libjacket

The World's Largest, Fastest GPU Library

The Matrix Companion to CUDA

libjacket

CUDA

CUBLAS | **CUFFT** | **Kernels**

Easy GPU Acceleration in C++

```
#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++

```
#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++

```
#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

f64
real double precision

f32
real single precision

b8
boolean byte

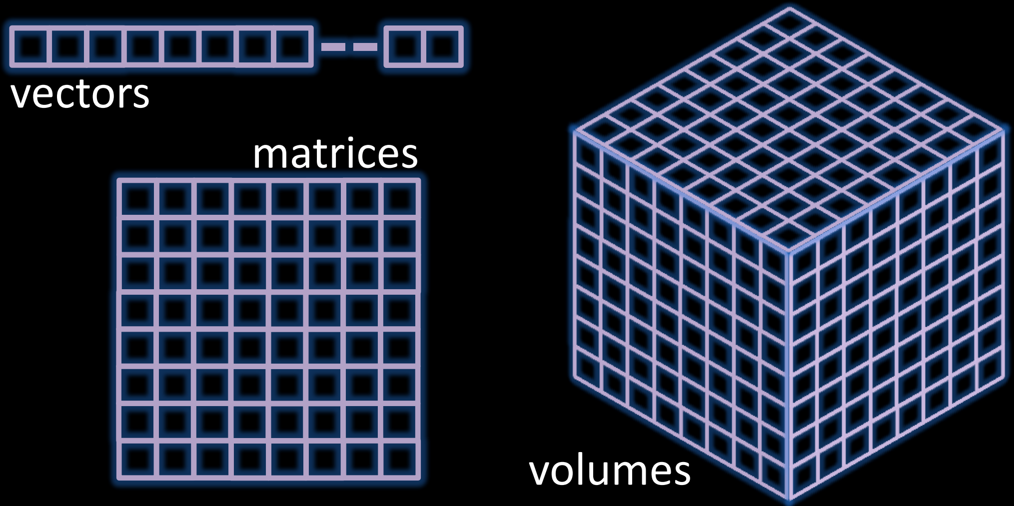
c32
complex single precision

c64
complex double precision

Matrix Types: ND Support

f64
real double precision

f32
real single precision



b8
boolean byte

... ND

c32
complex single precision

c64
complex double precision

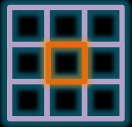
Matrix Types: Easy Manipulation

f64
real double precision

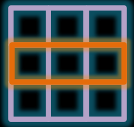
Jacket Keywords: **end, span**

A(span,span,2)

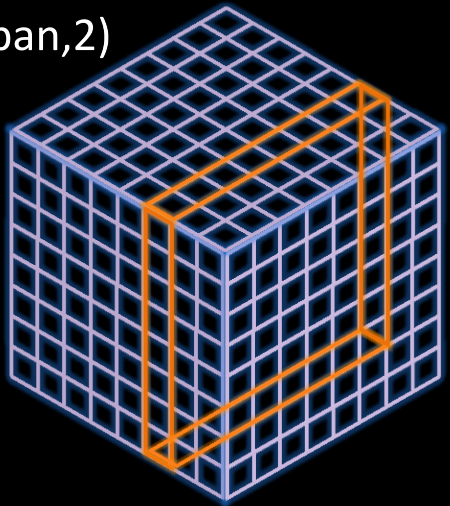
A(1,1)



A(1,span)

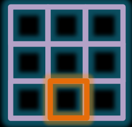


f32
real single precision

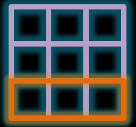


b8
boolean byte

A(end,1)



A(end,span)



c32
complex single precision

c64
complex double precision

Easy GPU Acceleration in C++

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

```fortran
! Fortran
type(f32)
dst = rand(4,4)
```

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

gfor: what is it?

- Data-Parallel for loop, e.g.

Serial matrix multiplications (3 kernel launches)

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

Parallel matrix multiplications (**1** kernel launch)

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

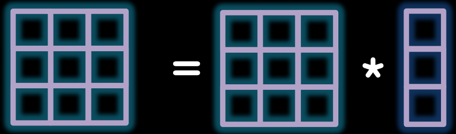
example: matrix multiply

- Data-Parallel for loop, e.g.

Serial matrix multiplications (3 kernel launches)

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

iteration i = 1



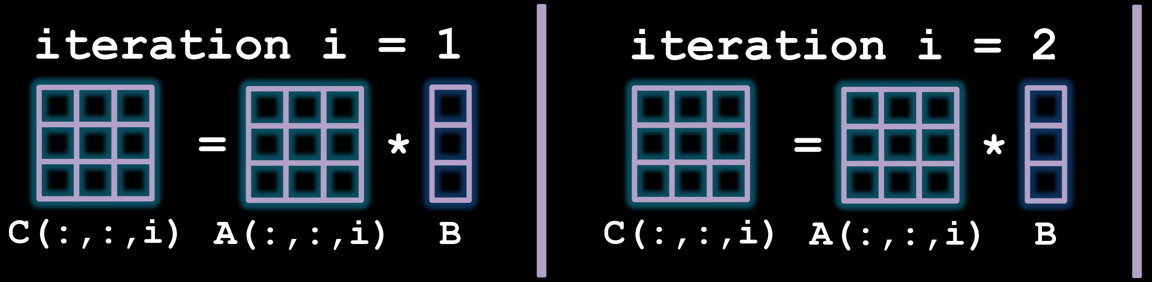
`C(:,:,i)` `A(:,:,i)` `B`

example: matrix multiply

- Data-Parallel for loop, e.g.

Serial matrix multiplications (3 kernel launches)

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

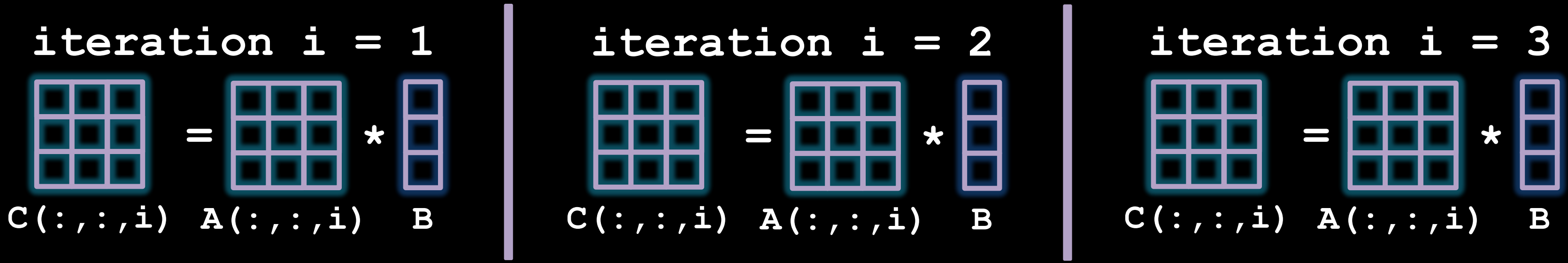


example: matrix multiply

- Data-Parallel for loop, e.g.

Serial matrix multiplications (3 kernel launches)

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

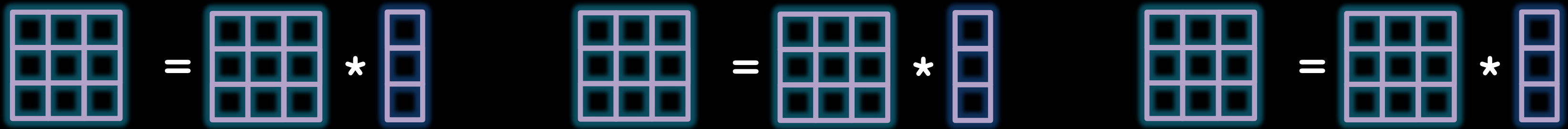


example: matrix multiply

Parallel matrix multiplications (**1** kernel launch)

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

simultaneous iterations i = 1:3



C(:,:,1) A(:,:,1) B   C(:,:,2) A(:,:,2) B   C(:,:,3) A(:,:,3) B

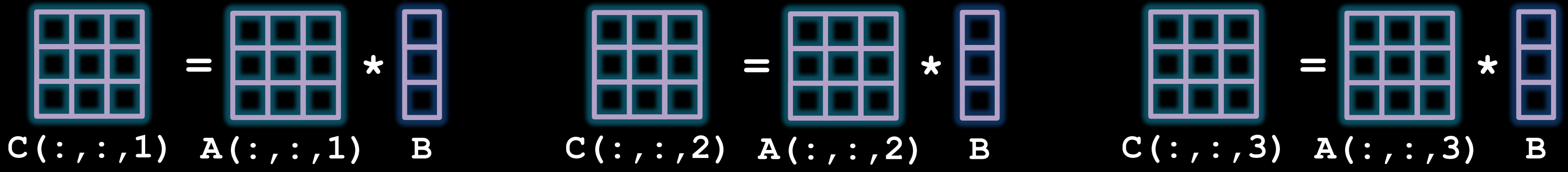
example: matrix multiply

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`



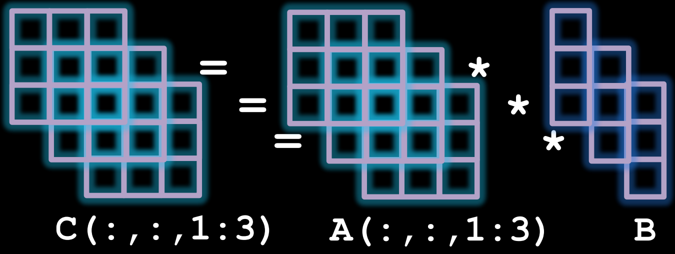
example: matrix multiply

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`



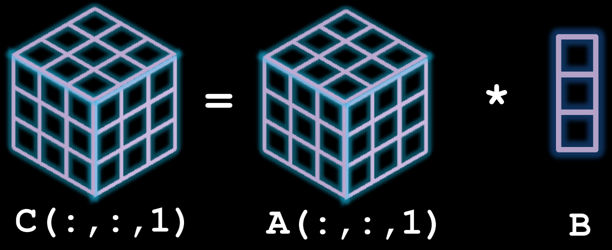
example: matrix multiply

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: 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

```
gfor i = 1:3
    disp(i);
```

example: printing to screen

- gfor runs its body only once on the CPU

Loop body executed three times

```
for i = 1:3
    disp(i);
```

```
>> go;
1
2
3
```

Loop body executed one time

```
gfor i = 1:3
    disp(i);
```

```
>> go;
1
```

example: summing over columns

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

Three passes to sum all columns of B

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

One pass to sum all columns of B

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

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

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

convolutions
- 2D, 3D, ND

FFTs
- 2D, 3D, ND

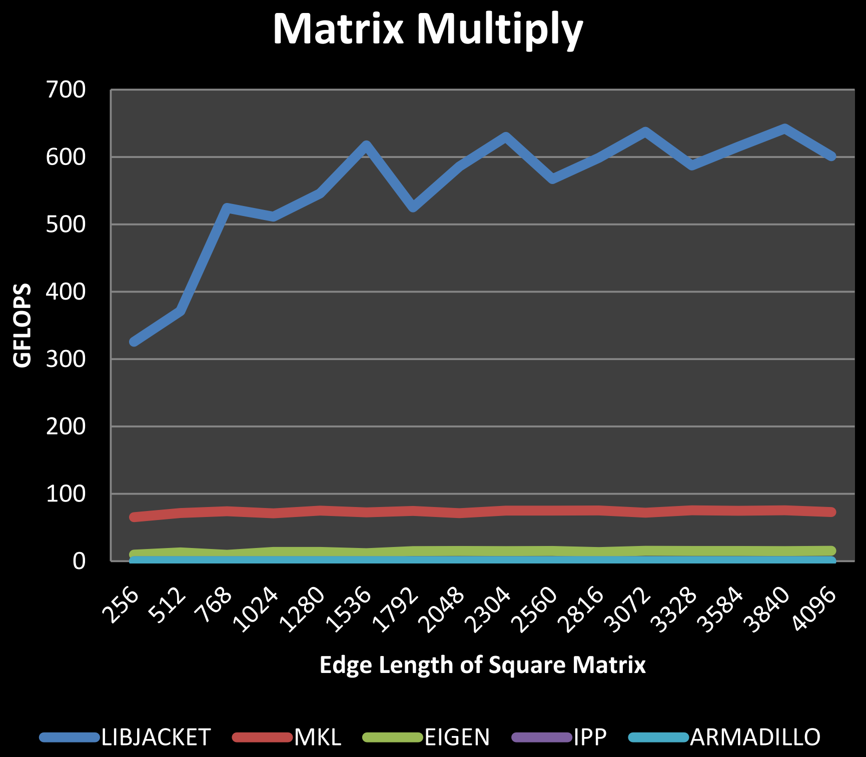
image processing
- filter, rotate, erode, dilate, bwmorph, resize, rgb2gray
- hist, histeq

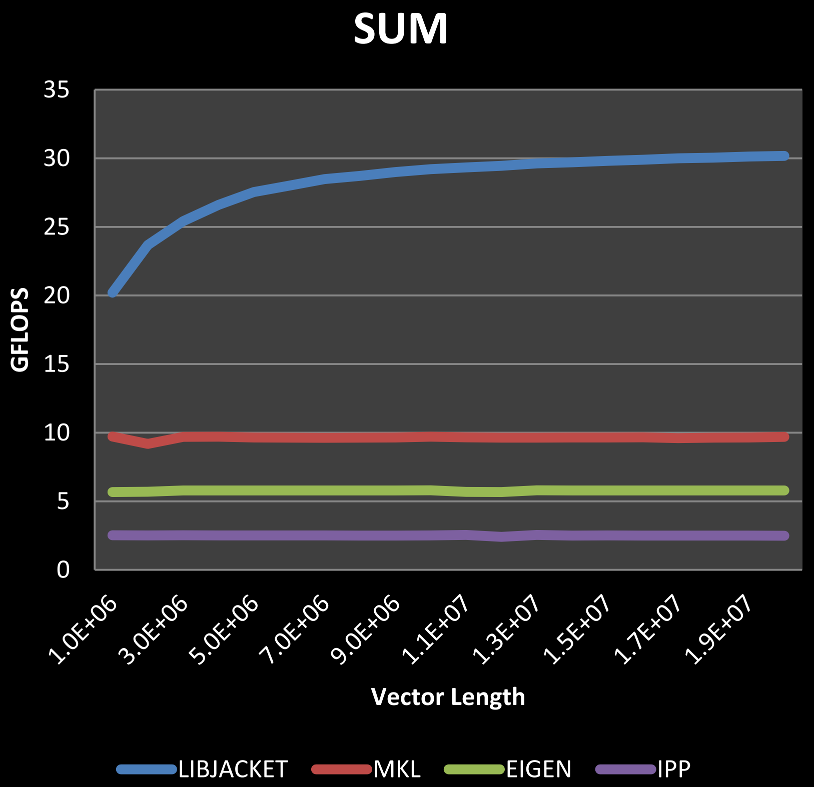
interp and rescale
- vectors, matrices
- rescaling

sorting
- along any dimension
- sort detection

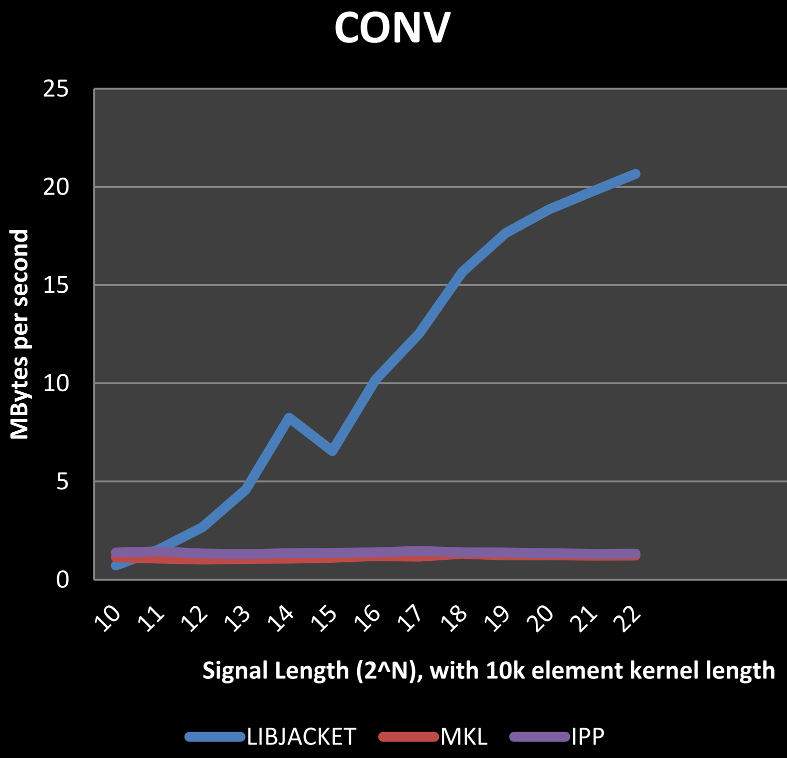
and many more…

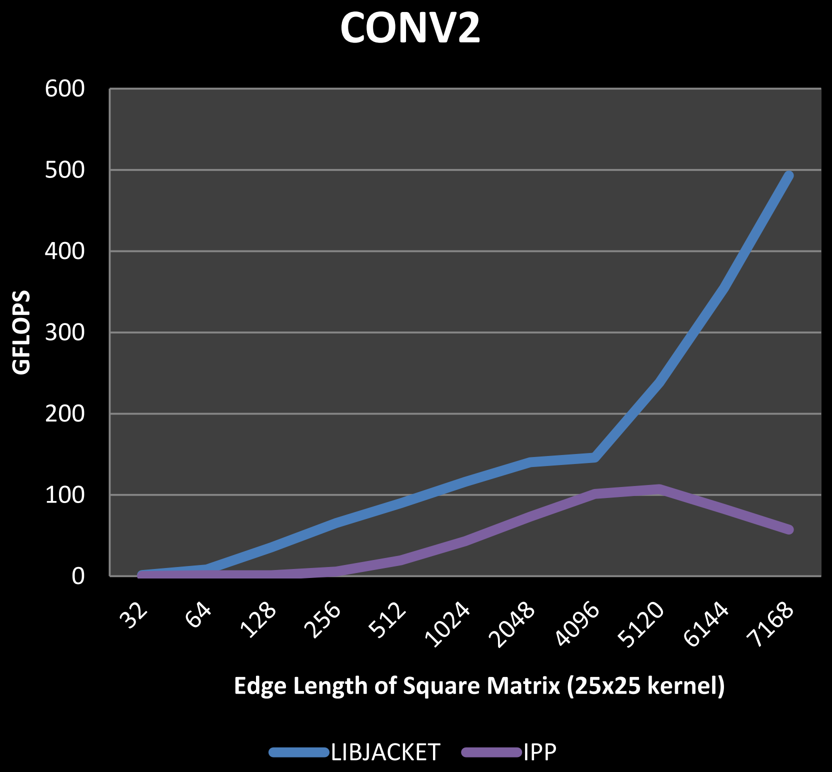
…Plenty of Speed



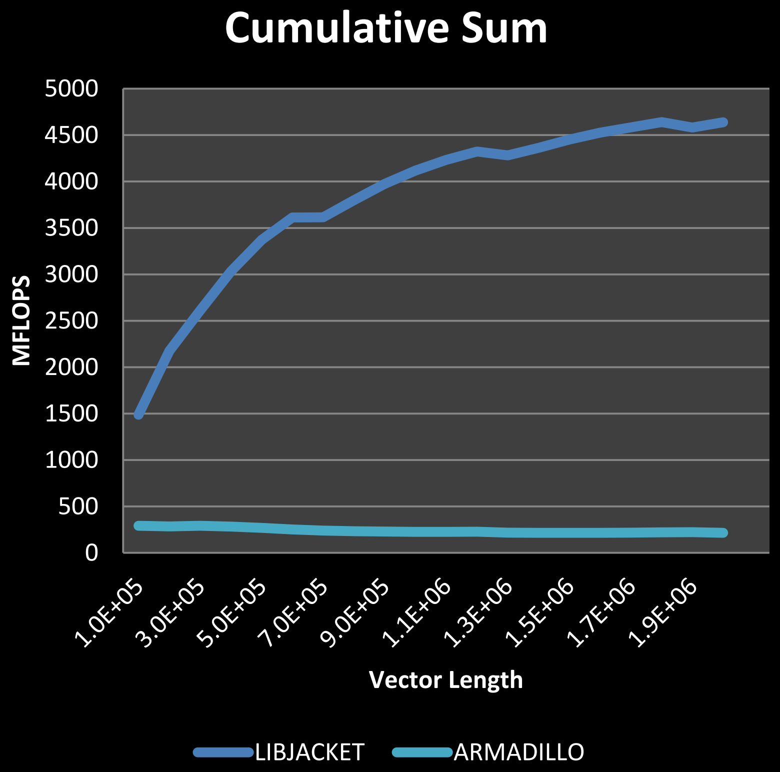


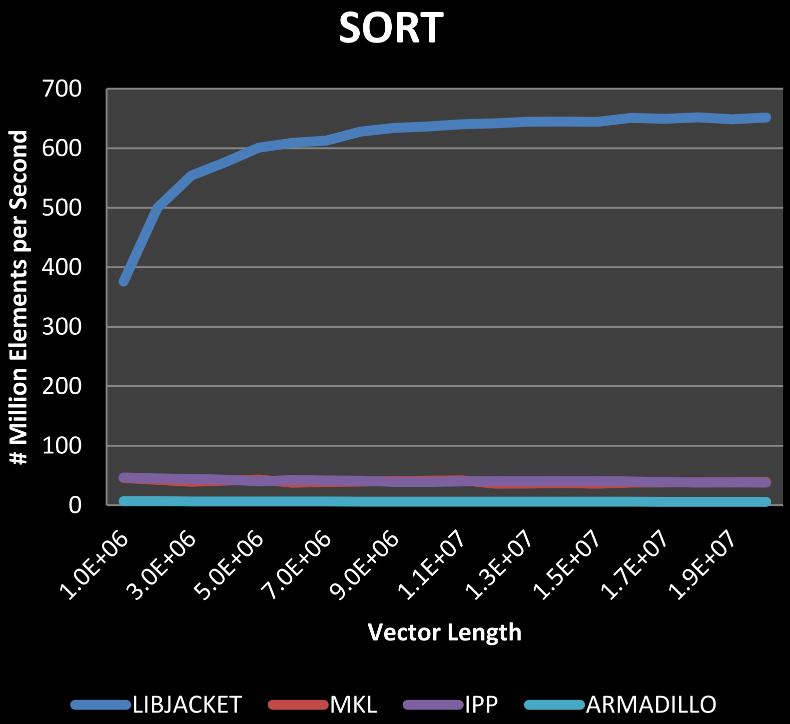
…Plenty of Speed



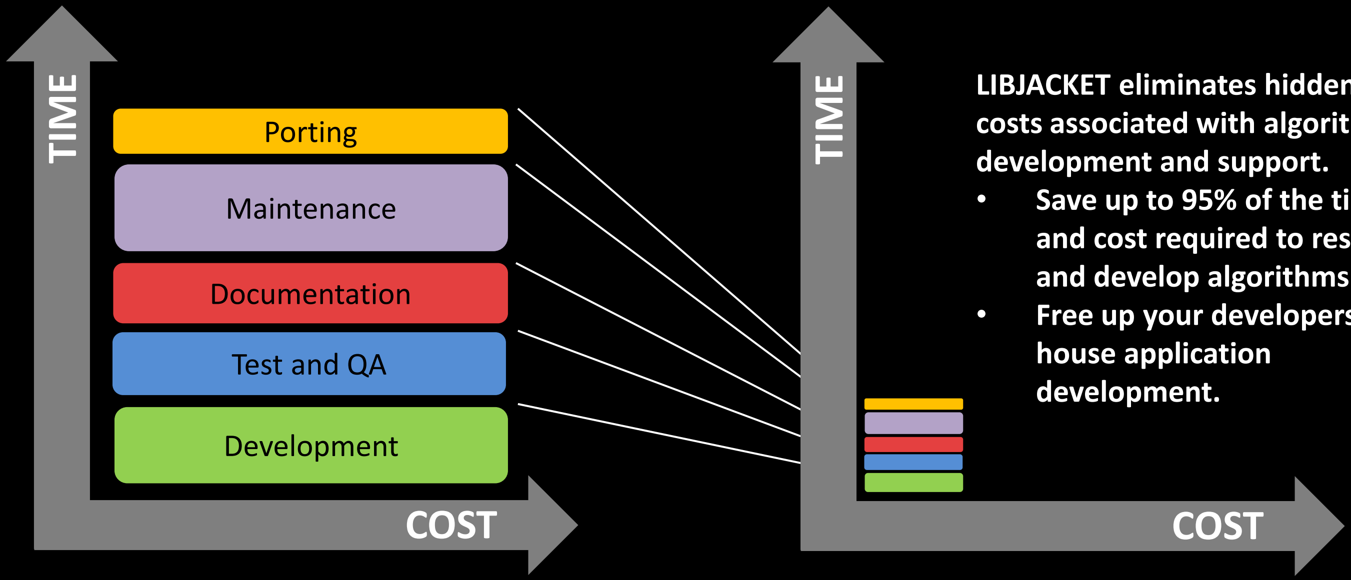


…Plenty of Speed





Build vs Buy



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