Examples of Cuda code

1) **The dot product**
2) **Matrix-vector multiplication**
3) **Sparse matrix multiplication**
4) **Global reduction**

Computing y = ax + y with a Serial Loop

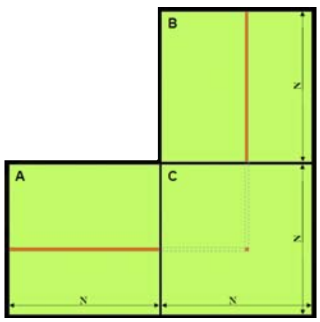
```
void saxpy_serial(int n, float alpha, float *x, float *y)
{
  for(int i = 0; i<n; ++i)
        y[i] = alpha*x[i] + y[i];
}
// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

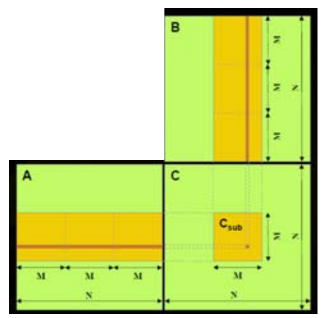
Computing y = ax + y in parallel using CUDA

```
_global_void saxpy_parallel(int n, float alpha, float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if( i<n ) y[i] = alpha*x[i] + y[i];
}
 // Invoke parallel SAXPY kernel (256 threads per block)\\
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

Computing Matrix-vector multiplication in parallel using CUDA

```
__global__ void mm_simple( float* C, float* A, float* B, int n)
{
   int row = blockIdx.y * blockDim.y + threadIdx.y;
   int col = blockIdx.x * blockDim.x + threadIdx.x;
         float sum = 0.0f;
         for (int k = 0; k < n; k++) {
                 sum += A[row*n+k] * B[k * n + col];
         }
         C[row*n+col] = sum;
}
```





Sparse matrix representation

$$A = \begin{bmatrix} 3 & 0 & 9 & 0 & 0 \\ 0 & 5 & 0 & 0 & 2 \\ 0 & 0 & 7 & 0 & 0 \\ 0 & 0 & 5 & 8 & 4 \\ 0 & 0 & 6 & 0 & 0 \end{bmatrix}$$

Av = [3 9 5 2 7 5 8 4 6] = non zero elements

Aj = [0 2 1 4 2 2 3 4 2] = column indices of elements

Ap = [0 2 4 5 8 9] = pointers to the first element in each row

Serial sparse matrix/vector multiplication

```
void csrmul_serial(int *Ap, int *Aj, float *Av, int num_rows,
                        float *x, float *y)
{
  for(int row=0; row<num_rows; ++row)
  {
    int row_begin = Ap[row];
    int row_end = Ap[row+1];
    y[row] = multiply_row(row_end - row_begin,  Aj+row_begin,
       Av+row_begin, x);
  }
}
```

```
float multiply_row(int rowsize,
            int *Aj,          // column indices for row
            float *Av,        // non-zero entries for row
            float *x)         // the RHS vector
{
 float sum = 0;
 for(int column=0; column < rowsize; ++column)
    sum += Av[column] * x[Aj[column]];
 return sum;
}
```

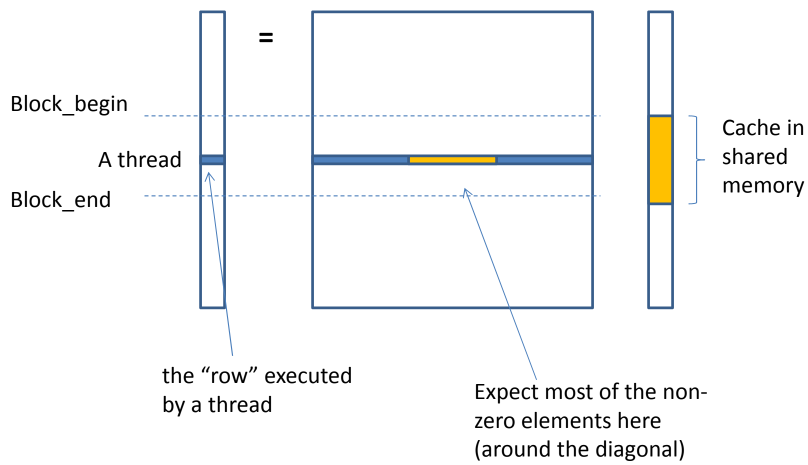
Parallel sparse matrix/vector multiplication

```
_global_void csrmul_kernel(int *Ap, int *Aj, float *Av, int num_rows,
                        float *x, float *y)
{
  int row = blockIdx.x*blockDim.x + threadIdx.x;
  if( row<num_rows )
  {
   int row_begin = Ap[row];
   int row_end = Ap[row+1];
   y[row] = multiply_row(row_end - row_begin, Aj+row_begin,
      Av+row_begin, x);
  }
}
```

The code to launch the above parallel kernel is:

```
unsigned int blocksize = 128; // or any size up to 512
unsigned int nblocks = (num_rows + blocksize - 1) / blocksize;
csrmul_kernel<<<nblocks,blocksize>>>(Ap, Aj, Av, num_rows, x, y);
```

Caching in shared memory




```
_global_void csrmul_cached(int *Ap, int *Aj, float *Av, int num_rows, const float *x, float *y)
{
   _shared_float cache[blocksize];      // Cache the rows of x[] corresponding to this block.
   int block_begin = blockIdx.x * blockDim.x;
   int block_end = block_begin + blockDim.x;
   int row = block_begin + threadIdx.x;
   // Fetch and cache our window of x[].
   if( row<num_rows) cache[threadIdx.x] = x[row];
   _syncthreads();
   if( row<num_rows )
   {
     int row_begin = Ap[row];
     int row_end = Ap[row+1];
      float x_j , sum = 0 ;
      for(int col=row_begin; col<row_end; ++col)
      {
        int j = Aj[col];
        if( j>=block_begin && j<block_end )     // Fetch x_j from our cache when possible
           x_j = cache[j-block_begin];
        else
          x_j = x[j];
        sum += Av[col] * x_j;
      }
     y[row] = sum;
   }
}
```

Parallel reduction

```
_global_void plus_reduce(int *input, int N, int *total)
{
  int tid = threadIdx.x;
  int i = blockIdx.x*blockDim.x + threadIdx.x;

  // Each block loads its elements into shared memory
 _shared_ int x[blocksize];
  x[tid] = (i<N) ? input[i] : 0;                  // last block may pad with 0's
  _syncthreads();

  // Build summation tree over elements.
  for(int s=blockDim.x/2; s>0; s=s/2)
  {
    if(tid < s) x[tid] += x[tid + s];
    _syncthreads();
  }

  // Thread 0 adds the partial sum to the total sum
 if( tid == 0 ) atomicAdd(total, x[tid]);
}
```