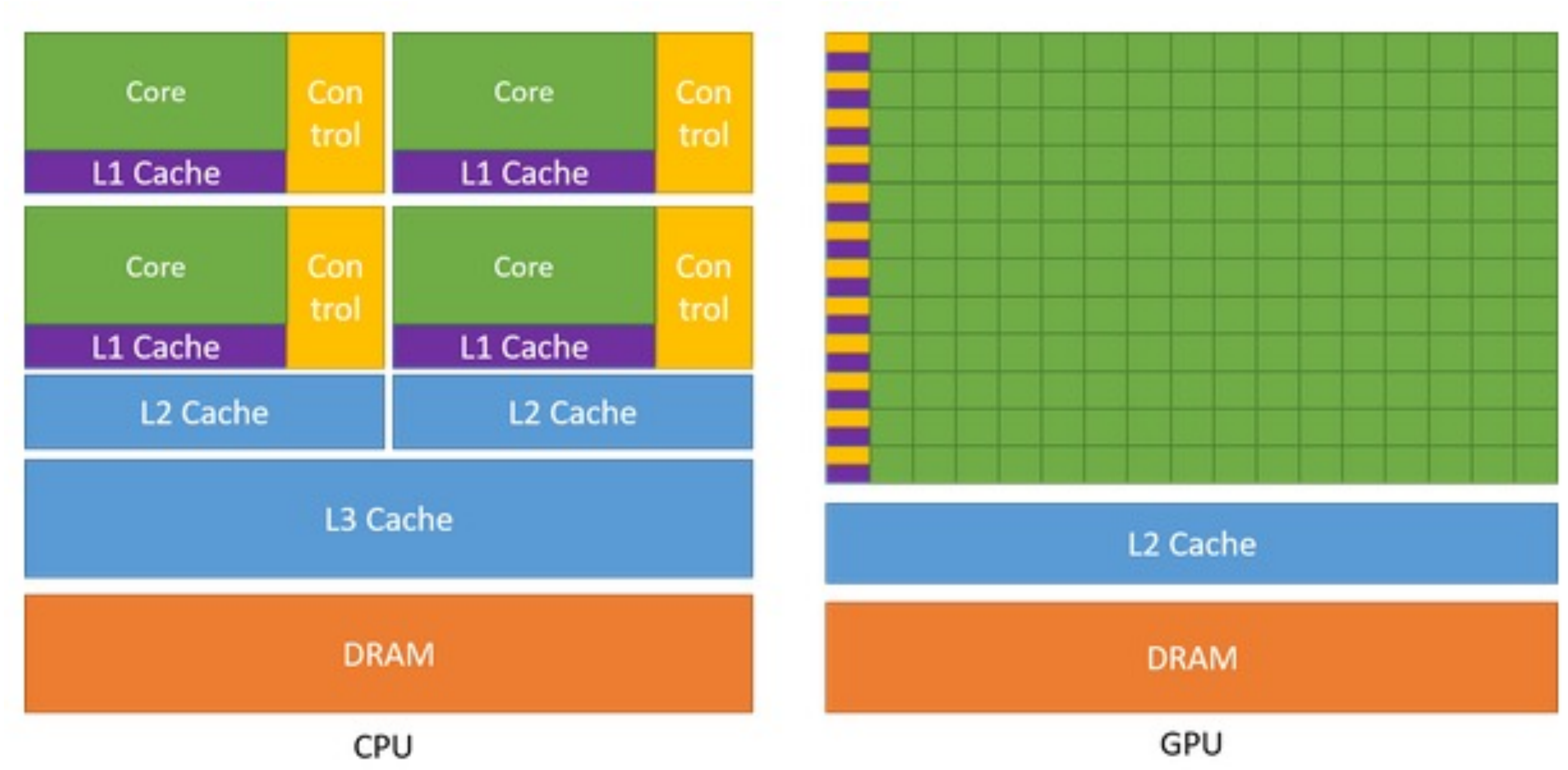
# «CUDAification» of PACE

Optimization of PwPA kernels

Sangiorgi Marco
University of Bologna
2025

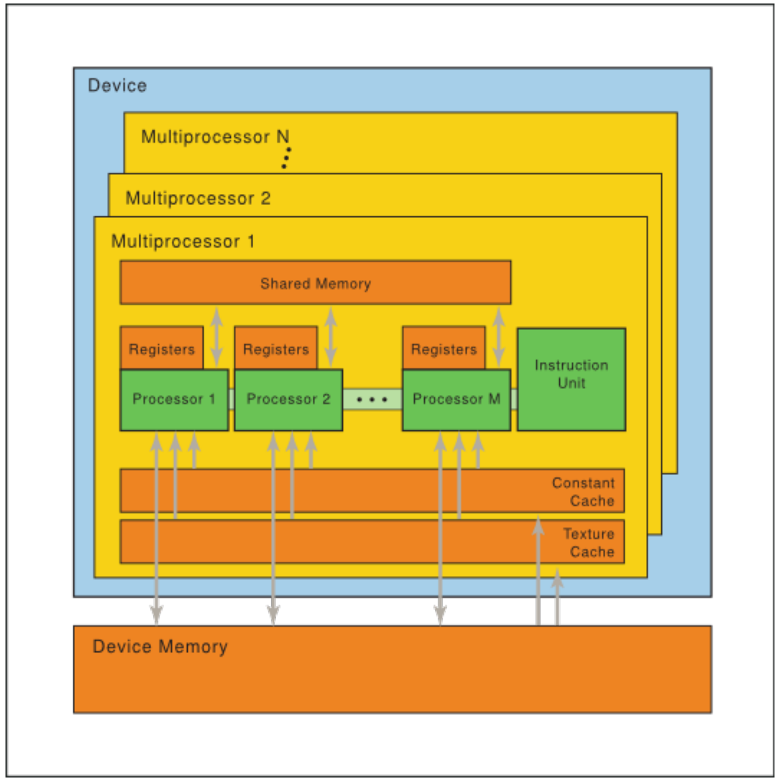
# Intro to CUDA – Hardware comparison



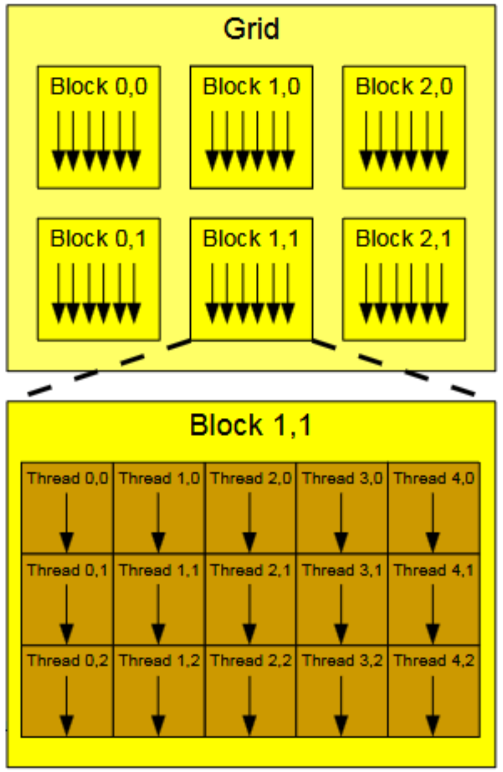
https://docs.nvidia.com/cuda/cuda-c-programming-guide/

# Intro to CUDA – Computational units

**Hardware view**

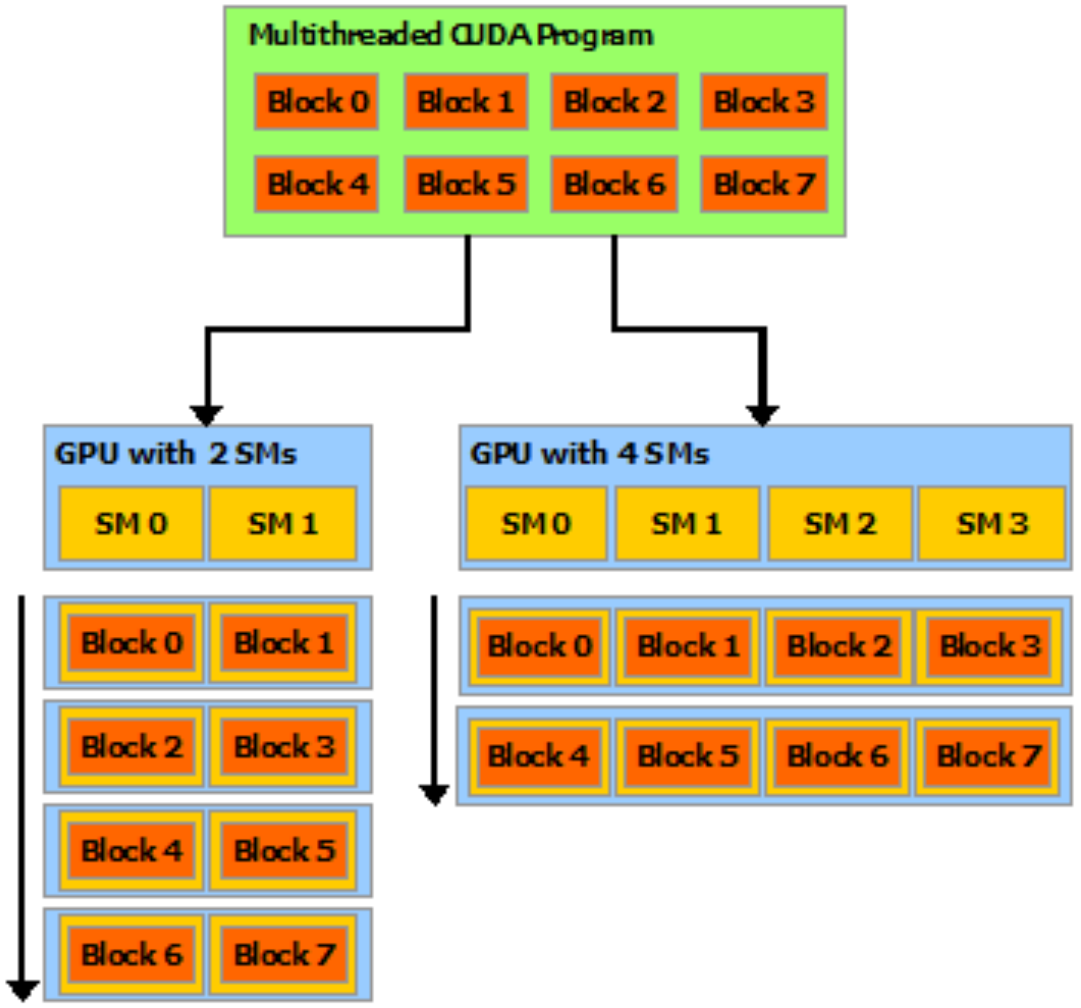


**Software view**



https://docs.nvidia.com/cuda/cuda-c-programming-guide/

# Intro to CUDA – Automatic Scalability



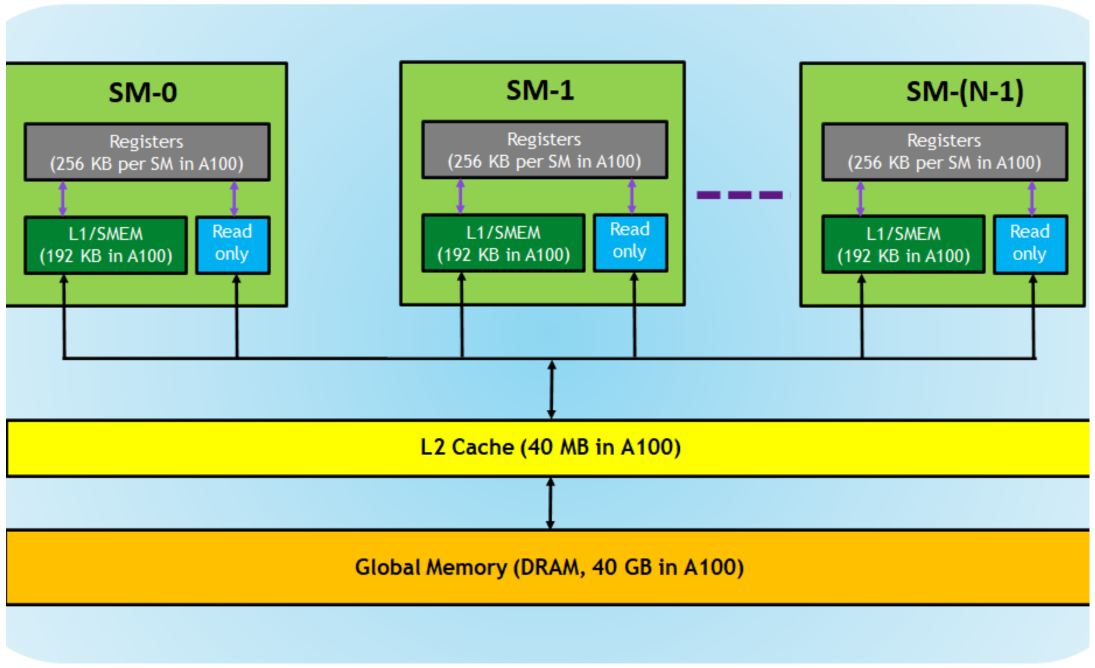
https://docs.nvidia.com/cuda/cuda-c-programming-guide/

# Intro to CUDA – Memory

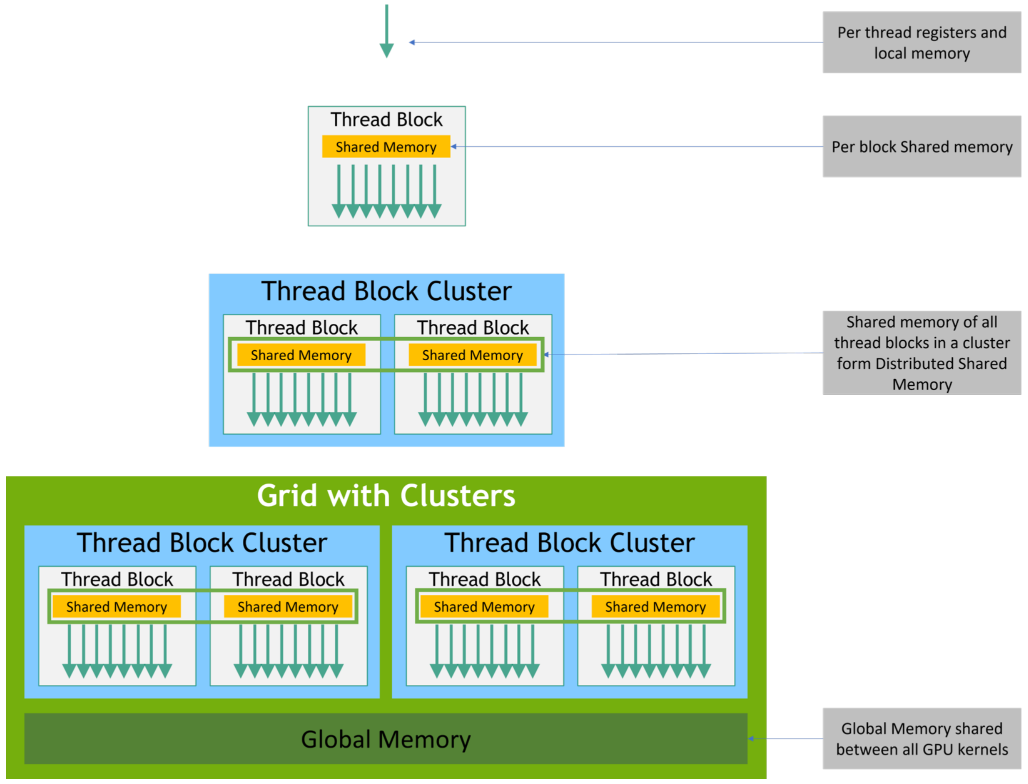
Each memory access moves 32 or 128 consecutive bytes

- So, if a thread just needs a single float (4B), this results in 32B or 128B being moved

## Hardware view



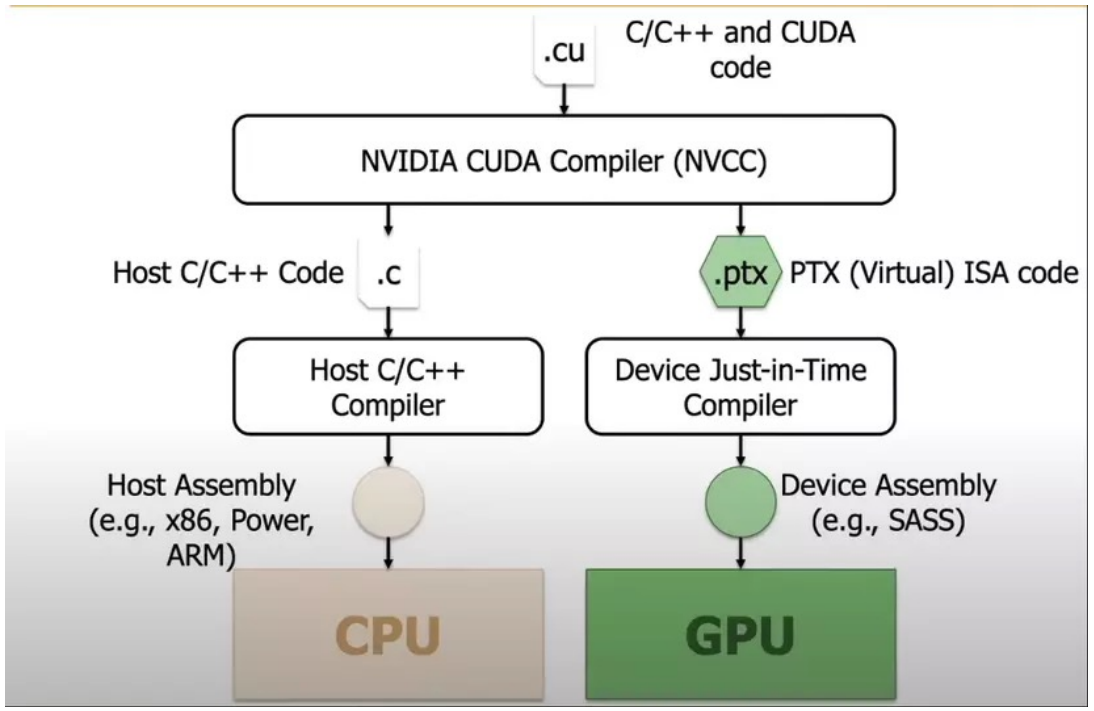
## Software view



https://docs.nvidia.com/cuda/cuda-c-programming-guide/

## Intro to CUDA – Compiler

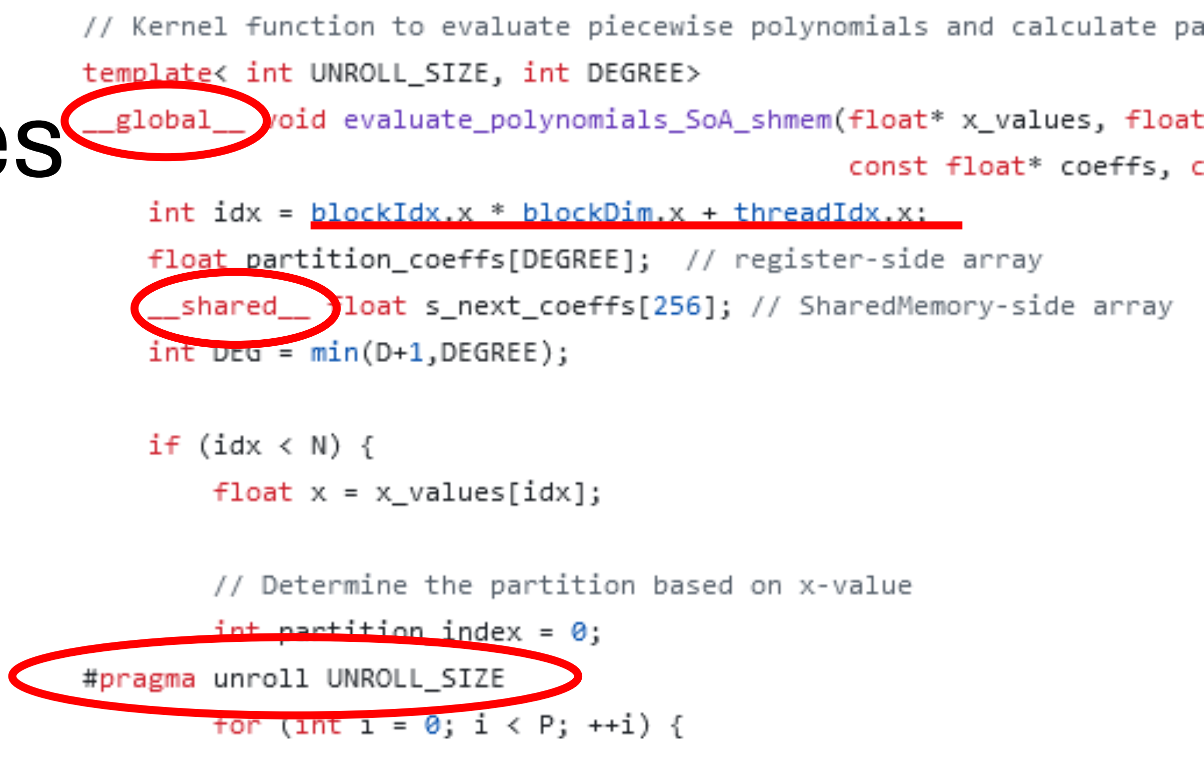
**Simplified Compilation path**


.cu
C/C++ and CUDA code
NVIDIA CUDA Compiler (NVCC)
Host C/C++ Code
.c
.ptx
PTX (Virtual) ISA code
Host C/C++ Compiler
Device Just-in-Time Compiler
Host Assembly (e.g., x86, Power, ARM)
Device Assembly (e.g., SASS)
CPU
GPU

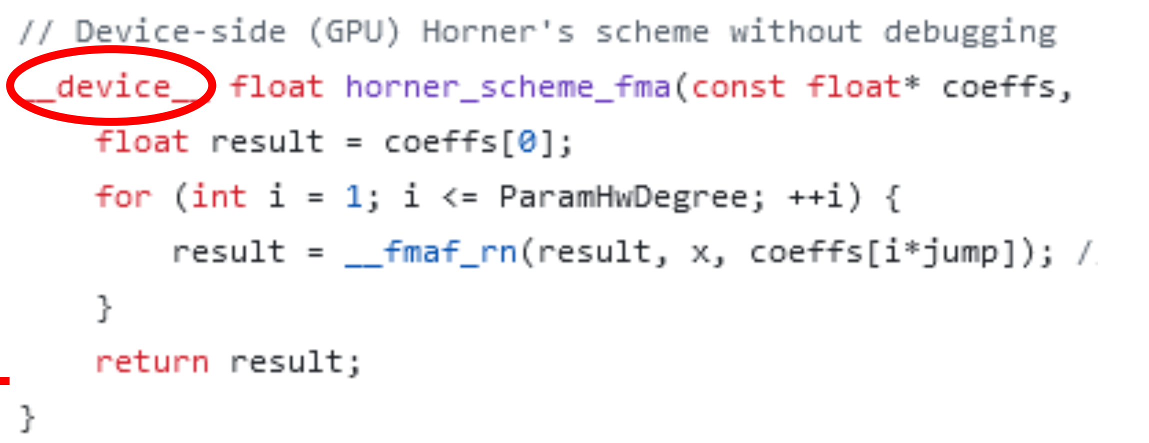
https://docs.nvidia.com/cuda/cuda-c-programming-guide/

# Intro to CUDA – Directives

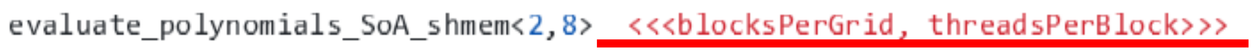
- _global_: declares host/device callable kernels
- _shared_: declares shared memory allocations
- _device__ : declares device callable kernels
- #pragmas: define nvcc pre-processing directives
- blockIdx, blockDim, threadIdx, threadDim : run-time thread variables to access grid/block location
- <<<gridDim, blockDim>>> : kernel directives to specify grid/block dimensions



```
// Kernel function to evaluate piecewise polynomials and calculate pa
template< int UNROLL_SIZE, int DEGREE>
__global__ void evaluate_polynomials_SoA_shmem(float* x_values, float
                                const float* coeffs, c
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    float partition_coeffs[DEGREE];  // register-side array
    __shared__ float s_next_coeffs[256]; // SharedMemory-side array
    int DEG = min(D+1,DEGREE);

    if (idx < N) {
        float x = x_values[idx];

        // Determine the partition based on x-value
        int partition_index = 0;
#pragma unroll UNROLL_SIZE
        for (int i = 0; i < P; ++i) {
```



```
// Device-side (GPU) Horner's scheme without debugging
__device__ float horner_scheme_fma(const float* coeffs,
    float result = coeffs[0];
    for (int i = 1; i <= ParamHwDegree; ++i) {
        result = __fmaf_rn(result, x, coeffs[i*jump]); /
    }
    return result;
}
```



```
evaluate_polynomials_SoA_shmem<2,8> <<<blocksPerGrid, threadsPerBlock>>>
```

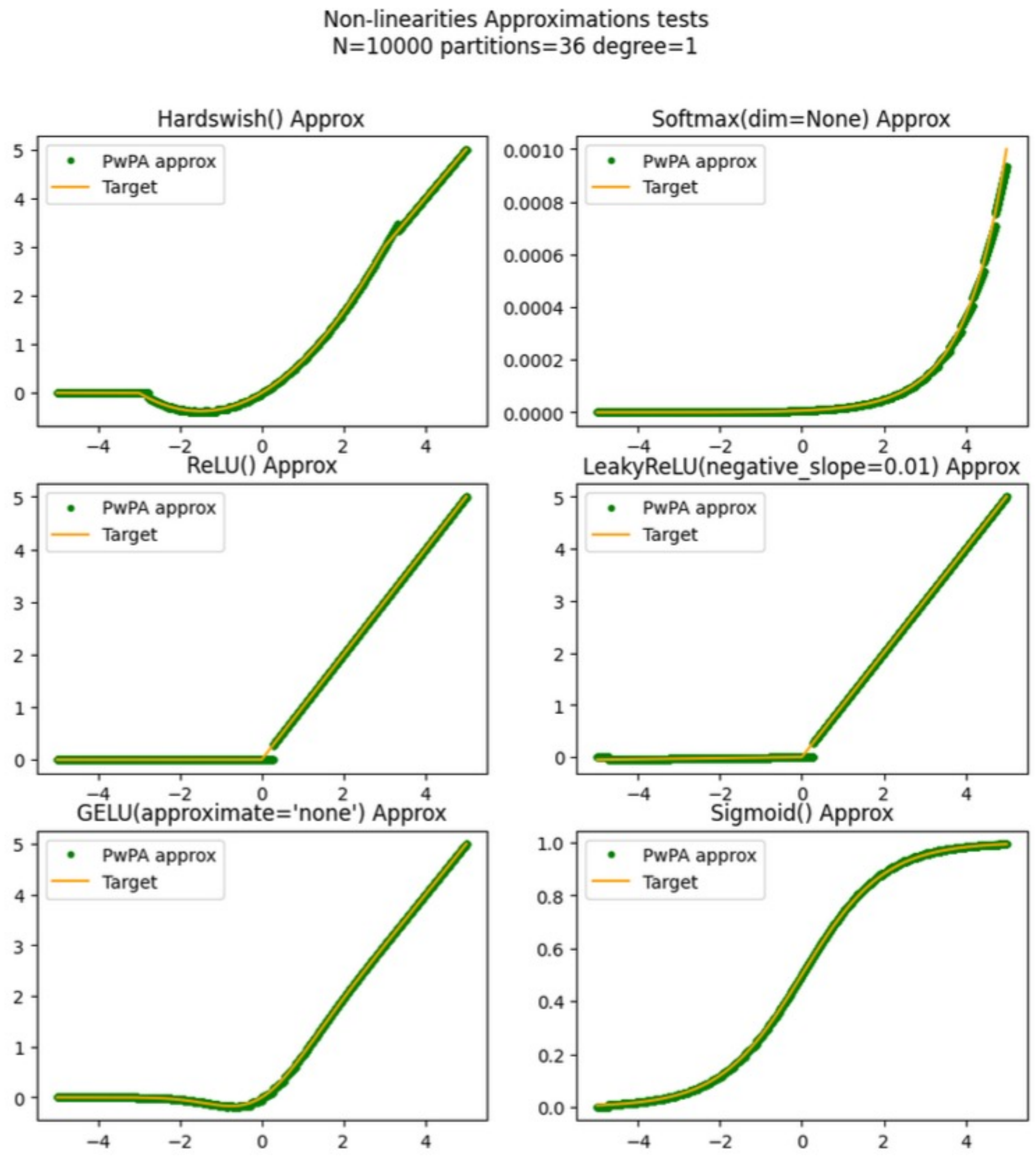
https://docs.nvidia.com/cuda/cuda-c-programming-guide/
https://github.com/SangioAI/torchPACE/blob/main/extra/test_optimizations.cu

# PwPA – Intro

- Point-wise Polynomial Approximation of an arbitrary function $p(x)$ using:
- Partitions: divide the x-axis in subportions
- Coefficients: approximate each partition with a polynomial of degree D and save the polynomial coefficients

  $a_0 \ \ldots \ a_n$
- Horner's method:

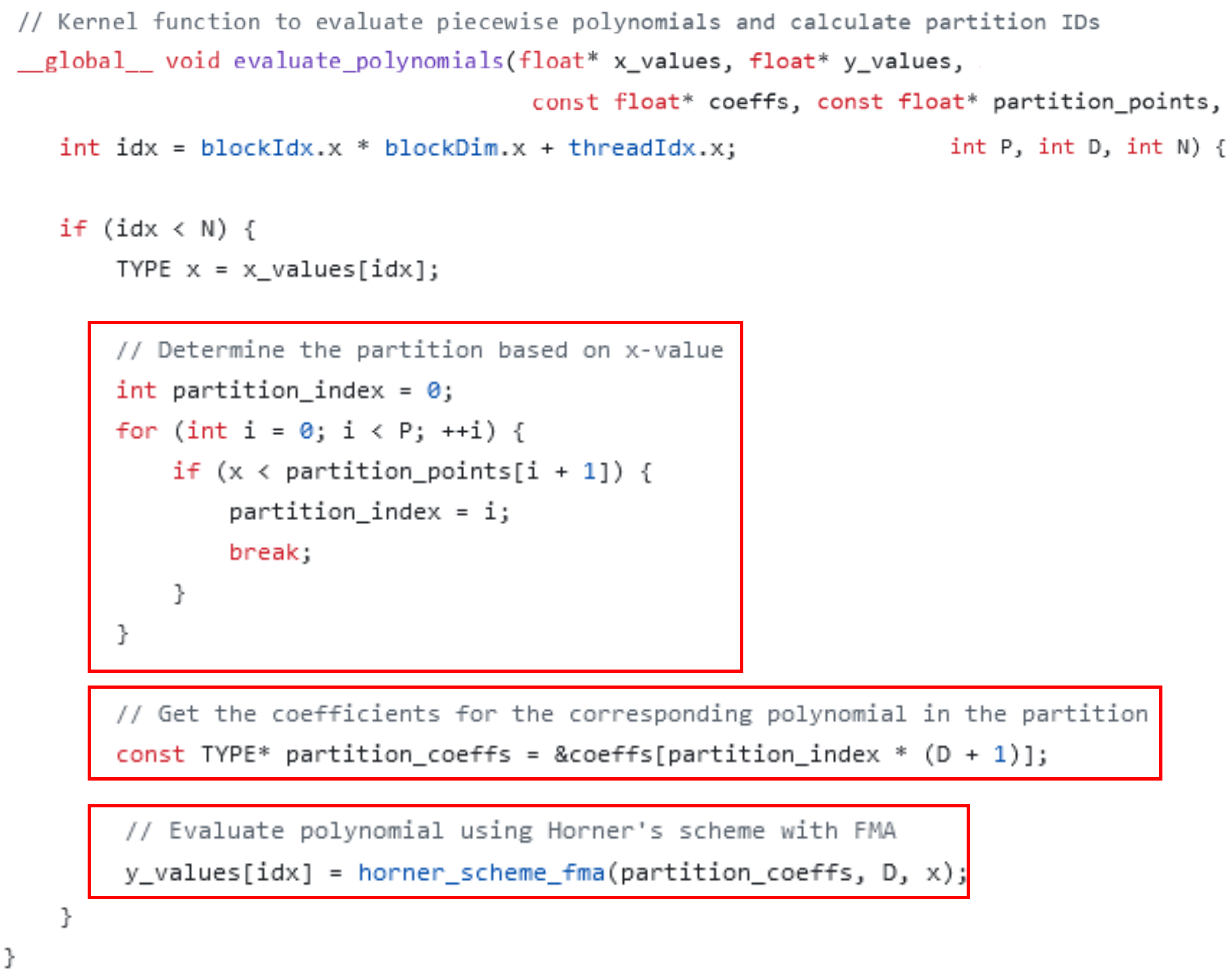
$$\begin{aligned} p(x) &= a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_n x^n \\ &= a_0 + x\bigg(a_1 + x\Big(a_2 + x\big(a_3 + \cdots + x(a_{n-1} + x\,a_n)\cdots\big)\Big)\bigg) \end{aligned}$$



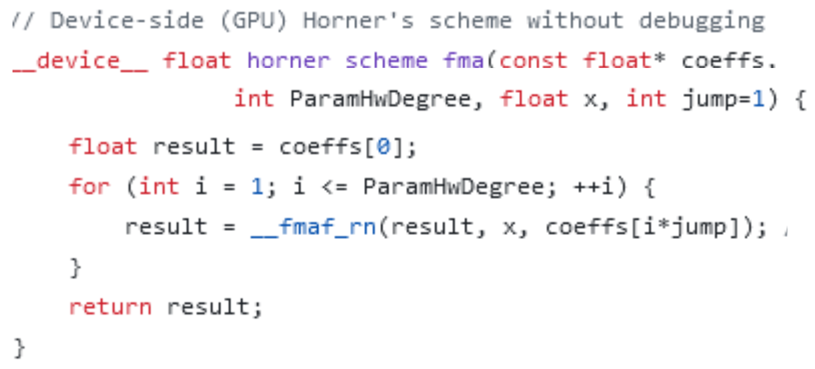
# PwPA – Code

Input-Output stationary algorithm:

- Partition ID extraction: O(P)
- Partition Coefficient pointer extraction: O(1)
- Horner scheme: O(D)



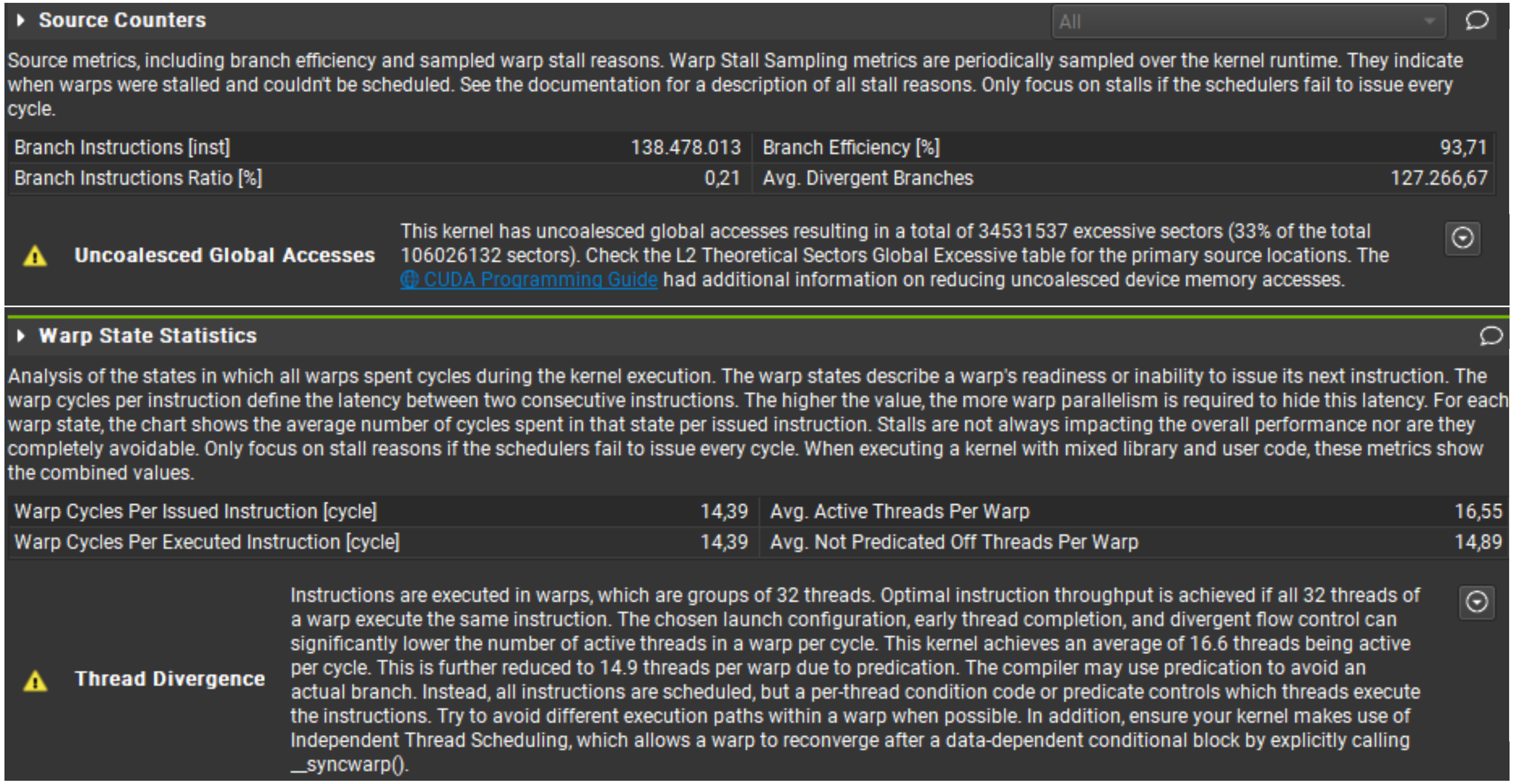
```
// Kernel function to evaluate piecewise polynomials and calculate partition IDs
__global__ void evaluate_polynomials(float* x_values, float* y_values,
                                     const float* coeffs, const float* partition_points,
    int idx = blockIdx.x * blockDim.x + threadIdx.x;          int P, int D, int N) {

    if (idx < N) {
        TYPE x = x_values[idx];

        // Determine the partition based on x-value
        int partition_index = 0;
        for (int i = 0; i < P; ++i) {
            if (x < partition_points[i + 1]) {
                partition_index = i;
                break;
            }
        }

        // Get the coefficients for the corresponding polynomial in the partition
        const TYPE* partition_coeffs = &coeffs[partition_index * (D + 1)];

        // Evaluate polynomial using Horner's scheme with FMA
        y_values[idx] = horner_scheme_fma(partition_coeffs, D, x);
    }
}
```



```
// Device-side (GPU) Horner's scheme without debugging
__device__ float horner_scheme_fma(const float* coeffs,
                int ParamHwDegree, float x, int jump=1) {
    float result = coeffs[0];
    for (int i = 1; i <= ParamHwDegree; ++i) {
        result = __fmaf_rn(result, x, coeffs[i*jump]);
    }
    return result;
}
```

https://github.com/SangioAI/torchPACE/blob/main/extra/test_optimizations.cu

# PwPA – Base kernel analysis



**Source Counters**

Source metrics, including branch efficiency and sampled warp stall reasons. Warp Stall Sampling metrics are periodically sampled over the kernel runtime. They indicate when warps were stalled and couldn't be scheduled. See the documentation for a description of all stall reasons. Only focus on stalls if the schedulers fail to issue every cycle.

| | | | |
|---|---|---|---|
| Branch Instructions [inst] | 138.478.013 | Branch Efficiency [%] | 93,71 |
| Branch Instructions Ratio [%] | 0,21 | Avg. Divergent Branches | 127.266,67 |

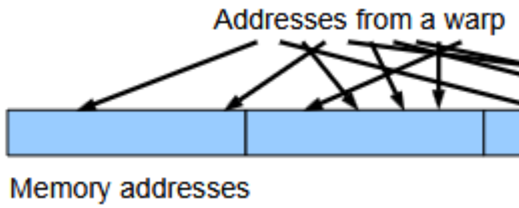
**Uncoalesced Global Accesses** This kernel has uncoalesced global accesses resulting in a total of 34531537 excessive sectors (33% of the total 106026132 sectors). Check the L2 Theoretical Sectors Global Excessive table for the primary source locations. The CUDA Programming Guide had additional information on reducing uncoalesced device memory accesses.

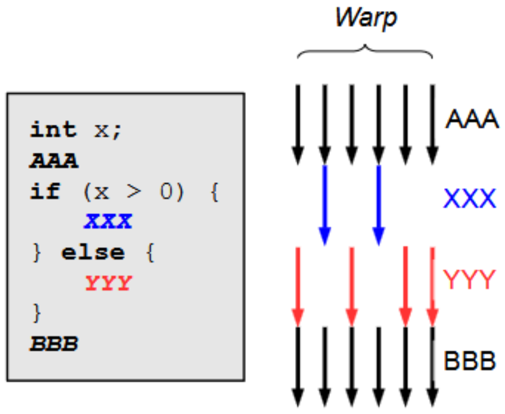
**Warp State Statistics**

Analysis of the states in which all warps spent cycles during the kernel execution. The warp states describe a warp's readiness or inability to issue its next instruction. The warp cycles per instruction define the latency between two consecutive instructions. The higher the value, the more warp parallelism is required to hide this latency. For each warp state, the chart shows the average number of cycles spent in that state per issued instruction. Stalls are not always impacting the overall performance nor are they completely avoidable. Only focus on stall reasons if the schedulers fail to issue every cycle. When executing a kernel with mixed library and user code, these metrics show the combined values.

| | | | |
|---|---|---|---|
| Warp Cycles Per Issued Instruction [cycle] | 14,39 | Avg. Active Threads Per Warp | 16,55 |
| Warp Cycles Per Executed Instruction [cycle] | 14,39 | Avg. Not Predicated Off Threads Per Warp | 14,89 |

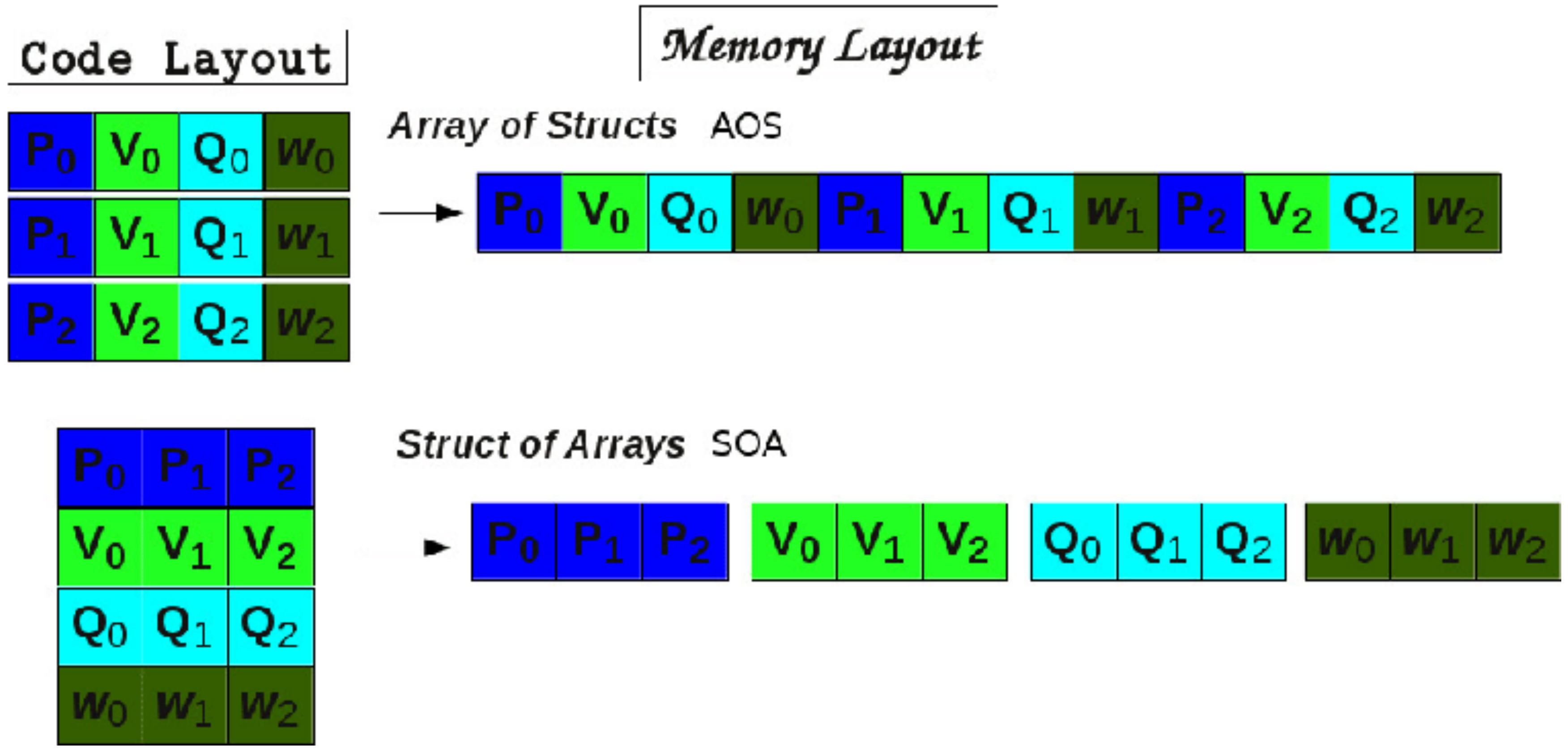
**Thread Divergence** Instructions are executed in warps, which are groups of 32 threads. Optimal instruction throughput is achieved if all 32 threads of a warp execute the same instruction. The chosen launch configuration, early thread completion, and divergent flow control can significantly lower the number of active threads in a warp per cycle. This kernel achieves an average of 16.6 threads being active per cycle. This is further reduced to 14.9 threads per warp due to predication. The compiler may use predication to avoid an actual branch. Instead, all instructions are scheduled, but a per-thread condition code or predicate controls which threads execute the instructions. Try to avoid different execution paths within a warp when possible. In addition, ensure your kernel makes use of Independent Thread Scheduling, which allows a warp to reconverge after a data-dependent conditional block by explicitly calling __syncwarp().





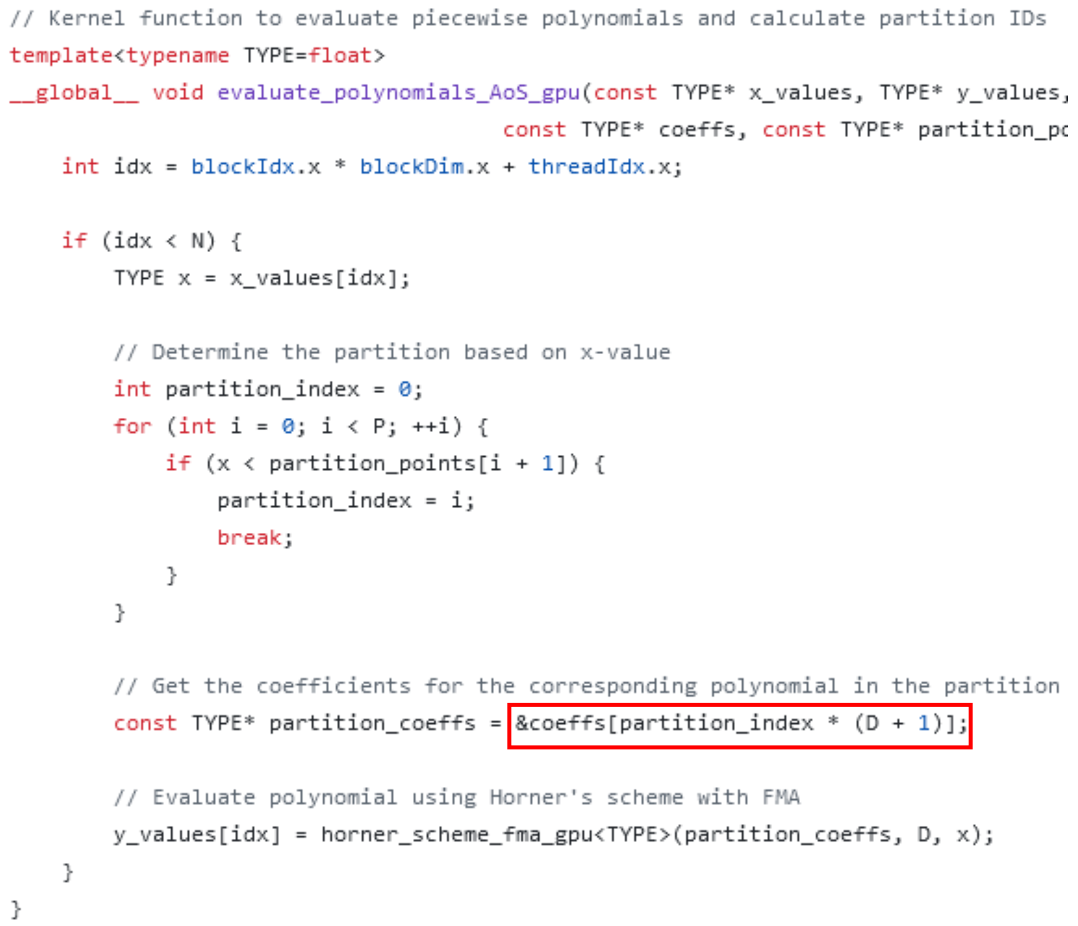
```
int x;
AAA
if (x > 0) {
    XXX
} else {
    YYY
}
BBB
```

# PwPA – SoA vs AoS

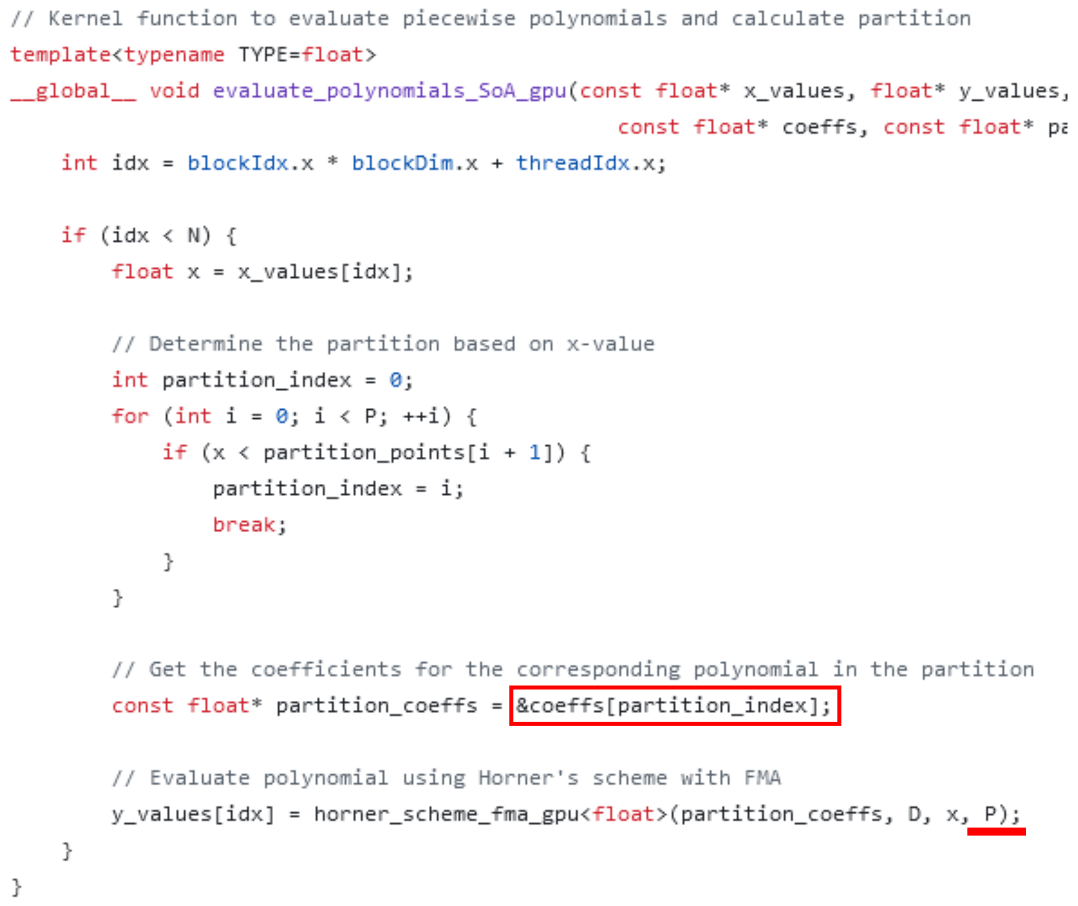


https://www.researchgate.net/figure/AOS-vs-SOA-data-access-patterns_fig27_261860648

# PwPA – AoS vs SoA coefficients



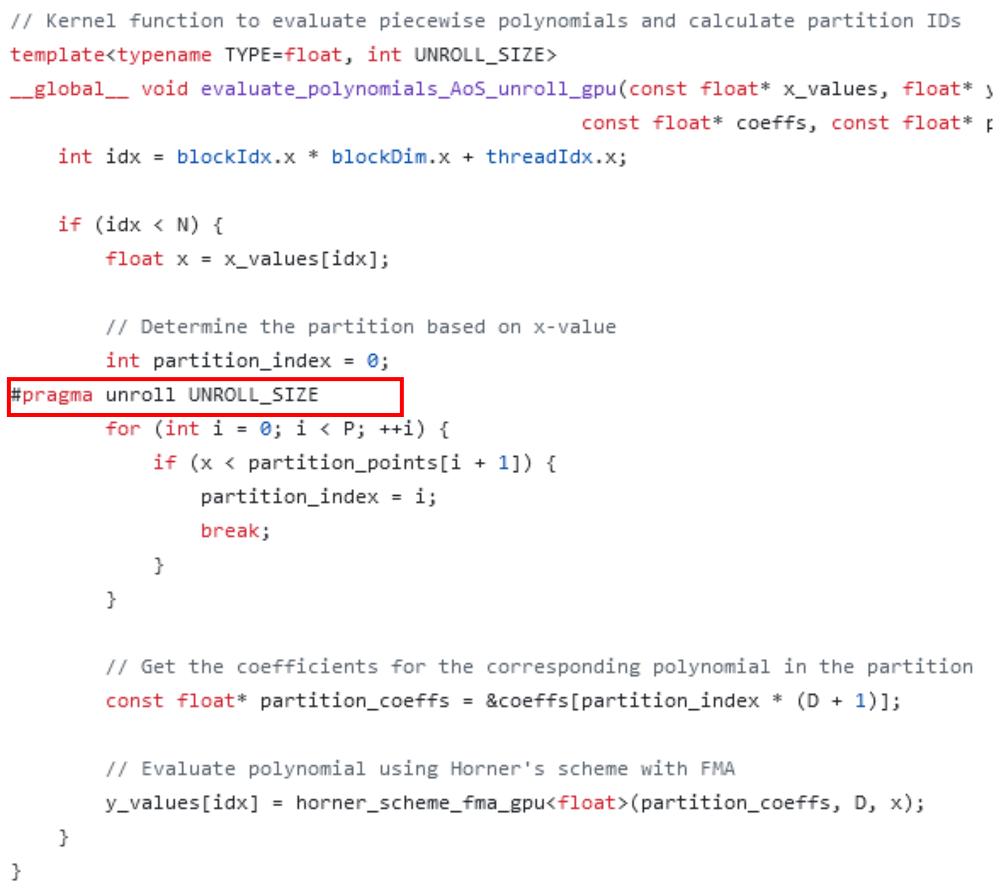
```
// Kernel function to evaluate piecewise polynomials and calculate partition IDs
template<typename TYPE=float>
__global__ void evaluate_polynomials_AoS_gpu(const TYPE* x_values, TYPE* y_values,
                                             const TYPE* coeffs, const TYPE* partition_p
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < N) {
        TYPE x = x_values[idx];

        // Determine the partition based on x-value
        int partition_index = 0;
        for (int i = 0; i < P; ++i) {
            if (x < partition_points[i + 1]) {
                partition_index = i;
                break;
            }
        }

        // Get the coefficients for the corresponding polynomial in the partition
        const TYPE* partition_coeffs = &coeffs[partition_index * (D + 1)];

        // Evaluate polynomial using Horner's scheme with FMA
        y_values[idx] = horner_scheme_fma_gpu<TYPE>(partition_coeffs, D, x);
    }
}
```



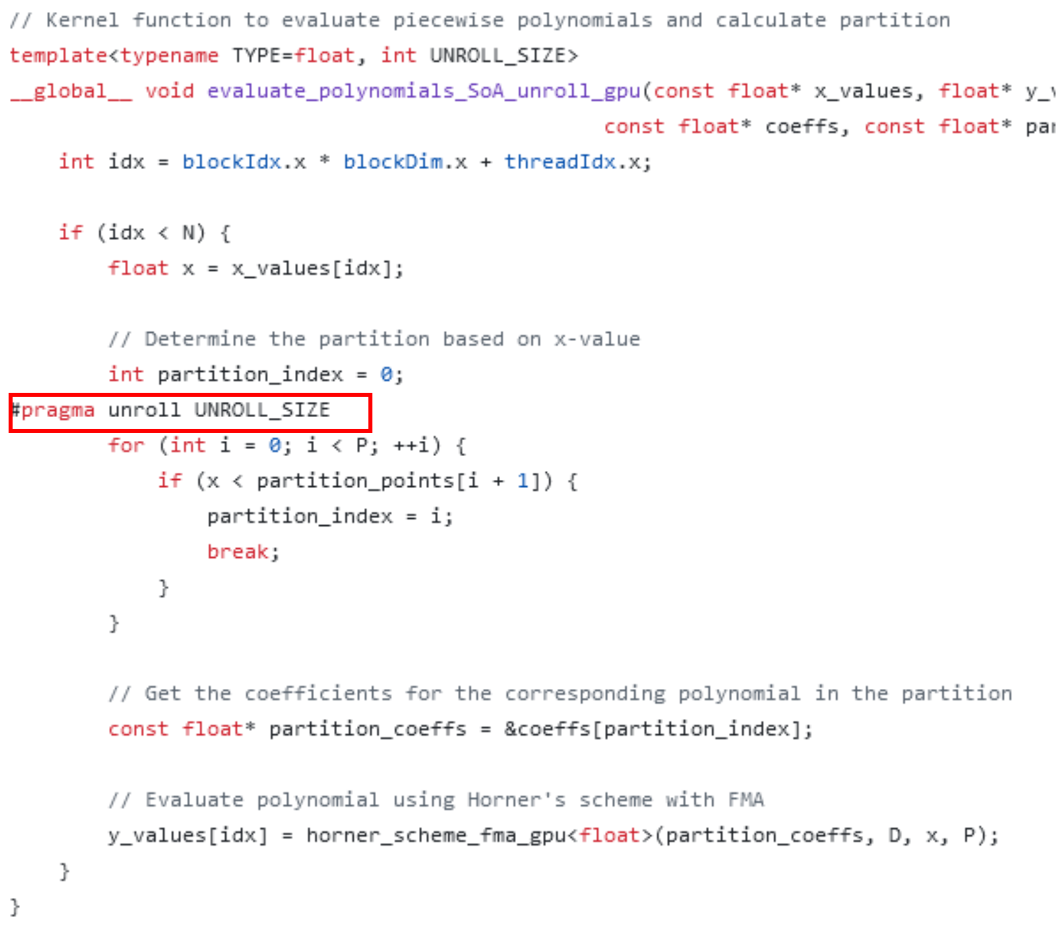
```
// Kernel function to evaluate piecewise polynomials and calculate partition
template<typename TYPE=float>
__global__ void evaluate_polynomials_SoA_gpu(const float* x_values, float* y_values,
                                             const float* coeffs, const float* pa

    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < N) {
        float x = x_values[idx];

        // Determine the partition based on x-value
        int partition_index = 0;
        for (int i = 0; i < P; ++i) {
            if (x < partition_points[i + 1]) {
                partition_index = i;
                break;
            }
        }

        // Get the coefficients for the corresponding polynomial in the partition
        const float* partition_coeffs = &coeffs[partition_index];

        // Evaluate polynomial using Horner's scheme with FMA
        y_values[idx] = horner_scheme_fma_gpu<float>(partition_coeffs, D, x, P);
    }
}
```

https://github.com/SangioAI/torchPACE/blob/main/extra/test_optimizations.cu

# PwPA – Unrolling



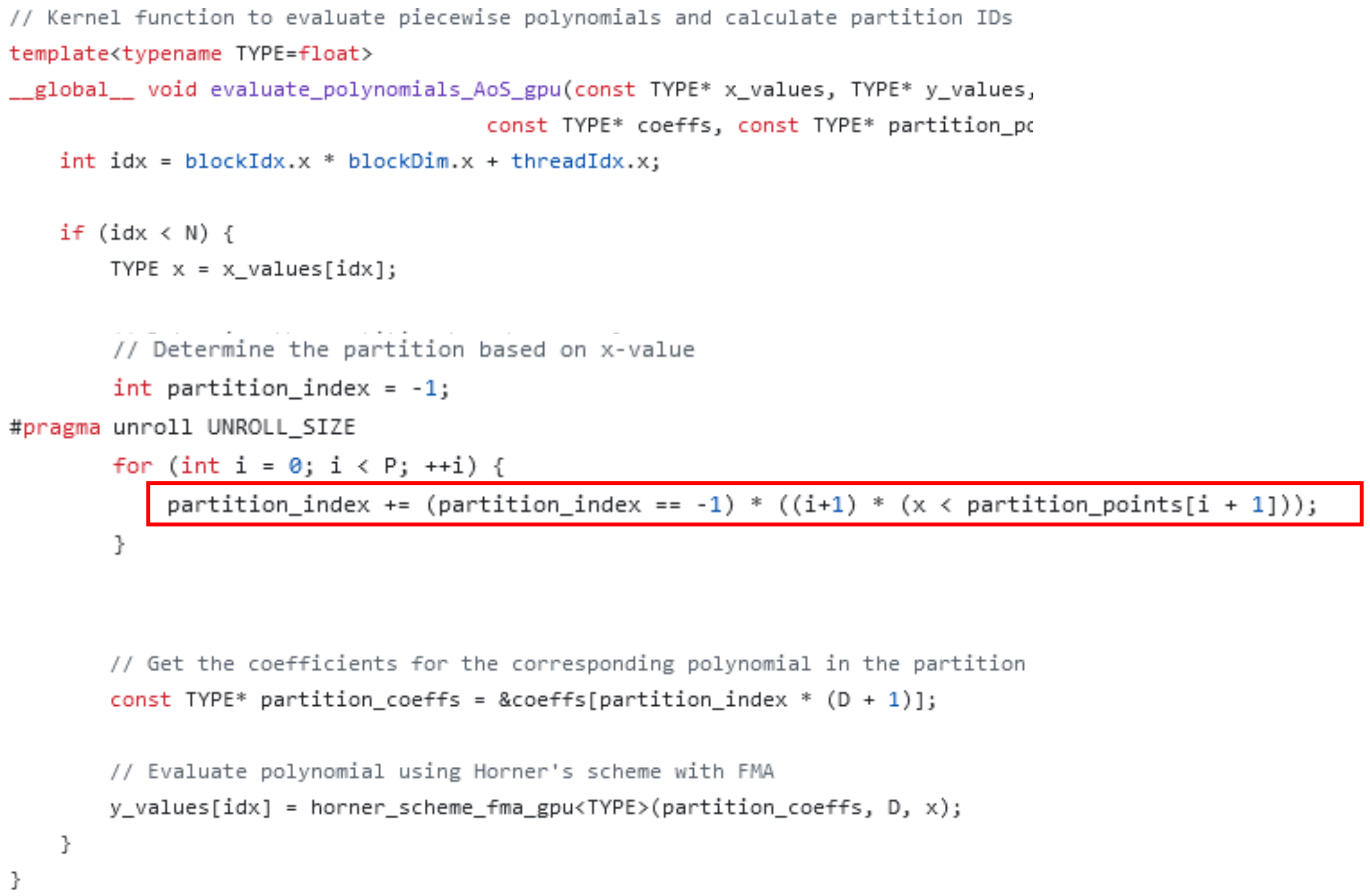
```
// Kernel function to evaluate piecewise polynomials and calculate partition IDs
template<typename TYPE=float, int UNROLL_SIZE>
__global__ void evaluate_polynomials_AoS_unroll_gpu(const float* x_values, float* y
                                                    const float* coeffs, const float* p
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < N) {
        float x = x_values[idx];

        // Determine the partition based on x-value
        int partition_index = 0;
#pragma unroll UNROLL_SIZE
        for (int i = 0; i < P; ++i) {
            if (x < partition_points[i + 1]) {
                partition_index = i;
                break;
            }
        }

        // Get the coefficients for the corresponding polynomial in the partition
        const float* partition_coeffs = &coeffs[partition_index * (D + 1)];

        // Evaluate polynomial using Horner's scheme with FMA
        y_values[idx] = horner_scheme_fma_gpu<float>(partition_coeffs, D, x);
    }
}
```



```
// Kernel function to evaluate piecewise polynomials and calculate partition
template<typename TYPE=float, int UNROLL_SIZE>
__global__ void evaluate_polynomials_SoA_unroll_gpu(const float* x_values, float* y_
                                                    const float* coeffs, const float* pa
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < N) {
        float x = x_values[idx];

        // Determine the partition based on x-value
        int partition_index = 0;
#pragma unroll UNROLL_SIZE
        for (int i = 0; i < P; ++i) {
            if (x < partition_points[i + 1]) {
                partition_index = i;
                break;
            }
        }

        // Get the coefficients for the corresponding polynomial in the partition
        const float* partition_coeffs = &coeffs[partition_index];

        // Evaluate polynomial using Horner's scheme with FMA
        y_values[idx] = horner_scheme_fma_gpu<float>(partition_coeffs, D, x, P);
    }
}
```

https://github.com/SangioAI/torchPACE/blob/main/extra/test_optimizations.cu

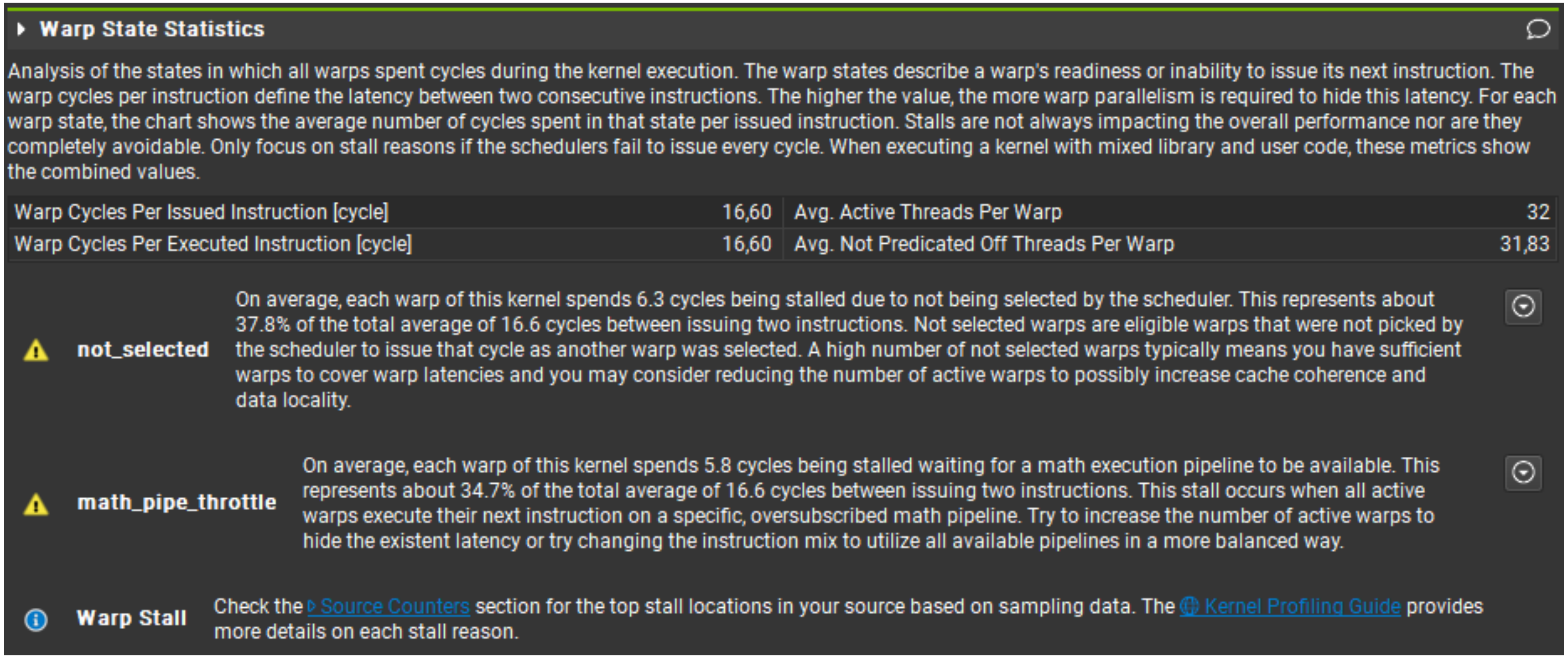
# PwPA – Non-Divergent if-branches



```
// Kernel function to evaluate piecewise polynomials and calculate partition IDs
template<typename TYPE=float>
__global__ void evaluate_polynomials_AoS_gpu(const TYPE* x_values, TYPE* y_values,
                                             const TYPE* coeffs, const TYPE* partition_pc
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < N) {
        TYPE x = x_values[idx];

        // Determine the partition based on x-value
        int partition_index = -1;
#pragma unroll UNROLL_SIZE
        for (int i = 0; i < P; ++i) {
            partition_index += (partition_index == -1) * ((i+1) * (x < partition_points[i + 1]));
        }


        // Get the coefficients for the corresponding polynomial in the partition
        const TYPE* partition_coeffs = &coeffs[partition_index * (D + 1)];

        // Evaluate polynomial using Horner's scheme with FMA
        y_values[idx] = horner_scheme_fma_gpu<TYPE>(partition_coeffs, D, x);
    }
}
```

https://github.com/SangioAI/torchPACE/blob/main/extra/test_optimizations.cu

# PwPA – SoA Non-divergent kernel analysis



**Warp State Statistics**

Analysis of the states in which all warps spent cycles during the kernel execution. The warp states describe a warp's readiness or inability to issue its next instruction. The warp cycles per instruction define the latency between two consecutive instructions. The higher the value, the more warp parallelism is required to hide this latency. For each warp state, the chart shows the average number of cycles spent in that state per issued instruction. Stalls are not always impacting the overall performance nor are they completely avoidable. Only focus on stall reasons if the schedulers fail to issue every cycle. When executing a kernel with mixed library and user code, these metrics show the combined values.

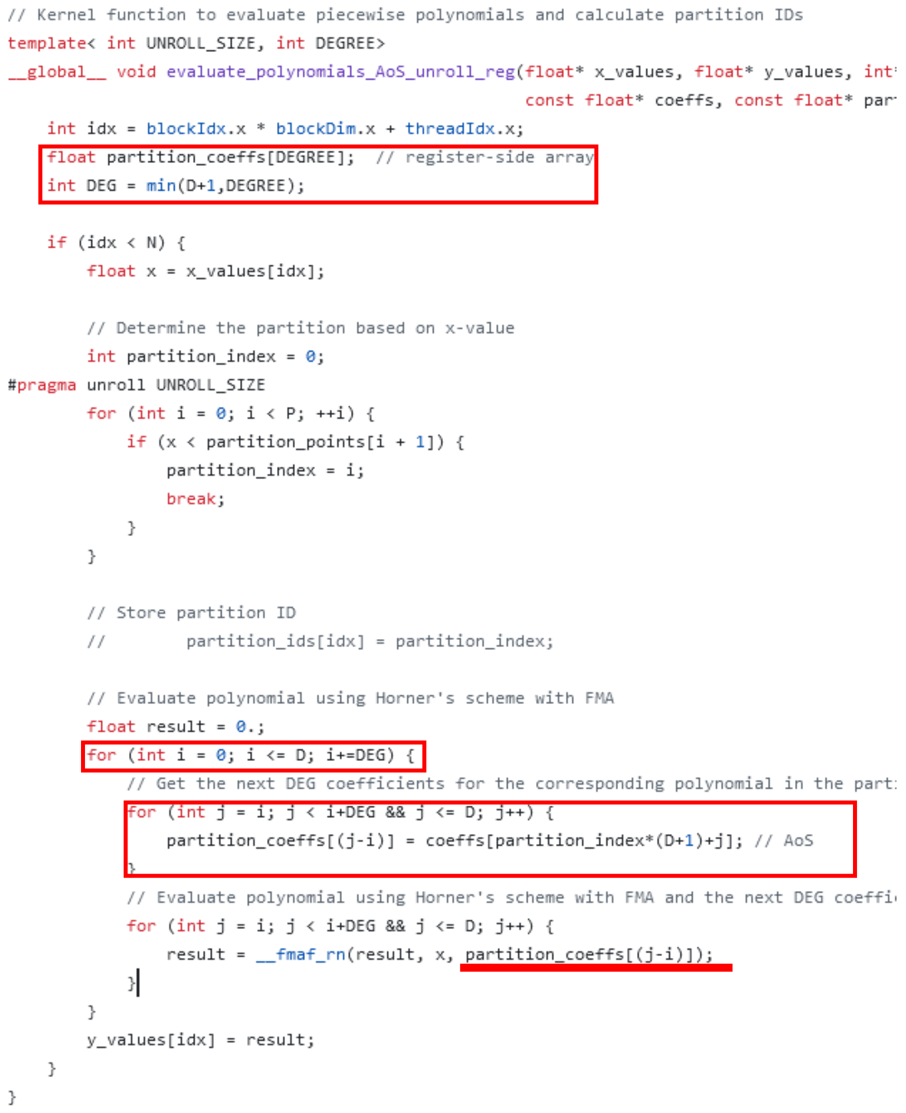
| | | | |
|---|---|---|---|
| Warp Cycles Per Issued Instruction [cycle] | 16,60 | Avg. Active Threads Per Warp | 32 |
| Warp Cycles Per Executed Instruction [cycle] | 16,60 | Avg. Not Predicated Off Threads Per Warp | 31,83 |

**not_selected** On average, each warp of this kernel spends 6.3 cycles being stalled due to not being selected by the scheduler. This represents about 37.8% of the total average of 16.6 cycles between issuing two instructions. Not selected warps are eligible warps that were not picked by the scheduler to issue that cycle as another warp was selected. A high number of not selected warps typically means you have sufficient warps to cover warp latencies and you may consider reducing the number of active warps to possibly increase cache coherence and data locality.

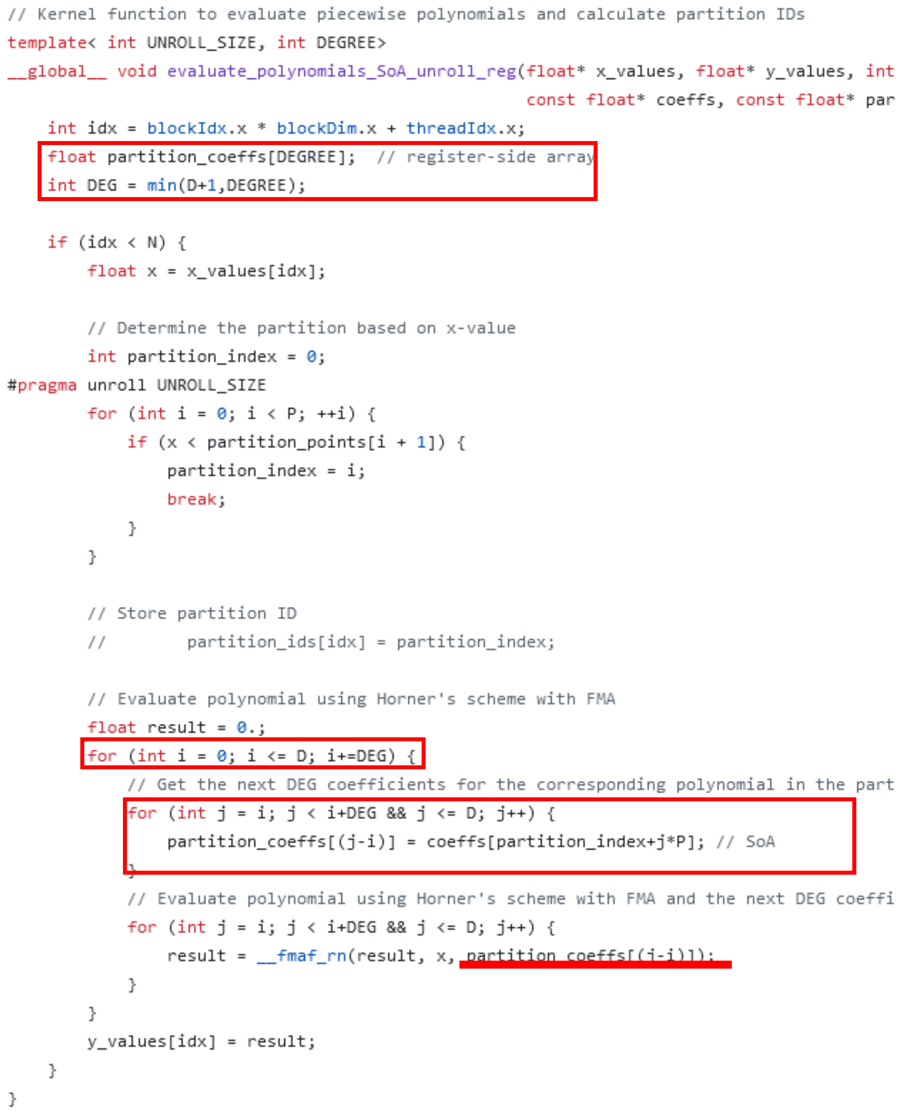
**math_pipe_throttle** On average, each warp of this kernel spends 5.8 cycles being stalled waiting for a math execution pipeline to be available. This represents about 34.7% of the total average of 16.6 cycles between issuing two instructions. This stall occurs when all active warps execute their next instruction on a specific, oversubscribed math pipeline. Try to increase the number of active warps to hide the existent latency or try changing the instruction mix to utilize all available pipelines in a more balanced way.

**Warp Stall** Check the Source Counters section for the top stall locations in your source based on sampling data. The Kernel Profiling Guide provides more details on each stall reason.

# PwPA – Data Reuse w/ Registers



```
// Kernel function to evaluate piecewise polynomials and calculate partition IDs
template< int UNROLL_SIZE, int DEGREE>
__global__ void evaluate_polynomials_AoS_unroll_reg(float* x_values, float* y_values, int
                                                     const float* coeffs, const float* par
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    float partition_coeffs[DEGREE];  // register-side array
    int DEG = min(D+1,DEGREE);

    if (idx < N) {
        float x = x_values[idx];

        // Determine the partition based on x-value
        int partition_index = 0;
#pragma unroll UNROLL_SIZE
        for (int i = 0; i < P; ++i) {
            if (x < partition_points[i + 1]) {
                partition_index = i;
                break;
            }
        }

        // Store partition ID
        //         partition_ids[idx] = partition_index;

        // Evaluate polynomial using Horner's scheme with FMA
        float result = 0.;
        for (int i = 0; i <= D; i+=DEG) {
            // Get the next DEG coefficients for the corresponding polynomial in the part
            for (int j = i; j < i+DEG && j <= D; j++) {
                partition_coeffs[(j-i)] = coeffs[partition_index*(D+1)+j]; // AoS
            }
            // Evaluate polynomial using Horner's scheme with FMA and the next DEG coeffi
            for (int j = i; j < i+DEG && j <= D; j++) {
                result = __fmaf_rn(result, x, partition_coeffs[(j-i)]);
            }
        }
        y_values[idx] = result;
    }
}
```

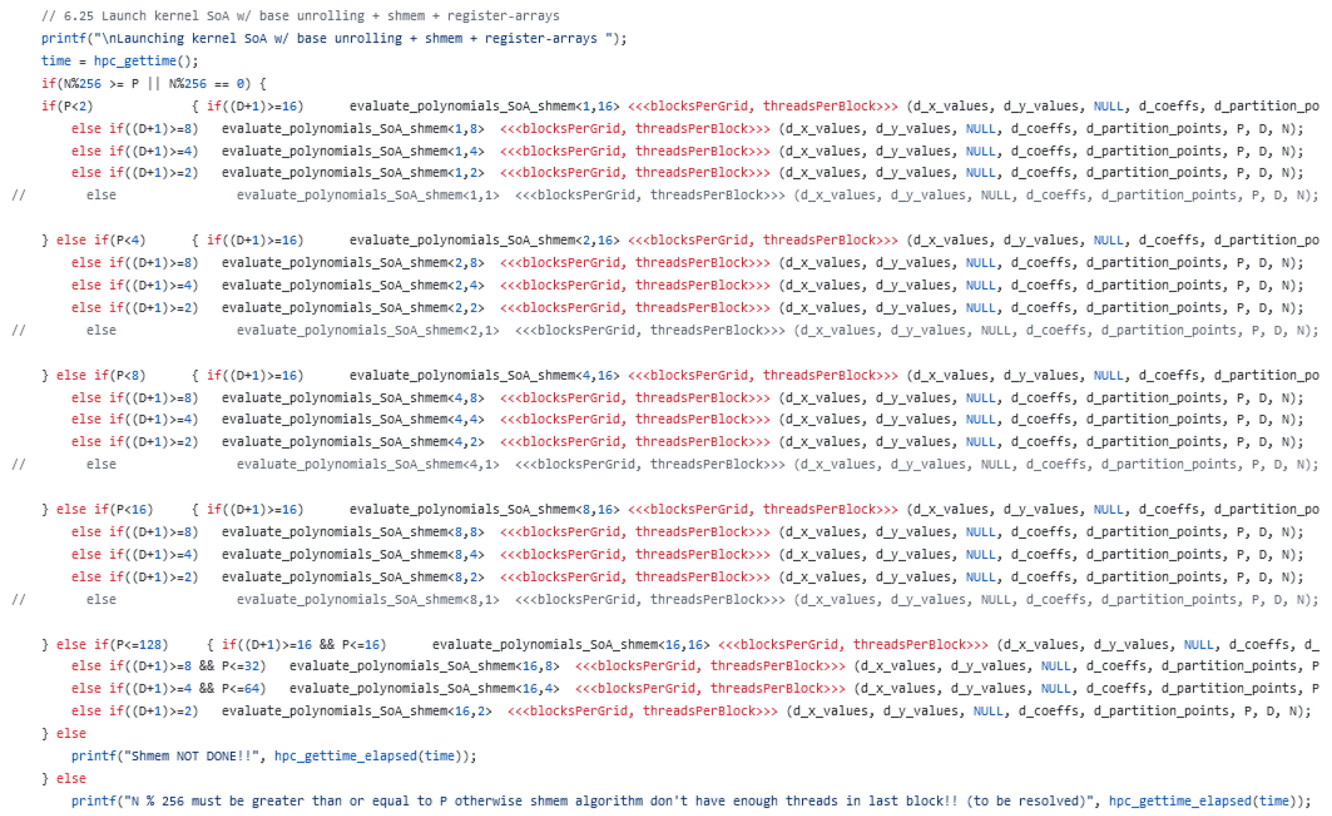


```
// Kernel function to evaluate piecewise polynomials and calculate partition IDs
template< int UNROLL_SIZE, int DEGREE>
__global__ void evaluate_polynomials_SoA_unroll_reg(float* x_values, float* y_values, int
                                                     const float* coeffs, const float* par
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    float partition_coeffs[DEGREE];  // register-side array
    int DEG = min(D+1,DEGREE);

    if (idx < N) {
        float x = x_values[idx];

        // Determine the partition based on x-value
        int partition_index = 0;
#pragma unroll UNROLL_SIZE
        for (int i = 0; i < P; ++i) {
            if (x < partition_points[i + 1]) {
                partition_index = i;
                break;
            }
        }

        // Store partition ID
        //         partition_ids[idx] = partition_index;

        // Evaluate polynomial using Horner's scheme with FMA
        float result = 0.;
        for (int i = 0; i <= D; i+=DEG) {
            // Get the next DEG coefficients for the corresponding polynomial in the part
            for (int j = i; j < i+DEG && j <= D; j++) {
                partition_coeffs[(j-i)] = coeffs[partition_index+j*P]; // SoA
            }
            // Evaluate polynomial using Horner's scheme with FMA and the next DEG coeffi
            for (int j = i; j < i+DEG && j <= D; j++) {
                result = __fmaf_rn(result, x, partition_coeffs[(j-i)]);
            }
        }
        y_values[idx] = result;
    }
}
```

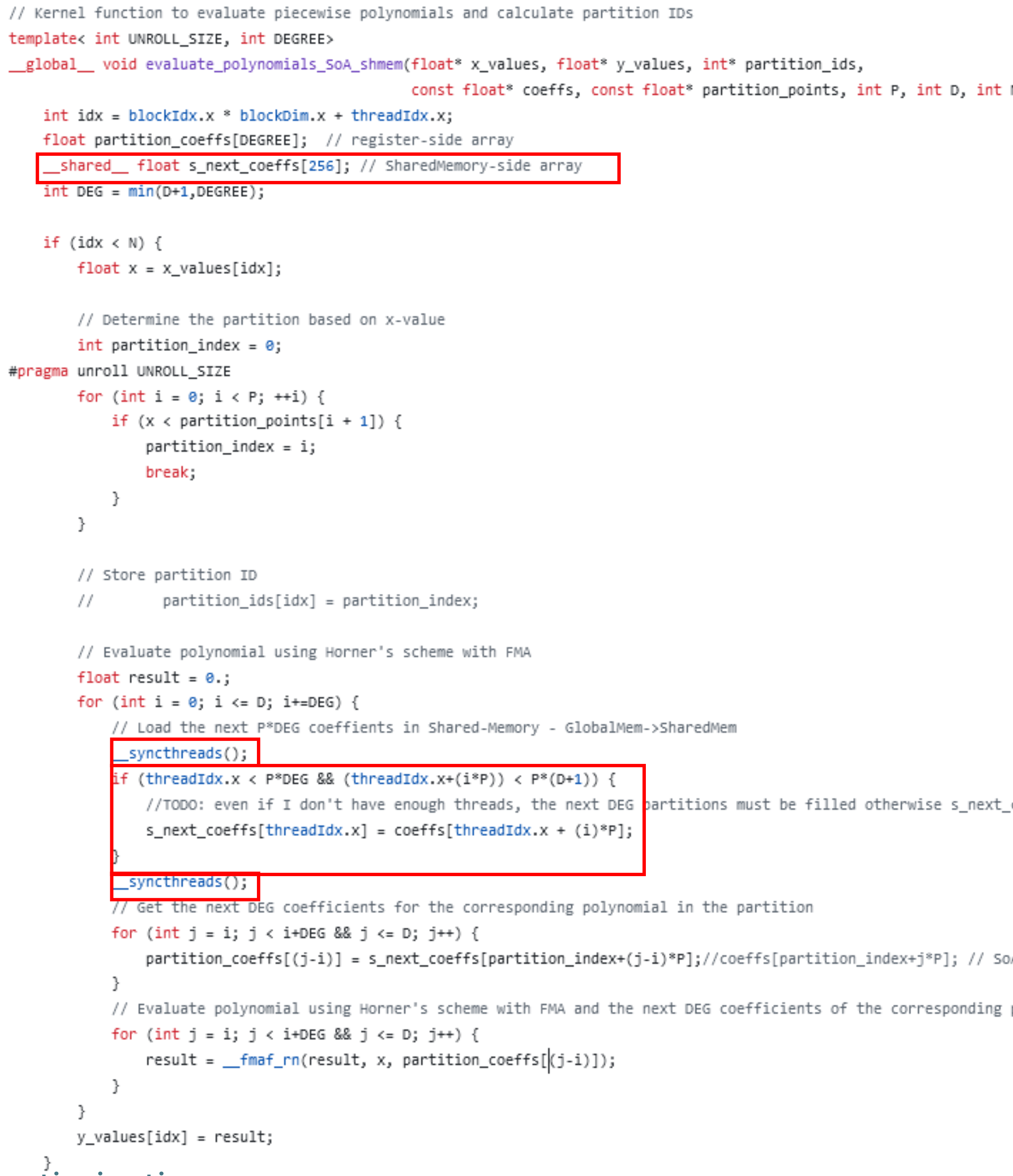
https://github.com/SangioAI/torchPACE/blob/main/extra/test_optimizations.cu

# PwPA – Shared Memory

Kernel calls in Host code:



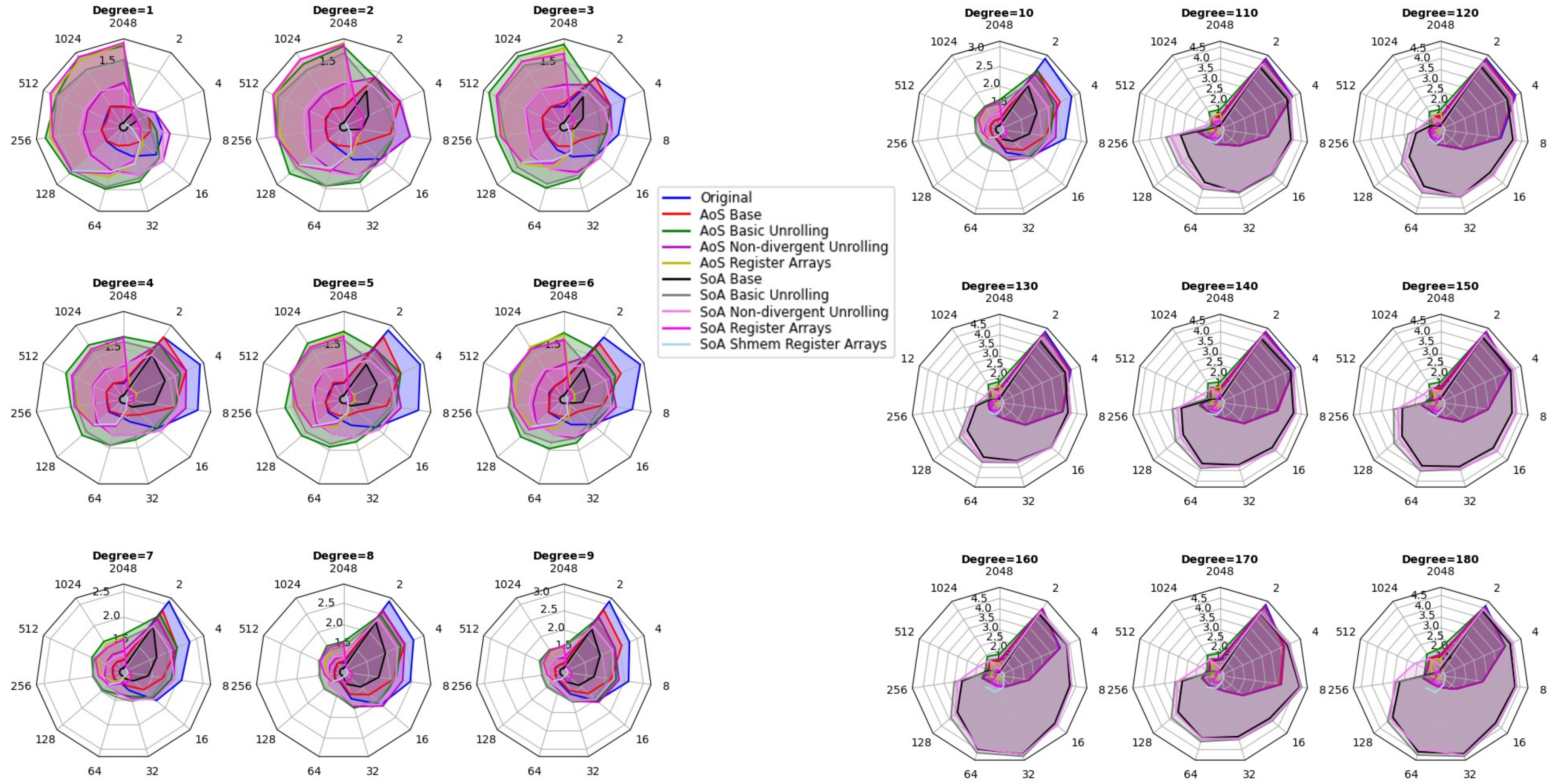
```
    // 6.25 Launch kernel SoA w/ base unrolling + shmem + register-arrays
    printf("\nLaunching kernel SoA w/ base unrolling + shmem + register-arrays ");
    time = hpc_gettime();
    if(N%256 >= P || N%256 == 0) {
    if(P<2)            { if((D+1)>=16)      evaluate_polynomials_SoA_shmem<1,16> <<<blocksPerGrid, threadsPerBlock>>> (d_x_values, d_y_values, NULL, d_coeffs, d_partition_po
        else if((D+1)>=8)   evaluate_polynomials_SoA_shmem<1,8>  <<<blocksPerGrid, threadsPerBlock>>> (d_x_values, d_y_values, NULL, d_coeffs, d_partition_points, P, D, N);
        else if((D+1)>=4)   evaluate_polynomials_SoA_shmem<1,4>  <<<blocksPerGrid, threadsPerBlock>>> (d_x_values, d_y_values, NULL, d_coeffs, d_partition_points, P, D, N);
        else if((D+1)>=2)   evaluate_polynomials_SoA_shmem<1,2>  <<<blocksPerGrid, threadsPerBlock>>> (d_x_values, d_y_values, NULL, d_coeffs, d_partition_points, P, D, N);
//        else                 evaluate_polynomials_SoA_shmem<1,1>  <<<blocksPerGrid, threadsPerBlock>>> (d_x_values, d_y_values, NULL, d_coeffs, d_partition_points, P, D, N);

    } else if(P<4)     { if((D+1)>=16)      evaluate_polynomials_SoA_shmem<2,16> <<<blocksPerGrid, threadsPerBlock>>> (d_x_values, d_y_values, NULL, d_coeffs, d_partition_po
        else if((D+1)>=8)   evaluate_polynomials_SoA_shmem<2,8>  <<<blocksPerGrid, threadsPerBlock>>> (d_x_values, d_y_values, NULL, d_coeffs, d_partition_points, P, D, N);
        else if((D+1)>=4)   evaluate_polynomials_SoA_shmem<2,4>  <<<blocksPerGrid, threadsPerBlock>>> (d_x_values, d_y_values, NULL, d_coeffs, d_partition_points, P, D, N);
        else if((D+1)>=2)   evaluate_polynomials_SoA_shmem<2,2>  <<<blocksPerGrid, threadsPerBlock>>> (d_x_values, d_y_values, NULL, d_coeffs, d_partition_points, P, D, N);
//        else                 evaluate_polynomials_SoA_shmem<2,1>  <<<blocksPerGrid, threadsPerBlock>>> (d_x_values, d_y_values, NULL, d_coeffs, d_partition_points, P, D, N);

    } else if(P<8)     { if((D+1)>=16)      evaluate_polynomials_SoA_shmem<4,16> <<<blocksPerGrid, threadsPerBlock>>> (d_x_values, d_y_values, NULL, d_coeffs, d_partition_po
        else if((D+1)>=8)   evaluate_polynomials_SoA_shmem<4,8>  <<<blocksPerGrid, threadsPerBlock>>> (d_x_values, d_y_values, NULL, d_coeffs, d_partition_points, P, D, N);
        else if((D+1)>=4)   evaluate_polynomials_SoA_shmem<4,4>  <<<blocksPerGrid, threadsPerBlock>>> (d_x_values, d_y_values, NULL, d_coeffs, d_partition_points, P, D, N);
        else if((D+1)>=2)   evaluate_polynomials_SoA_shmem<4,2>  <<<blocksPerGrid, threadsPerBlock>>> (d_x_values, d_y_values, NULL, d_coeffs, d_partition_points, P, D, N);
//        else                 evaluate_polynomials_SoA_shmem<4,1>  <<<blocksPerGrid, threadsPerBlock>>> (d_x_values, d_y_values, NULL, d_coeffs, d_partition_points, P, D, N);

    } else if(P<16)    { if((D+1)>=16)      evaluate_polynomials_SoA_shmem<8,16> <<<blocksPerGrid, threadsPerBlock>>> (d_x_values, d_y_values, NULL, d_coeffs, d_partition_po
        else if((D+1)>=8)   evaluate_polynomials_SoA_shmem<8,8>  <<<blocksPerGrid, threadsPerBlock>>> (d_x_values, d_y_values, NULL, d_coeffs, d_partition_points, P, D, N);
        else if((D+1)>=4)   evaluate_polynomials_SoA_shmem<8,4>  <<<blocksPerGrid, threadsPerBlock>>> (d_x_values, d_y_values, NULL, d_coeffs, d_partition_points, P, D, N);
        else if((D+1)>=2)   evaluate_polynomials_SoA_shmem<8,2>  <<<blocksPerGrid, threadsPerBlock>>> (d_x_values, d_y_values, NULL, d_coeffs, d_partition_points, P, D, N);
//        else                 evaluate_polynomials_SoA_shmem<8,1>  <<<blocksPerGrid, threadsPerBlock>>> (d_x_values, d_y_values, NULL, d_coeffs, d_partition_points, P, D, N);

    } else if(P<=128)    { if((D+1)>=16 && P<=16)      evaluate_polynomials_SoA_shmem<16,16> <<<blocksPerGrid, threadsPerBlock>>> (d_x_values, d_y_values, NULL, d_coeffs, d_
        else if((D+1)>=8 && P<=32)   evaluate_polynomials_SoA_shmem<16,8>  <<<blocksPerGrid, threadsPerBlock>>> (d_x_values, d_y_values, NULL, d_coeffs, d_partition_points, P
        else if((D+1)>=4 && P<=64)   evaluate_polynomials_SoA_shmem<16,4>  <<<blocksPerGrid, threadsPerBlock>>> (d_x_values, d_y_values, NULL, d_coeffs, d_partition_points, P
        else if((D+1)>=2)   evaluate_polynomials_SoA_shmem<16,2>  <<<blocksPerGrid, threadsPerBlock>>> (d_x_values, d_y_values, NULL, d_coeffs, d_partition_points, P, D, N);
    } else
        printf("Shmem NOT DONE!!", hpc_gettime_elapsed(time));
    } else
        printf("N % 256 must be greater than or equal to P otherwise shmem algorithm don't have enough threads in last block!! (to be resolved)", hpc_gettime_elapsed(time));
```



```
// Kernel function to evaluate piecewise polynomials and calculate partition IDs
template< int UNROLL_SIZE, int DEGREE>
__global__ void evaluate_polynomials_SoA_shmem(float* x_values, float* y_values, int* partition_ids,
                                               const float* coeffs, const float* partition_points, int P, int D, int

    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    float partition_coeffs[DEGREE];  // register-side array
    __shared__ float s_next_coeffs[256]; // SharedMemory-side array
    int DEG = min(D+1,DEGREE);

    if (idx < N) {
        float x = x_values[idx];

        // Determine the partition based on x-value
        int partition_index = 0;
#pragma unroll UNROLL_SIZE
        for (int i = 0; i < P; ++i) {
            if (x < partition_points[i + 1]) {
                partition_index = i;
                break;
            }
        }

        // Store partition ID
        //          partition_ids[idx] = partition_index;

        // Evaluate polynomial using Horner's scheme with FMA
        float result = 0.;
        for (int i = 0; i <= D; i+=DEG) {
            // Load the next P*DEG coeffients in Shared-Memory - GlobalMem->SharedMem
            __syncthreads();
            if (threadIdx.x < P*DEG && (threadIdx.x+(i*P)) < P*(D+1)) {
                //TODO: even if I don't have enough threads, the next DEG partitions must be filled otherwise s_next_
                s_next_coeffs[threadIdx.x] = coeffs[threadIdx.x + (i)*P];
            }
            __syncthreads();
            // Get the next DEG coefficients for the corresponding polynomial in the partition
            for (int j = i; j < i+DEG && j <= D; j++) {
                partition_coeffs[(j-i)] = s_next_coeffs[partition_index+(j-i)*P];//coeffs[partition_index+j*P]; // So
            }
            // Evaluate polynomial using Horner's scheme with FMA and the next DEG coefficients of the corresponding 
            for (int j = i; j < i+DEG && j <= D; j++) {
                result = __fmaf_rn(result, x, partition_coeffs[(j-i)]);
            }
        }
        y_values[idx] = result;
    }
```

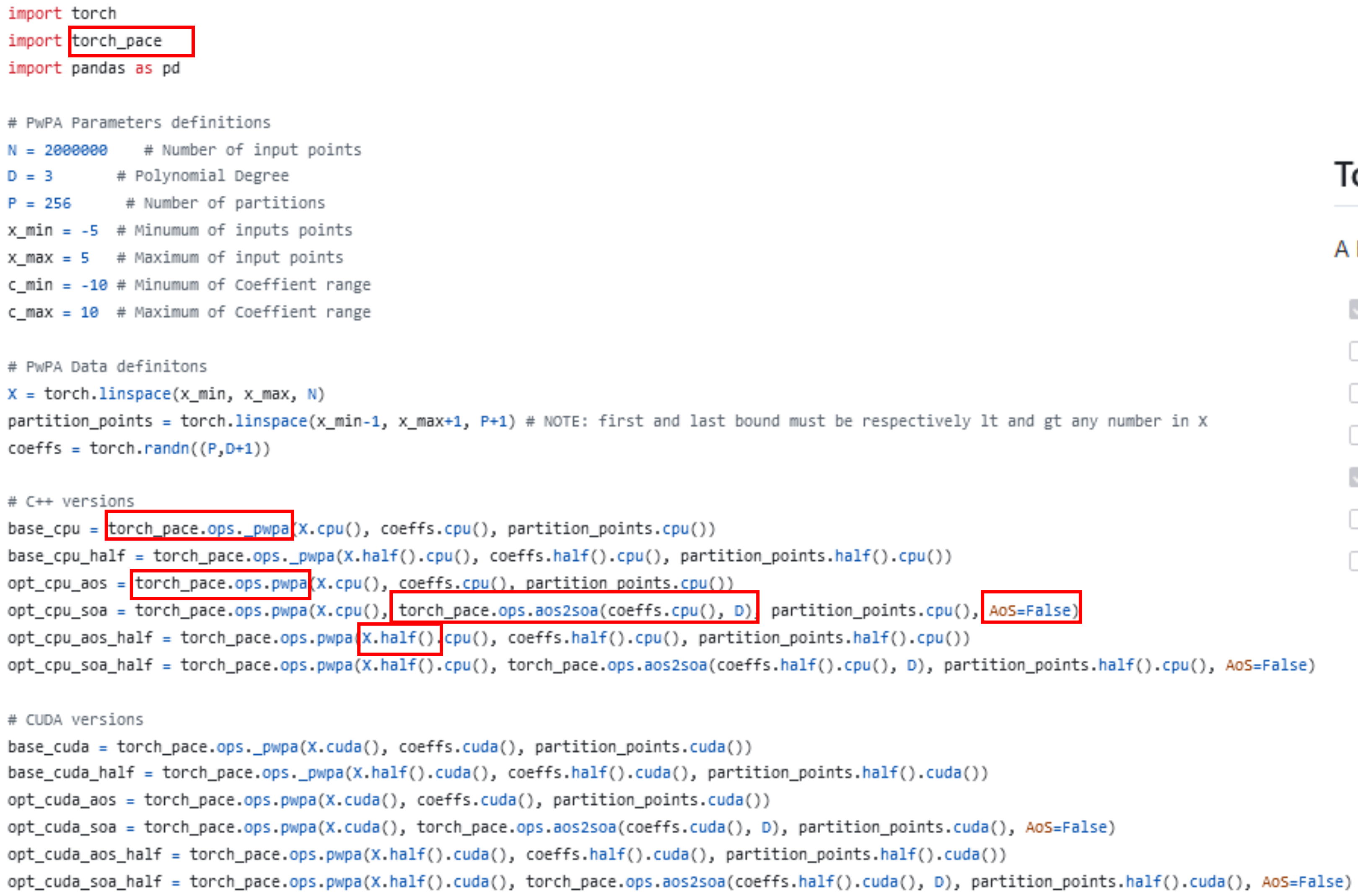
https://github.com/SangioAI/torchPACE/blob/main/extra/test_optimizations.cu

# PwPA – Speedup Results

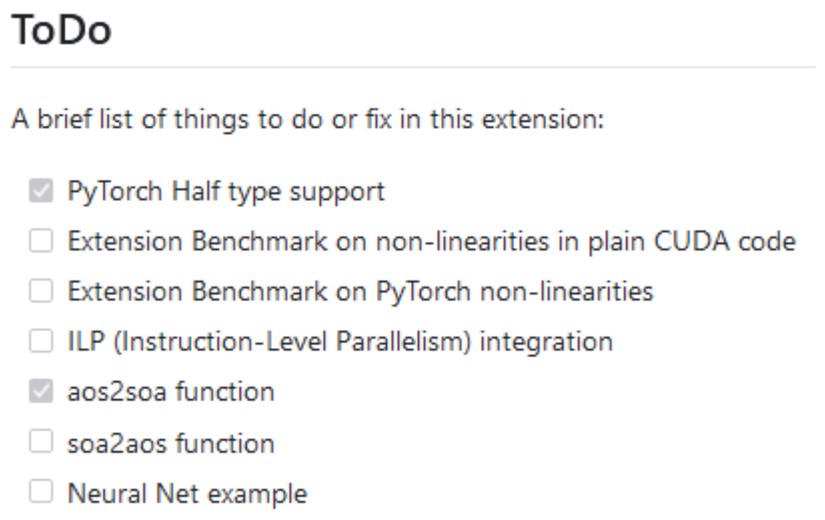


https://github.com/SangioAI/torchPACE/blob/main/extra/README.md

# torchPACE – PyTorch PwPA extension



```python
import torch
import torch_pace
import pandas as pd

# PwPA Parameters definitions
N = 2000000     # Number of input points
D = 3       # Polynomial Degree
P = 256      # Number of partitions
x_min = -5  # Minumum of inputs points
x_max = 5   # Maximum of input points
c_min = -10 # Minumum of Coeffient range
c_max = 10  # Maximum of Coeffient range

# PwPA Data definitons
X = torch.linspace(x_min, x_max, N)
partition_points = torch.linspace(x_min-1, x_max+1, P+1) # NOTE: first and last bound must be respectively lt and gt any number in X
coeffs = torch.randn((P,D+1))

# C++ versions
base_cpu = torch_pace.ops._pwpa(X.cpu(), coeffs.cpu(), partition_points.cpu())
base_cpu_half = torch_pace.ops._pwpa(X.half().cpu(), coeffs.half().cpu(), partition_points.half().cpu())
opt_cpu_aos = torch_pace.ops.pwpa(X.cpu(), coeffs.cpu(), partition_points.cpu())
opt_cpu_soa = torch_pace.ops.pwpa(X.cpu(), torch_pace.ops.aos2soa(coeffs.cpu(), D), partition_points.cpu(), AoS=False)
opt_cpu_aos_half = torch_pace.ops.pwpa(X.half().cpu(), coeffs.half().cpu(), partition_points.half().cpu())
opt_cpu_soa_half = torch_pace.ops.pwpa(X.half().cpu(), torch_pace.ops.aos2soa(coeffs.half().cpu(), D), partition_points.half().cpu(), AoS=False)

# CUDA versions
base_cuda = torch_pace.ops._pwpa(X.cuda(), coeffs.cuda(), partition_points.cuda())
base_cuda_half = torch_pace.ops._pwpa(X.half().cuda(), coeffs.half().cuda(), partition_points.half().cuda())
opt_cuda_aos = torch_pace.ops.pwpa(X.cuda(), coeffs.cuda(), partition_points.cuda())
opt_cuda_soa = torch_pace.ops.pwpa(X.cuda(), torch_pace.ops.aos2soa(coeffs.cuda(), D), partition_points.cuda(), AoS=False)
opt_cuda_aos_half = torch_pace.ops.pwpa(X.half().cuda(), coeffs.half().cuda(), partition_points.half().cuda())
opt_cuda_soa_half = torch_pace.ops.pwpa(X.half().cuda(), torch_pace.ops.aos2soa(coeffs.half().cuda(), D), partition_points.half().cuda(), AoS=False)
```



## ToDo

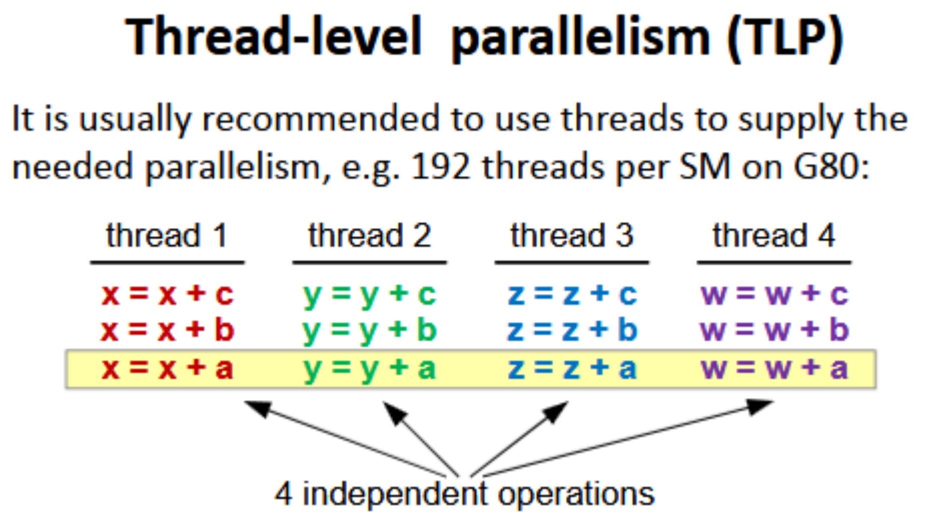
A brief list of things to do or fix in this extension:

- [x] PyTorch Half type support
- [ ] Extension Benchmark on non-linearities in plain CUDA code
- [ ] Extension Benchmark on PyTorch non-linearities
- [ ] ILP (Instruction-Level Parallelism) integration
- [x] aos2soa function
- [ ] soa2aos function
- [ ] Neural Net example

https://github.com/SangioAI/torchPACE

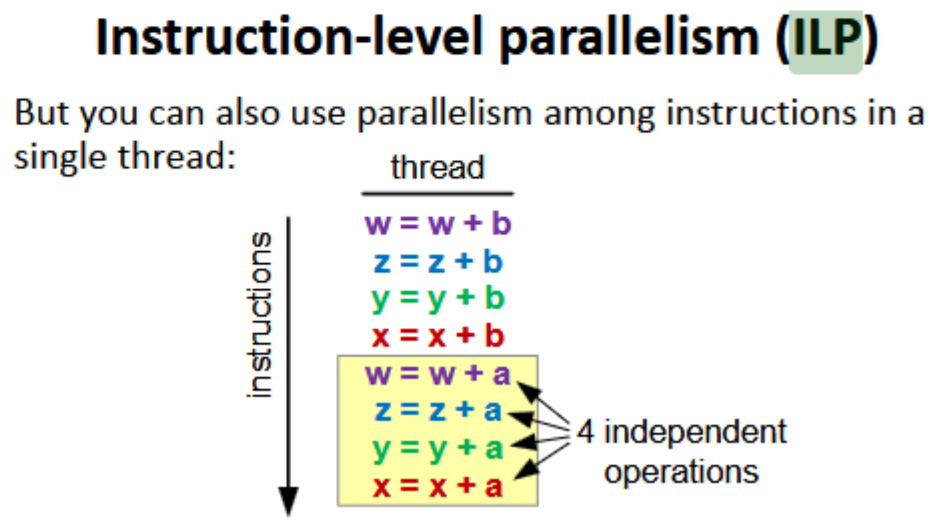
# TODO: Instruction-Level Parallelism

https://www.nvidia.com/content/GTC-2010/pdfs/2238_GTC2010.pdf



**Thread-level parallelism (TLP)**

It is usually recommended to use threads to supply the needed parallelism, e.g. 192 threads per SM on G80:



**Instruction-level parallelism (ILP)**

But you can also use parallelism among instructions in a single thread:

Thank you for the attention.