libjacket

The World’s Largest, Fastest GPU Library
The Matrix Companion to CUDA
Easy GPU Acceleration in C++

```cpp
#include <stdio.h>
#include <jacket.h>
using namespace jkt;
int main() {
    int n = 20e6; // 20 million random samples
    f32 x = f32::rand(n,1), y = f32::rand(n,1);
    // how many fell inside unit circle?
    float pi = 4.0 * sum_vector(sqrt(x*x + y*y) < 1) / n;
    printf("pi = %g\n", pi);
    return 0;
}
```
Easy GPU Acceleration in C++

```cpp
#include <stdio.h>
#include <jacket.h>
using namespace jkt;

int main() {
    int n = 20e6; // 20 million random samples
    f32 x = f32::rand(n,1), y = f32::rand(n,1);
    // how many fell inside unit circle?
    float pi = 4.0 * sum_vector(sqrt(x*x + y*y) < 1) / n;
    printf("pi = %g\n", pi);
    return 0;
}
```
Easy GPU Acceleration in C++

```cpp
#include <stdio.h>
#include <jacket.h>
using namespace jkt;

int main() {
    int n = 20e6; // 20 million random samples
    f32 x = f32::rand(n, 1), y = f32::rand(n, 1);
    // how many fell inside unit circle?
    float pi = 4.0 * sum_vector(sqrt(x*x + y*y) < 1) / n;
    printf("pi = %g\n", pi);
    return 0;
}
```

On GPU
Matrix Types

- **f32**  
  real single precision

- **f64**  
  real double precision

- **c32**  
  complex single precision

- **c64**  
  complex double precision

- **b8**  
  boolean byte
Matrix Types: ND Support

- **f64**
  - real double precision
  - Vectors
- **f32**
  - real single precision
  - Matrices
- **c32**
  - complex single precision
- **c64**
  - complex double precision
- **b8**
  - boolean byte
- **... ND**

Varying dimensions (ND) support.
Matrix Types: Easy Manipulation

**Jacket Keywords:** \texttt{end, span}

- \texttt{f64} (real double precision)
  - A(span, span, 2)

- \texttt{f32} (real single precision)
  - A(1,1)
  - A(1, span)
  - A(end, 1)
  - A(end, span)

- \texttt{c32} (complex single precision)

- \texttt{c64} (complex double precision)

- \texttt{b8} (boolean byte)
```cpp
#include <stdio.h>
#include <jacket.h>
using namespace jkt;

int main() {
    int n = 20e6; // 20 million random samples
    f32 x = f32::rand(n, 1), y = f32::rand(n, 1);
    // how many fell inside unit circle?
    float pi = 4.0 * sum_vector(sqrt(x*x + y*y) < 1) / n;
    printf("pi = \%g\n", pi);
    return 0;
}
```

Result on CPU
Python, C, and Fortran

# python
import jacket as jkt
b = jkt.rand(4,4)

// C
float *d_x = NULL;
cudaMalloc(&d_x, 64);
jkt_rand_S(d_x, 16);

! Fortran
type(f32)
dst = rand(4,4)
gfor: what is it?

• Data-Parallel for loop, e.g.

Serial matrix multiplications (3 kernel launches)

```matlab
for i = 1:3
    C(:, :, i) = A(:, :, i) * B;
end
```

Parallel matrix multiplications (1 kernel launch)

```matlab
gfor i = 1:3
    C(:, :, i) = A(:, :, i) * B;
end
```
example: matrix multiply

- Data-Parallel for loop, e.g.

```
for i = 1:3
    C(:, :, i) = A(:, :, i) * B;
```

iteration i = 1

\[
\begin{align*}
C(:, :, i) &= \begin{bmatrix}
\end{align*}
\begin{align*}
A(:, :, i) &\times \begin{bmatrix}
\end{align*}
\begin{align*}
B
\end{align*}
\]
example: matrix multiply

- Data-Parallel for loop, e.g.

  Serial matrix multiplications (3 kernel launches)

  for i = 1:3
  \[
  C(:,:,i) = A(:,:,i) * B;
  \]

  iteration i = 1
  
  \[
  \begin{array}{c}
  C(:,:,i) = A(:,:,i) * B
  \end{array}
  \]

  iteration i = 2
  
  \[
  \begin{array}{c}
  C(:,:,i) = A(:,:,i) * B
  \end{array}
  \]
example: matrix multiply

• Data-Parallel for loop, e.g.

Serial matrix multiplications (3 kernel launches)

\[
\text{for } i = 1:3 \\
C(:, :, i) = A(:, :, i) \times B;
\]
example: matrix multiply

Parallel matrix multiplications (1 kernel launch)

```matlab
for i = 1:3
    C(:,:,i) = A(:,:,i) * B;
end
```

simultaneous iterations $i = 1:3$
example: matrix multiply

Parallel matrix multiplications (1 kernel launch)

gfor i = 1:3
C(:, :, i) = A(:, :, i) * B;

Think “gfor i=1:3” as “extend 1:3 in the next dimension”

simultaneous iterations i = 1:3
example: matrix multiply

Parallel matrix multiplications (1 kernel launch)

gfor i = 1:3
    C(:,:,:i) = A(:,:,:i) * B;

Think “gfor i=1:3” as “extend 1:3 in the next dimension”

simultaneous iterations i = 1:3

C(:,:,1:3) = A(:,:,1:3) * B
example: matrix multiply

Parallel matrix multiplications (1 kernel launch)

```matlab
for i = 1:3
    C(:,:,i) = A(:,:,i) * B;
end
```

Think “gfor i=1:3” as “extend 1:3 in the next dimension”

simultaneous iterations i = 1:3
example: printing to screen

- gfor runs its body only once on the CPU

```
Loop body executed three times
for i = 1:3
    disp(i);
```

```
Loop body executed one time
for i = 1:3
    disp(i);
```
example: printing to screen

- gfor runs its body only once on the CPU

```
for i = 1:3
    disp(i);
end
```
Loop body executed three times

```
gfor i = 1:3
    disp(i);
end
```
Loop body executed one time
example: summing over columns

• Think of gfor as “syntactic sugar” to write vectorized code in an iterative style.

```matlab
for i = 1:3
    A(i) = sum(B(:,i));
end
```

Three passes to sum all columns of B

```matlab
gfor i = 1:3
    A(i) = sum(B(:,i));
end
```

One pass to sum all columns of B

Both equivalent to “`sum(B)`”, but latter is faster (more explicitly written)
Easy Multi GPU Scaling

f32 *y = new f32[n];
for (int i = 0; i < n; ++i) {
    device(i);                // change GPUs
    f32 x = f32::rand(5,5);   // add work to GPU’s queue
    y[i] = fft(x);            // more work in queue
}

// all GPUs are now computing simultaneously, until done
Hundreds of Functions...

- **reductions**
  - sum, min, max, any, all, nnz, prod
  - vectors, columns, rows, etc

- **convolutions**
  - 2D, 3D, ND

- **dense linear algebra**
  - LU, QR, Cholesky, SVD, Eigenvalues, Inversion, det, Matrix Power, Solvers

- **FFTs**
  - 2D, 3D, ND

- **image processing**
  - filter, rotate, erode, dilate, bwmorph, resize, rgb2gray
  - hist, histeq

- **interp and rescale**
  - vectors, matrices
  - rescaling

- **sorting**
  - along any dimension
  - sort detection

and many more...
...Plenty of Speed

Matrix Multiply

SUM

GFLOPS vs. Edge Length of Square Matrix

GFLOPS vs. Vector Length
...Plenty of Speed

**CONV**

![Graph showing data points with labels: Signal Length (2^N), with 10k element kernel length.](image)

**CONV2**

![Graph showing data points with labels: Edge Length of Square Matrix (25x25 kernel).](image)

Legend:
- LIBJACKET
- MKL
- IPP
Plenty of Speed

Cumulative Sum

- MFLOPS vs. Vector Length
- LIBJACKET and ARMADILLO

SORT

- Million Elements per Second vs. Vector Length
- LIBJACKET, MKL, IPP, ARMADILLO
LIBJACKET eliminates hidden costs associated with algorithm development and support:

- Save up to 95% of the time and cost required to research and develop algorithms
- Free up your developers’ in-house application development.
Easy To Maintain

• Write your code once and let LibJacket carry you through the coming hardware evolution.
  – Each new LibJacket release improves the speed of your code, without any code modification.
  – Each new LibJacket release leverages latest GPU hardware (e.g. Fermi), without any code modification.
Get *libjacket*

The World’s Largest, Fastest GPU Library

www.accelereyes.com