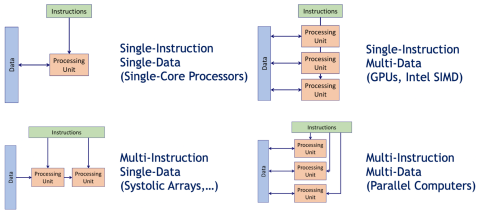
Instruction Type (SIMD = vector dot prod)

		Data Stream	
		Single	Multi
Instruction Stream	Single	SISD (Single-Core Processors)	SIMD (GPUs, Intel SSE/AVX extensions, ...)
	Multi	MISD (Systolic Arrays, ...)	MIMD (VLIW, Parallel Computers)



Intel AVX 512 registers

XMM0-15 (128), YMM0-15 (256), ZMM0-31 (256)
Does 32x 8 bits operations in one instruction

Instruction types

- **Pointwise:** saxpy (ax+y), add, scale
- **Reduction:** sum, average, min, max

```
#pragma opm parallel for reduction([op]:[var])
```

Performance bound types

Which takes longer, I/O or the algorithm itself
- **Memory:** larger complexity for I/O
- **Compute:** larger complexity for the algorithm itself

Laplacian matrix

Treats convolution as a (sparse) matrix multiplication

Conjugate gradient operation

Solves for **Ax=b** given a solution exists, allows systems of linear equation to be solved at once (**r = b-Ax**)
- **Pros:** more compact and cost effective for dense **A**
- **Cons:** space inefficient if **A** is sparse

Sparse Operations

I.e. saxpy `y[offset[i]] += a*x[offset[i]]`
`offset` can be stored in blocks `blockOffset`

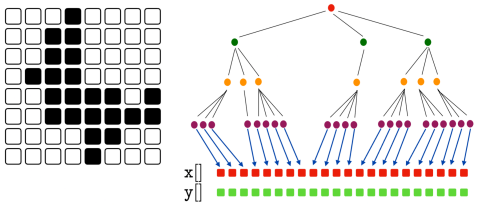
```
#pragma omp parallel for
    for (int b = 0; b < blockOffsets.size(); b++) {
        _mm_prefetch ( &x[blockOffsets[b+L2_PREFETCH_DISTANCE]], _MM_HINT_T2 );
        _mm_prefetch ( &x[blockOffsets[b+L1_PREFETCH_DISTANCE]], _MM_HINT_T1 );
        _mm_prefetch ( &y[blockOffsets[b+L2_PREFETCH_DISTANCE]], _MM_HINT_T2 );
        _mm_prefetch ( &y[blockOffsets[b+L1_PREFETCH_DISTANCE]], _MM_HINT_T1 );
#pragma omp simd
        for (int i = 0; i < BLOCK_SIZE; i++)
            y[blockOffsets[b]+i] += scale * x[blockOffsets[b]+i];
```

Struct of array stores `x,y,z` separately
- **Pros:** allows SIMD, better locality per channel, easier to resize one of the channels
- **Cons:** less compatibility with SIMD for sparse, uses multiple localities (one per channel), concerns with vector type (i.e. coordinates), TLB pressure

Array of struct `arr[0]=(x[0],y[0],z[0])` allows saxpy to be done on offset directly
- **Pros:** more natural access, better locality if all channels are accessed, exploits SIMD if sparse
- **Cons:** waste access cost if one channel is accessed, alignment issue if not 2^n, SIMD problem if not sparse

Arrays of structs of arrays (hybrid of the two)

Sparse Grid stored as 2^K-KD Tree



Problem: neighboring data can be from different tree

CSR (Compressed Sparse Row) Matrix

$$\mathbf{A} = \begin{pmatrix} 10 & 11 & & 12 \\ & 13 & 14 & \\ 15 & & 16 & 17 \\ & 18 & & 19 \end{pmatrix}$$

```
int offsets[] = { 0, 3, 5, 8, 10}
int col[]     = { 0, 1, 3, 1, 2, 0, 2, 3, 1, 3}
int value[] = {10,11,12,13,14,15,16,17,18,19}
```

`len(offsets) = n+1` starts with `0` and ends with `len(value)`.
- **Pros:** very space efficient for sparse matrices
- **Cons:** does nothing if **A** is full/dense

```
for (int i = 0; i < N; i++)
    y[i] = 0.;
    for (int k = rowOffsets[i]; k < rowOffsets[i+1]; k++)
        const int j = columnIndices[k];
        y[i] += values[k] * x[j];
```

`mkl_cspblas_scsrgemv` multiplies CSR matrix with vector

GEMM operation

Does **matrix-matrix multiplication** (C=AB) as blocks
Aligns the entries since matrix is **ROW MAJOR**
Transposes B to make it **COLUMN MAJOR**

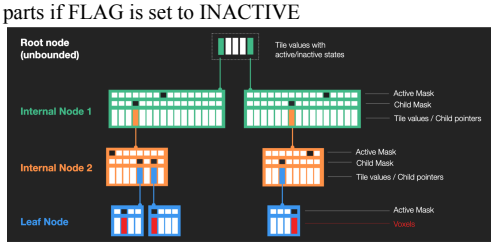
```
for (int ii = 0; ii < BLOCK_SIZE; ii++)
for (int jj = 0; jj < BLOCK_SIZE; jj++)
#pragma omp simd
    for (int kk = 0; kk < BLOCK_SIZE; kk++)
        localC[ii][jj] += localA[ii][kk] * localB[jj][kk];
```

Complexity: O(N^3) but better spatial locality
Small block: better locality, wastes space, data bound
Large block: doesn't fit in cache, compute bound
Use `i,k,j` instead of `i,j,k` if B is not transposed

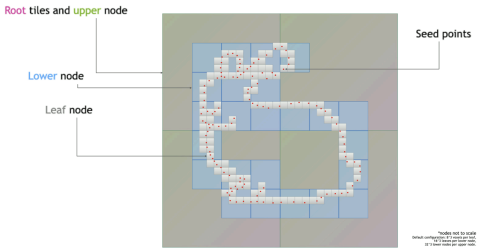
```
for (int ii = 0; ii < BLOCK_SIZE; ii++)
    for (int kk = 0; kk < BLOCK_SIZE; kk++) {
#pragma omp simd
    for (int jj = 0; jj < BLOCK_SIZE; jj++)
        bC[ii][jj] += bA[ii][kk] * bB[kk][jj];
```

OpenVDB/NanoVDB

Stores 8x8x8 block for a leaf node instead of one point
Uses caching structure with easy neighbor access
IndexGrid: tells VDB what the coordinate range is. GridBuilder creates a tree. IndexGrid also stored data
- **Pros:** works better for dense matrices since nearby values are stored in the same block. Also skips sparse parts if FLAG is set to INACTIVE



- **Root:** hash table or linear array (store as needed)
- **Upper (1):** store 32x32x32 lower nodes
- **Lower (2):** store 16x16x16 leaf nodes
- **Leaf node:** stores 512 x (data point +1 bit active flag)



OpenVDB stores tree as pointers, NanoVDB stores the tree in a flattened array (+offset)
`firstPrivate([var])` var is created in all threads but only the first thread's value is copied to master. Copies the first because VDB can alter data entries

Assembly Intrinsics (Built-in ASM)

```
#include "immintrin.h"
```

```
for (int ii = 0; ii < BLOCK_SIZE; ii++)
for (int kk = 0; kk < BLOCK_SIZE; kk++)
    for (int jj = 0; jj < BLOCK_SIZE; jj += nW) {
        __m256 vB = _mm256_load_ps(&bB[kk][jj]);
        __m256 vA = _mm256_set1_ps(bA[ii][kk]);
        __m256 vC = _mm256_load_ps(&bC[ii][jj]);
        vC = _mm256_fmadd_ps(vA, vB, vC);
        _mm256_store_ps(&bC[ii][jj], vC);
    }
```

`nW=16` number of floats to go into SIMD kernel call

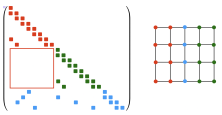
Solving Linear Equations with Triangular L

Lx=b easy to solve if L lower triangular (`cblas_?trsm`)
Forward substitution method, solves x1, x2, …, xm

$$\begin{aligned} \ell_{1,1}x_1 &= b_1 & x_1 &= \frac{b_1}{\ell_{1,1}}, \\ \ell_{2,1}x_1 + \ell_{2,2}x_2 &= b_2 & x_2 &= \frac{b_2 - \ell_{2,1}x_1}{\ell_{2,2}}, \\ \vdots \quad \vdots \quad \ddots \quad \vdots & & \vdots & \\ \ell_{m,1}x_1 + \ell_{m,2}x_2 + \cdots + \ell_{m,m}x_m &= b_m & x_m &= \frac{b_m - \sum_{i=1}^{m-1} \ell_{m,i}x_i}{\ell_{m,m}}. \end{aligned}$$

BLAS level 3 direct solver for AX=B if A is dense and symmetric (saves bandwidth), X and B are matrices. GEMM is a part of it.
LAPACK solve AX = B, preprocesses A into more manageable forms (prefer A is lower triangular matrix or symmetric matrix)
- **Iterative:** easier to set up and no preprocess required, but doesn't guarantee convergence and is less accurate
- **Direct:** preprocesses A, easy to parallel if A is dense, always work, but more expensive up to O(N^3)

PARDISO:direct solver if A is sparse. CSL/CSR format
- 1 reorder A into a favorable form (more diagonal)
- 2 Cholesky decomposition A = LL*, faster if L is more sparse, is very serial (like Gaussian elimination)



Red rect in A also exists in L
- 3 solve the equation since L is now triangular

Convolution

```
for (int i = 0; i < Nx; i += 8)
for (int j = 0; j < Ny; j += 8)
for (int k = 0; k < Nz; k += 8) {

    float inLocal[10][10][10][Di];

    for (int di = -1; di <= 8; di++)
    for (int dj = -1; dj <= 8; dj++)
    for (int dk = -1; dk <= 8; dk++)
        for (int inDim = 0; inDim < Di; inDim++)
            inLocal[di+1][dj+1][dk+1][inDim] = in[i+di][j+dj][k+dk][inDim];

    using inLocal_tensor_array_t  = float (&) [10][10][10][Di];
    inLocal_tensor_array_t inLocalShifted = reinterpret_cast<inLocal_tensor_array_t>(
          inLocal[1][1][1][0]);
    float outLocal[8][8][8][Do] = { 0 };

    localStencilOp<Di,Do>(inLocalShifted, outLocal, stencil);

    for (int di = 0; di < 8; di++)
    for (int dj = 0; dj < 8; dj++)
    for (int dk = 0; dk < 8; dk++)
        for (int outDim = 0; outDim < Do; outDim++)
            out[i+di][j+dj][k+dk][outDim] += outLocal[di][dj][dk][outDim];
}
```

implementation paradigm than we have before! (Although this mimics to a cert degree the "local copies of blocks" appr we saw in GEMM)

Optimized version, each 8x8x8 region of output is computed from a 10x10x10 region of input. localStencil does convolution naively

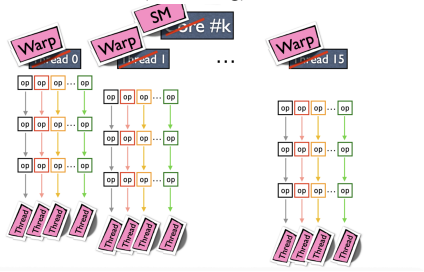
```
using in_tensor_array_t      = float (&) [Nx][Ny][Nz][Di];
using out_tensor_array_t     = float (&) [Nx][Ny][Nz][Do];
using stencil_array_t        = float (&) [3][3][3][Di][Do];

for (int di = -1; di <= 1; di++)
for (int dj = -1; dj <= 1; dj++)
for (int dk = -1; dk <= 1; dk++)
    for (int i = 0; i < 8; i++)
    for (int j = 0; j < 8; j++)
    for (int k = 0; k < 8; k++)
        for (int inDim = 0; inDim < Di; inDim++)
        for (int outDim = 0; outDim < Do; outDim++)
            out[i][j][k][outDim] +=
                stencil[di][dj][dk][inDim][outDim] *
                in[i+di][j+dj][k+dk][inDim];
```

`di,dj,dk` goes in the outer loop so stencil stays in cache as long as possible (`stencil[di][dj][dk]` is const)
Inner 3 loops are effectively GEMM operations on `in[i+di][j+dj] * stencil[di][dj]` → `out[i][j]` (becomes a saxpy equivalent)

Hardware Organization: CPU vs GPU

One core can consist of several threads, per core, one thread runs at a time (scheduling) and share L1-3 caches



GPU:

- Core→**Streaming Multiprocessor (SM)**, tens of SM per GPU with its own cache
- Independent stream→**warp**, each capable of doing 32-wide SIMD instructions
- 1 SM can have up to 32 Warps, warp runs in round robin or scheduler (literally like CPU schedulers)
- 32x32=1024 streams per SM, these are called **threads**
- threads from the same warp are dependent if SIMD is used (cannot execute different instructions)
- using **SIMT**, each thread can run different instructions

```
__global__
void func() {
  int threadID = threadIdx.x; // index [0..1023] of this thread
  int warpID = threadID>>5;   // faster way to do threadID % 32
  if (warpID == 0)
    doFirstWarpStuff();
  else
    doOtherWarpStuff();
}
```

- **thread block** = divide all threads into blockid/threadid

```
__global__
void myKernel(float *array) { // Kernel on GPU
  int threadID = threadIdx.x;
  int blockID = blockIdx.x;
  int elementID = blockID * blockDim.x + threadID;

  array[elementID] += 1.0; }

#define SIZE 1024*1024

void myCPUFunc(){
   float *myArray = ...; // Already in GPU memory
   int blockDim = 512; // thread block size
   int gridDim = SIZE/blockDim;
   myKernel<<<gridDim,blockDim>>>(&myArray[0]);
}
```

- **higher dimension case:**

```
__device__
float GPUfunc(float x) { return x*x*cos(x); }

__global__
void kernel(float (&u)[Nx][Ny],float (&Lu)[Nx][Ny]) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  Lu[i][j] = GPUfunc(u[i][j]);
}

void myCPUFunc(){
   float myU[Nx][Ny] = ...; // Already in GPU memory
   float myLU[Nx][Ny] = ...; // Already in GPU memory
   dim3 blockDim(4,4,1); // use "1" for unused dims
   dim3 gridDim(Nx/4,Ny/4,1);
   Kernel<<<gridDim,blockDim>>>(myU, myLU);
}
```

- `__global__` can be called from anywhere
- `__device__` can be called only from GPU
- `__host__` can be called only from CPU
- only arguments are copied from CPU→GPU

```
cudaMemcpyHostToDevice: Copy CPU->GPU
cudaMemcpyDeviceToHost: Copy GPU->CPU
cudaMemcpyDeviceToDevice: Copy GPU->GPU
```

- `cudaMalloc` and `cudaMemcpy` allocates and copies

ถ้า operation เหมือนกัน ทำพร้อมกัน

卧槽

```
// Device code
__global__ void VecAdd(float* A, float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}

// Host code
int main()
{
    int N = ...;
    size_t size = N * sizeof(float);

    // Allocate input vectors h_A and h_B in host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);
    float* h_C = (float*)malloc(size);

    // Initialize input vectors
    ...

    // Allocate vectors in device memory
    float* d_A;
    cudaMalloc(&d_A, size);
    float* d_B;
    cudaMalloc(&d_B, size);
    float* d_C;
    cudaMalloc(&d_C, size);

    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
```

Inter Thread Communication

Shared memory: user managed cache that allows threads other, there is 64KB of shared memory per SM. Shared between blocks when device code is called

```
#define ROWS 65536
#define COLS 64
// Device code
__global__ void reverseRowsKernel(int (&data)[][COLS])
{
  int threadID = threadIdx.x;
  int rowID = blockIdx.x;
  __shared__ tempData[COLS];
  auto &rowData = data[rowID];
  // type of rowData is int (&)[COLS]
  tempData[threadID] = row[threadID];
  // copy to shared memory
  __syncthreads();
  // make sure all threads have written their part
  row[threadID] = tempData[COLS-1-threadID]; // reverse
}
```

- in the code above, `__syncthreads()` awaits all threads to read to `tempData (on the same BLOCK)`

```
__global__ void sumRowsKernel(int (&data)[][COLS], int (&sums)[ROWS])
{
    int threadID = threadIdx.x;
    int warpID = threadID >> 5;
    int threadInWarpID = threadID & 0x1f; // Number of thread in warp
    int rowID = blockIdx.x;
    constexpr int nWarps = COLS/32; // Warps in block, here nWarps=2

    __shared__ partialSums[nWarps]; // put only partial sums in smem
    auto &rowData = data[rowID]; // type of rowData is int (&)[COLS]
    auto &sum = sums[rowID];

    int partialSum = __reduce_add_sync(0xffffffff, rowData[threadID]);

    partialSums[warpID] = partialSum; // all threads in warp copy the
                                      // same value to shared mem (this is ok)

    __syncthreads(); // make sure all threads have written their part

    if (threadID == 0) {
        sum = 0;
        for (int warp = 0; warp < nWarps; warp++)
            sum += partialSums[warp]; // much less work to add up
    }
}
```

- `__reduce_add_sync` does reduction on the same **WARP**

```
__global__ void sumRowsKernel(int (&data)[][COLS], int &totalSum)
{
    int threadID = threadIdx.x;
    int rowID = blockIdx.x;

    __shared__ tempData[COLS];
    auto &rowData = data[rowID]; // type of rowData is int (&)[COLS]

    tempData[threadID] = row[threadID]; // copy to shared memory

    __syncthreads(); // make sure all threads have written their part

    if (threadID == 0) {
        int blocksum = 0;
        for (int col = 0; col < COLS; col++)
            blockSum += tempData[col];
        atomicAdd(&totalSum, blockSum);
    }
    __syncthreads();
}

// Host code
void myCPUcode()
{
    int data[ROWS][COLS] = ... // Presume this is already on GPU
    int totalSum;  // Presume this is on GPU, and initialized to zero

    sumRowsKernel<<<ROWS,COLS>>>(data,totalSum);
}
```

- `atomicAdd` adds `blocksum` (local) to `totalSum` (shared)

「がんばろうね！」ミクちゃんと言っていました

หีเอยหีอมควยตุ่ย หีเป่าขลุ่ยกะควยเป่าแต

早上好中国 现在我有冰激淋