Fast, real-time sound processing on the GPU

Max Tandetzky

FSU Jena

2010-02-12
Fast, real-time sound processing on the GPU

Max Tandetzky

JACK

CUDA FFT-libraries

Programmes

Performance

Problems

Further development
Fast, real-time sound processing on the GPU

Max Tandetzky

JACK

CUDA FFT-libraries

Programmes

Performance

Problems

Further development

What is JACK

- JACK stands for JACK Audio Connection Kit
- there is a server which manages the connections → jackd
- allows different applications to share signal data
- data can be any stream of floats, not only audio data

JACK API

- JACK’s own data type for signal data is usually a typeset to float
- it is designed to typeset to double
What is JACK

- JACK stands for JACK Audio Connection Kit
  - there is a server which manages the connections → jackd
  - allows different applications to share signal data
  - data can be any stream of floats, not only audio data

JACK API

- JACK’s own data type for signal data is usually a typeset to float
- it is designed to typeset to double
What is JACK

- JACK stands for JACK Audio Connection Kit
- there is a server which manages the connections → jackd
  - allows different applications to share signal data
  - data can be any stream of floats, not only audio data

JACK API

- JACK’s own data type for signal data is usually a typeset to float
- it is designed to typeset to double
What is JACK

- JACK stands for JACK Audio Connection Kit
- there is a server which manages the connections → jackd
- allows different applications to share signal data
- data can be any stream of floats, not only audio data

JACK API

- JACK’s own data type for signal data is usually a typeset to float
- it is designed to typeset to double
What is JACK

- JACK stands for JACK Audio Connection Kit
- there is a server which manages the connections → jackd
- allows different applications to share signal data
- data can be any stream of floats, not only audio data

JACK API

- JACK’s own data type for signal data is usually a typeset to float
- it is designed to typeset to double
What is JACK

- JACK stands for JACK Audio Connection Kit
- there is a server which manages the connections → jackd
- allows different applications to share signal data
- data can be any stream of floats, not only audio data

JACK API

- JACK’s own data type for signal data is usually a typeset to float
- it is designed to typeset to double
What is JACK

- JACK stands for JACK Audio Connection Kit
- there is a server which manages the connections → jackd
- allows different applications to share signal data
- data can be any stream of floats, not only audio data

JACK API

- JACK’s own data type for signal data is usually a typeset to float
- it is designed to typeset to double
What is JACK

- JACK stands for JACK Audio Connection Kit
- there is a server which manages the connections → jackd
- allows different applications to share signal data
- data can be any stream of floats, not only audio data

JACK API

- JACK’s own data type for signal data is usually a typeset to float
- it is designed to typeset to double
JACK API

- there are special functions which will be called by jackd when some events take place, for example:
  - process()
    - will be called when there is any work to do
    - is responsible for dealing the data which comes in / goes out
  - further special functions which are called when for example
    - errors occur
    - sample rate of jackd changes
    - jack server shuts down
there are special functions which will be called by jackd when some events take place, for example:

- **process()**
  - will be called when there is any work to do
  - is responsible for dealing the data which comes in / goes out

- further special functions which are called when for example
  - errors occur
  - sample rate of jackd changes
  - jack server shuts down
JACK API

- there are special functions which will be called by jackd when some events take place, for example:
  - `process()`
    - will be called when there is any work to do
    - is responsible for dealing the data which comes in / goes out
  - further special functions which are called when for example
    - errors occur
    - sample rate of jackd changes
    - jack server shuts down
Fast, real-time sound processing on the GPU

Max Tandetzky

JACK

CUDA

FFT-libraries

Programmes

Performance

Problems

Further development

JACK

there are special functions which will be called by jackd when some events take place, for example:

- process()
  - will be called when there is any work to do
  - is responsible for dealing the data which comes in / goes out

- further special functions which are called when for example
  - errors occur
  - sample rate of jackd changes
  - jack server shuts down
Fast, real-time sound processing on the GPU

Max Tandetzky

JACK

CUDA FFT-libraries

Programmes

Performance

Problems

Further development

JACK API

- there are special functions which will be called by jackd when some events take place, for example:
  - process()
    - will be called when there is any work to do
    - is responsible for dealing the data which comes in / goes out
  - further special functions which are called when for example
    - errors occur
    - sample rate of jackd changes
    - jack server shuts down
CUDA FFT-libraries

General

- compute a DFT of any given data – regardless of the number of supporting points (here: amount of samples)
- when the number of samples is a power of a small prime the FFT will be fastest
- allows multiple transformations in parallel

API

- malloc and memcopy must be done manually
- you must create so called plans
- the library will figure out the best configuration → no explicit kernel call
- forward transformation produces always complex numbers
CUDA FFT-libraries

General

- compute a DFT of any given data – regardless of the number of supporting points (here: amount of samples)
- when the number of samples is a power of a small prime the FFT will be fastest
- allows multiple transformations in parallel

API

- malloc and memcopy must be done manually
- you must create so called plans
- the library will figure out the best configuration → no explicit kernel call
- forward transformation produces always complex numbers
CUDA FFT-libraries

**General**

- compute a DFT of any given data – regardless of the number of supporting points (here: amount of samples)
- when the number of samples is a power of a small prime the FFT will be fastest
- allows multiple transformations in parallel

**API**

- malloc and memcopy must be done manually
- you must create so called plans
- the library will figure out the best configuration → no explicit kernel call
- forward transformation produces always complex numbers
CUDA FFT-libraries

General

- compute a DFT of any given data – regardless of the number of supporting points (here: amount of samples)
- when the number of samples is a power of a small prime the FFT will be fastest
- allows multiple transformations in parallel

API

- malloc and memcopy must be done manually
- you must create so called plans
- the library will figure out the best configuration → no explicit kernel call
- forward transformation produces always complex numbers
CUDA FFT-libraries

**General**

- compute a DFT of any given data – regardless of the number of supporting points (here: amount of samples)
- when the number of samples is a power of a small prime the FFT will be fastest
- allows multiple transformations in parallel

**API**

- malloc and memcopy must be done manually
- you must create so called plans
- the library will figure out the best configuration → no explicit kernel call
- forward transformation produces always complex numbers
CUDA FFT-libraries

General

- compute a DFT of any given data – regardless of the number of supporting points (here: amount of samples)
- when the number of samples is a power of a small prime the FFT will be fastest
- allows multiple transformations in parallel

API

- malloc and memcopy must be done manually
  - you must create so called plans
  - the library will figure out the best configuration → no explicit kernel call
  - forward transformation produces always complex numbers
CUDA FFT-libraries

**General**
- compute a DFT of any given data – regardless of the number of supporting points (here: amount of samples)
- when the number of samples is a power of a small prime the FFT will be fastest
- allows multiple transformations in parallel

**API**
- malloc and memcopy must be done manually
- you must create so called plans
  - the library will figure out the best configuration → no explicit kernel call
- forward transformation produces always complex numbers
CUDA FFT-libraries

General

- compute a DFT of any given data – regardless of the number of supporting points (here: amount of samples)
- when the number of samples is a power of a small prime the FFT will be fastest
- allows multiple transformations in parallel

API

- malloc and memcopy must be done manually
- you must create so called plans
- the library will figure out the best configuration → no explicit kernel call
- forward transformation produces always complex numbers
CUDA FFT-libraries

**General**

- compute a DFT of any given data – regardless of the number of supporting points (here: amount of samples)
- when the number of samples is a power of a small prime the FFT will be fastest
- allows multiple transformations in parallel

**API**

- malloc and memcopy must be done manually
- you must create so called plans
- the library will figure out the best configuration → no explicit kernel call
- forward transformation produces always complex numbers
Programmes

List of programmes

- **audio mixer**
  - has 2 input and 1 output port
  - reduces the intensity of each incoming signal to the half
  → output is the sum of this values

- **sound generator**
  - generates sinus waves
  - frequency can be changed while the programme runs (as a multiple of a fundamental frequency)

- **audio data manipulation programm**
  - when standard settings: output = input
  - while programme runs user can change behavior
    - adjust loudness of the signal
    - enable a (very) simple equalizer (5 kHz filter)
  - uses CUDA and CUDA-FFT functions
Programmes

List of programmes

- audio mixer
  - has 2 input and 1 output port
  - reduces the intensity of each incoming signal to the half
  → output is the sum of this values

- sound generator
  - generates sinus waves
  - frequency can be changed while the programme runs (as a multiple of a fundamental frequency)

- audio data manipulation programm
  - when standard settings: output = input
  - while programme runs user can change behavior
    - adjust loudness of the signal
    - enable a (very) simple equalizer (5 kHz filter)
  - uses CUDA and CUDA-FFT functions
Programmes

List of programmes

- audio mixer
  - has 2 input and 1 output port
  - reduces the intensity of each incoming signal to the half
    → output is the sum of this values

- sound generator
  - generates sinus waves
  - frequency can be changed while the programme runs (as a multiple of a fundamental frequency)

- audio data manipulation programm
  - when standard settings: output = input
  - while programme runs user can change behavior
    - adjust loudness of the signal
    - enable a (very) simple equalizer (5 kHz filter)
  - uses CUDA and CUDA-FFT functions
Programmes

List of programmes

- audio mixer
  - has 2 input and 1 output port
  - reduces the intensity of each incoming signal to the half
    → output is the sum of this values

- sound generator
  - generates sinus waves
  - frequency can be changed while the programme runs (as a multiple of a fundamental frequency)

- audio data manipulation programme
  - when standard settings: output = input
  - while programme runs user can change behavior
    - adjust loudness of the signal
    - enable a (very) simple equalizer (5 kHz filter)
  - uses CUDA and CUDA-FFT functions
Programmes

List of programmes

- **audio mixer**
  - has 2 input and 1 output port
  - reduces the intensity of each incoming signal to the half
    → output is the sum of this values

- **sound generator**
  - generates sinus waves
  - frequency can be changed while the programme runs (as a multiple of a fundamental frequency)

- **audio data manipulation programme**
  - when standard settings: output = input
  - while programme runs user can change behavior
    - adjust loudness of the signal
    - enable a (very) simple equalizer (5 kHz filter)

- uses CUDA and CUDA-FFT functions
Programmes

List of programmes

- **audio mixer**
  - has 2 input and 1 output port
  - reduces the intensity of each incoming signal to the half
  → output is the sum of this values

- **sound generator**
  - generates sinus waves
  - frequency can be changed while the programme runs (as a multiple of a fundamental frequency)

- **audio data manipulation programm**
  - when standard settings: output = input
  - while programme runs user can change behavior
    - adjust loudness of the signal
    - enable a (very) simple equalizer (5 kHz filter)

- uses CUDA and CUDA-FFT functions
Performance

- implementation with CUDA:
  - average processing time:
    - about 0.04 ms without equalizer
    - between 0.1 ms and 0.2 ms with enabled equalizer
  - maximum processing time:
    - 0.6 ms with enabled equalizer
    - 0.5 ms without equalizer

- implementation on CPU with FFTW (Fastest Fourier Transform in the West)
  - average: about 0.3 ms with equalizer
  - maximum: about 10 ms with equalizer (malloc in process)
  - values for CPU implementation not very convincing

- performance on GPU will rise a lot when more than 1 channel is computed at the same time
Performance

- **implementation with CUDA:**
  - average processing time:
    - about 0.04 ms without equalizer
    - between 0.1 ms and 0.2 ms with enabled equalizer
  - maximum processing time:
    - 0.6 ms with enabled equalizer
    - 0.5 ms without equalizer

- **implementation on CPU with FFTW (Fastest Fourier Transform in the West)**
  - average: about 0.3 ms with equalizer
  - maximum: about 10 ms with equalizer (malloc in process)
  - values for CPU implementation not very convincing

- performance on GPU will rise a lot when more than 1 channel is computed at the same time
Performance

- **implementation with CUDA:**
  - average processing time:
    - about 0.04 ms without equalizer
    - between 0.1 ms and 0.2 ms with enabled equalizer
  - maximum processing time:
    - 0.6 ms with enabled equalizer
    - 0.5 ms without equalizer

- **implementation on CPU with FFTW (Fastest Fourier Transform in the West):**
  - average: about 0.3 ms with equalizer
  - maximum: about 10 ms with equalizer (malloc in process)
  - values for CPU implementation not very convincing

- performance on GPU will rise a lot when more than 1 channel is computed at the same time
Performance

- implementation with CUDA:
  - average processing time:
    - about 0.04 ms without equalizer
    - between 0.1 ms and 0.2 ms with enabled equalizer
  - maximum processing time:
    - 0.6 ms with enabled equalizer
    - 0.5 ms without equalizer

- implementation on CPU with FFTW (Fastest Fourier Transform in the West)
  - average: about 0.3 ms with equalizer
  - maximum: about 10 ms with equalizer (malloc in process)
  - values for CPU implementation not very convincing

- performance on GPU will rise a lot when more than 1 channel is computed at the same time
Performance

- implementation with CUDA:
  - average processing time:
    - about 0,04 ms without equalizer
    - between 0,1 ms and 0,2 ms with enabled equalizer
  - maximum processing time:
    - 0,6 ms with enabled equalizer
    - 0,5ms without equalizer

- implementation on CPU with FFTW (Fastest Fourier Transform in the West)
  - average: about 0,3ms with equalizer
  - maximum: about 10ms with equalizer (malloc in process)
  - values for CPU implementation not very convincing

- performance on GPU will rise a lot when more than 1 channel is computed at the same time
Performance

- implementation with CUDA:
  - average processing time:
    - about 0.04 ms without equalizer
    - between 0.1 ms and 0.2 ms with enabled equalizer
  - maximum processing time:
    - 0.6 ms with enabled equalizer
    - 0.5 ms without equalizer

- implementation on CPU with FFTW (Fastest Fourier Transform in the West)
  - average: about 0.3 ms with equalizer
  - maximum: about 10 ms with equalizer (malloc in process)
    - values for CPU implementation not very convincing

- performance on GPU will rise a lot when more than 1 channel is computed at the same time
Performance

- implementation with CUDA:
  - average processing time:
    - about 0.04 ms without equalizer
    - between 0.1 ms and 0.2 ms with enabled equalizer
  - maximum processing time:
    - 0.6 ms with enabled equalizer
    - 0.5 ms without equalizer

- implementation on CPU with FFTW (Fastest Fourier Transform in the West)
  - average: about 0.3 ms with equalizer
  - maximum: about 10 ms with equalizer (malloc in process)
  - values for CPU implementation not very convincing

- performance on GPU will rise a lot when more than 1 channel is computed at the same time
Performance

- implementation with CUDA:
  - average processing time:
    - about 0.04 ms without equalizer
    - between 0.1 ms and 0.2 ms with enabled equalizer
  - maximum processing time:
    - 0.6 ms with enabled equalizer
    - 0.5ms without equalizer

- implementation on CPU with FFTW (Fastest Fourier Transform in the West)
  - average: about 0.3ms with equalizer
  - maximum: about 10ms with equalizer (malloc in process)
  - values for CPU implementation not very convincing

- performance on GPU will rise a lot when more than 1 channel is computed at the same time
List of problems

- result of cufft is not normalized (no hint in documentation of cufft but in documentation of FFTW)
- allocate memory on the device outside process()-function
Problems

List of problems

- result of cufft is not normalized (no hint in documentation of cufft but in documentation of FFTW)
- allocate memory on the device outside process()-function
Problems

List of problems

- result of cufft is not normalized (no hint in documentation of cufft but in documentation of FFTW)
- allocate memory on the device outside process()-function
Further development

Programmes will not be used as they are → more useful functions will follow
- use more than 1 channel (mono)
- sound synthesis (with up to 64 signals simultaneous)
- make use of effects
- compute (many) general DSP operations very fast simultaneously
Further development

Programmes will not be used as they are → more useful functions will follow

- use more than 1 channel (mono)
- sound synthesis (with up to 64 signals simultaneous)
- make use of effects
- compute (many) general DSP operations very fast simultaneously
Further development

- Programmes will not be used as they are → more useful functions will follow
  - use more than 1 channel (mono)
  - sound synthesis (with up to 64 signals simultaneous)
  - make use of effects
  - compute (many) general DSP operations very fast simultaneously
Further development

- Programmes will not be used as they are → more useful functions will follow
  - use more than 1 channel (mono)
  - sound synthesis (with up to 64 signals simultaneous)
  - make use of effects
  - compute (many) general DSP operations very fast simultaneously
Further development

Programmes will not be used as they are → more useful functions will follow

- use more than 1 channel (mono)
- sound synthesis (with up to 64 signals simultaneous)
- make use of effects
- compute (many) general DSP operations very fast simultaneously
Further development

- Programmes will not be used as they are → more useful functions will follow
  - use more than 1 channel (mono)
  - sound synthesis (with up to 64 signals simultaneous)
  - make use of effects
  - compute (many) general DSP operations very fast simultaneously
Fast, real-time sound processing on the GPU

Max Tandetzky

JACK
CUDA
FFT-libraries
Programmes
Performance
Problems
Further development